

Web security fundamentals

HTTP basics, cookies vs sessions vs tokens, JWT, and every major authentication method

A study reference for learning web request internals and modern authentication, built for hands-on practice with tools like Burp Suite.

Brought to you by

contentintech.com | Instagram **@contentintech**

1. HTTP methods

HTTP methods describe the intent of a request — what the client wants to do with a resource. Idempotent means calling it once or many times has the same effect on server state; safe means it does not change server state at all.

Method	Purpose	Body?	Idempotent?	Safe?
GET	Retrieve a resource	No	Yes	Yes
POST	Create a resource or submit data	Yes	No	No
PUT	Replace a resource entirely	Yes	Yes	No
PATCH	Partially update a resource	Yes	No (usually)	No
DELETE	Remove a resource	Sometimes	Yes	No
OPTIONS	Ask what methods/headers are allowed	No	Yes	Yes

Example: DELETE is idempotent because deleting an already-deleted resource still ends in the same state, it is gone. POST usually is not idempotent, since submitting the same order form twice creates two orders.

2. HTTP status codes

Status codes are grouped by their first digit, which tells you the category of outcome at a glance.

Range	Meaning	Common codes
2xx	Success	200 OK, 201 Created, 204 No content
3xx	Redirect	301 Permanent, 302 Temporary, 304 Not modified
4xx	Client error	400 Bad request, 401 Unauthorized, 403 Forbidden, 404 Not found, 429 Rate limited
5xx	Server error	500 Internal error, 502 Bad gateway, 503 Unavailable

For security testing, 401 versus 403 matters: 401 means no valid credentials were presented at all, while 403 means the credentials were valid but the account lacks permission for that resource.

3. Request and response headers

Headers are metadata sent alongside a request or response body. They tell the other side how to handle what is coming, without needing to inspect the body itself.

Common response headers

Header	Meaning
Content-Type	What kind of data this is, for example application/json or text/html
Content-Length	Size of the body in bytes
Set-Cookie	Tells the browser to store a cookie, commonly used for sessions
Cache-Control	Whether and how long the browser can cache this response
Location	Used with redirects to tell the browser where to go next
Access-Control-Allow-Origin	CORS header, which origins may read this response
Server	What software runs the server, useful for reconnaissance
X-Frame-Options	Whether this page can be embedded in an iframe
Strict-Transport-Security	Forces the browser to only use HTTPS going forward

Common request headers

Header	Meaning
Cookie	Session state sent back automatically by the browser
Authorization	Carries bearer tokens or basic auth credentials
Content-Type	How the request body is encoded, for example JSON or form data
Origin / Referer	Where the request came from, relevant to CORS and CSRF checks
X-CSRF-Token	Anti-CSRF token used by some applications

4. Cookies vs sessions vs tokens

HTTP is stateless, each request is independent with no built-in memory of previous ones. These three concepts exist to solve the same problem: how a server remembers who a client is across multiple requests.

Cookies: the transport mechanism

A cookie is a small piece of text the server asks the browser to store, using a Set-Cookie header. The browser then automatically attaches it back on every future request to that domain. A cookie is just a storage and auto-send mechanism, it does not by itself mean a user is logged in, that meaning is defined by the server.

```
Set-Cookie: session_id=a8f3e9; HttpOnly; Secure; SameSite=Strict
```

HttpOnly blocks JavaScript from reading the cookie, which helps against some XSS-based theft. Secure restricts the cookie to HTTPS only. SameSite restricts whether the cookie is sent on cross-site requests, which helps against CSRF.

Sessions: server-side state referenced by a cookie

On login, the server verifies the credentials and creates a session record in its own memory or database, mapping a session ID to a user. It sends that ID back as a cookie. On every future request, the browser sends the cookie, and the server looks up the ID to know who is asking. The cookie is only a claim ticket, the real identity data lives on the server. This makes revocation trivial, just delete the session record, but it does not scale across multiple servers unless the session store is shared.

Tokens: self-contained credentials

Instead of the server storing who a given ID belongs to, a token can contain that information itself, cryptographically signed so it cannot be tampered with. JSON Web Token, or JWT, is the most common format. Tokens are commonly sent using an Authorization header instead of a cookie, and stored in localStorage or memory on the client.

	Session + cookie	Token (JWT)
Where is the state?	Server-side	Inside the token itself
Revoke early?	Easy, delete server record	Hard, valid until expiry unless a blocklist exists
Scales across servers?	Needs a shared session store	Yes, any server with the secret can verify it
Typical storage	Cookie, sent automatically	localStorage or Authorization header, sent manually

5. JWT in detail

A JWT is three base64-encoded parts separated by dots: a header, a payload, and a signature.

```
eyJhbGciOiJIUzI1NiJ9 . eyJ1c2VyIjoIYWxpY2UifQ . 4f8s93jf...
```

Header

Identifies the signing algorithm, for example HS256, and the token type.

```
{ "alg": "HS256", "typ": "JWT" }
```

Payload

Holds the actual claims, such as user identity, role, and expiry time. This is only encoded, not encrypted, so anyone can decode and read it. Secrets should never be placed here.

```
{ "user": "alice", "role": "admin", "exp": 1720000000 }
```

Signature

Generated by the server using a secret key that never leaves the server: HMACSHA256(base64(header) + "." + base64(payload), secret_key). On every future request, the server recomputes this signature from the header and payload it received and compares it to the signature attached to the token. If they match, nothing was altered since signing and the payload is trusted. If they do not match, even a single changed character, the token is rejected.

Why this matters for testing

Because the secret never leaves the server, nobody else can compute a valid signature for a modified payload. If a server fails to actually verify the signature, or accepts an algorithm of none, a payload such as a role claim can be edited freely and the server will believe it. This is a real, documented class of JWT vulnerability worth testing for in Burp Repeater.

6. Authentication methods

The following are the most common ways an application verifies who is making a request, roughly ordered from oldest and simplest to more modern.

6.1 Basic authentication

The oldest and simplest scheme: the client sends a username and password with every single request, base64-encoded inside the Authorization header. There is no session and no token, credentials travel on every call.

How it works

1. The client combines username and password as "username:password".
2. That string is base64-encoded, not encrypted, and placed after "Basic" in the Authorization header.
3. The server decodes it, checks the credentials against its records on every request.
4. If valid, the server responds with the resource. If invalid, it responds with 401 and a WWW-Authenticate header.

Example flow

```
Request:  
GET /api/orders HTTP/1.1  
Authorization: Basic YWxpY2U6cGFzc3dvcmQxMjM=
```

Decoded: `alice:password123`

Strengths	Trade-offs
Extremely simple to implement	Credentials sent on every request
No server-side session storage needed	Only safe over HTTPS
	No easy way to expire access without changing the password

Where you'll see it: Simple internal APIs, some router/device admin panels, quick service-to-service scripts.

6.2 Session-based (cookie) authentication

The classic web app pattern. The server checks your password once, then remembers you using a piece of state it stores itself, referenced by a small ID the browser carries automatically.

How it works

1. User submits login form with username and password.
2. Server verifies the credentials, then creates a session record server-side, for example in memory or Redis, mapping a random session ID to the user.
3. Server responds with Set-Cookie, giving the browser that session ID.
4. On every later request, the browser automatically attaches the cookie.
5. Server looks up the session ID in its store to know who is making the request.
6. Logging out simply deletes the session record, instantly invalidating the cookie.

Example flow

Login response:

Set-Cookie: `sessionid=a8f3e9c1; HttpOnly; Secure; SameSite=Lax`

Every later request:

Cookie: `sessionid=a8f3e9c1`

Strengths	Trade-offs
Easy, instant revocation server-side	Needs a shared session store to scale across multiple servers
Browser handles sending it automatically	Vulnerable to CSRF if SameSite/CSRF tokens are missing
Well understood, mature tooling	

Where you'll see it: Django, Rails, and PHP apps by default; most traditional server-rendered websites.

6.3 Token-based authentication (opaque tokens and JWT)

Instead of a cookie pointing at server-side state, the client holds a token directly and sends it manually. Two flavors exist depending on whether the token needs a database lookup to mean anything.

How it works

1. User logs in, server issues a token instead of a cookie.
2. Opaque token: a random string stored in a server-side table, looked up on every request, functionally identical to a session ID.
3. JWT: a signed, self-contained token holding the user's claims, verified using a signature check rather than a database lookup.
4. Client stores the token, typically in memory or localStorage, and attaches it manually.
5. Server validates the token on each request, either by lookup (opaque) or signature check (JWT).

Example flow

Request header (either type):

Authorization: Bearer

Opaque: Bearer 9f8x2kslqTanZ... (random, meaningless alone)

JWT: Bearer eyJhbGciOiJIUzI1NiJ9... (signed, decodable)

Strengths	Trade-offs
JWTs scale well, no shared DB needed to verify	JWTs are hard to revoke early without a blocklist
Works naturally for mobile apps and APIs without cookies	Stored in localStorage, tokens can be exposed via XSS if the app is not careful

Where you'll see it: Mobile app backends, single-page apps, most modern public REST and GraphQL APIs.

6.4 OAuth 2.0

Not authentication by itself, it is a delegated authorization framework. It lets one application get limited access to a user's data on another service, without that application ever seeing the user's password.

How it works

1. User clicks "connect" or "sign in with X" on the client application.
2. Client redirects the user to the authorization server, for example Google, with a requested scope such as read-only calendar access.
3. User logs into Google directly and approves the specific permissions requested.
4. Google redirects back to the client with a short-lived authorization code.
5. Client exchanges that code, server-side, for an access token, and sometimes a refresh token.
6. Client uses the access token to call the resource server, for example the Calendar API, on the user's behalf.

Example flow

1. Redirect: `accounts.google.com/o/oauth2/v2/auth?scope=calendar.readonly...`
2. Callback: `https://client.app/callback?code=4/0AX4...`
3. Token exchange -> `access_token=ya29.a0Af..., refresh_token=1//0g...`

Strengths	Trade-offs
User's password is never shared with the client app	More moving parts to implement correctly
Access can be scoped narrowly and revoked from the provider's side	Misconfigured <code>redirect_uri</code> validation is a common real vulnerability

Where you'll see it: Sign in with Google/GitHub/Facebook buttons, third-party apps connecting to Gmail, Slack, or Salesforce.

6.5 OpenID Connect (OIDC)

Built directly on top of OAuth 2.0 to answer a question OAuth does not: who is this user. It adds a standardized identity layer to OAuth's access-delegation machinery.

How it works

1. Client starts an OAuth flow but requests the special "openid" scope alongside any data scopes.
2. User authenticates with the identity provider, for example Google or Microsoft, and consents.
3. Provider redirects back with both an access token and an ID token.
4. The ID token is itself a JWT, containing standardized identity claims such as sub (user ID), email, and name.
5. Client verifies the ID token's signature and reads the claims to know exactly who logged in.

Example flow

```
id_token (JWT) payload:
{ "iss": "https://accounts.google.com",
  "sub": "110169484474386276334",
  "email": "alice@example.com" }
```

Strengths	Trade-offs
Standardized identity claims across providers	Inherits all of OAuth's complexity
Reuses OAuth infrastructure, no separate protocol to run	Still requires validating issuer, audience, and expiry correctly

Where you'll see it: Almost every modern "continue with Google/Microsoft/Apple" identity button.

6.6 API keys

A long static string that identifies an application or account, rather than an individual person. Simplicity over security nuance, meant for machine clients calling an API directly.

How it works

1. Developer generates a key from the provider's dashboard, tied to their account.
2. Key is placed in every request, either as a header or a query parameter.
3. Server checks the key against its records to identify which account is calling and apply rate limits or billing.
4. There is normally no built-in expiry, the key stays valid until manually revoked.

Example flow

Header form:

Authorization: Api-Key sk_live_51H8x2KAbc...

Query param form (less safe, appears in logs):

GET /v1/data?api_key=sk_live_51H8x2KAbc...

Strengths	Trade-offs
Very simple for server-to-server calls	No per-user identity, just per-app
No login flow or session needed	Leaks easily into logs, git history, or client-side code if mishandled

Where you'll see it: Stripe, weather APIs, map APIs, most machine-to-machine or backend-to-backend integrations.

6.7 Multi-factor authentication (MFA)

A layer added on top of any method above, not a replacement for one. It requires a second, independent proof of identity before a session or token is ever issued.

How it works

1. User submits the first factor, typically a password (something you know).
2. Server accepts the password but withholds full access, prompting for a second factor.
3. User provides a second factor: an app-generated one-time code, an SMS code, or a biometric check (something you have or are).
4. Server verifies the second factor, then issues the normal session or token as usual.

Example flow

Step 1: POST /login { user, password } -> 200, mfa_required: true

Step 2: POST /login/verify { otp: "482913" } -> 200, session issued

Strengths	Trade-offs
Blocks most account takeovers from a leaked password alone	Adds friction to every login
Works alongside any auth method already in place	SMS-based codes are phishable and can be SIM-swapped

Where you'll see it: Banking apps, Google/Microsoft account login, most enterprise SSO portals.

6.8 Certificate-based authentication (mutual TLS)

Instead of a password, the client presents a cryptographic certificate during the TLS handshake itself, before any HTTP request is even sent.

How it works

1. Client and server both hold certificates issued by a trusted certificate authority.
2. During the TLS handshake, the server requests the client's certificate.
3. Client presents its certificate, proving possession of the matching private key.
4. Server validates the certificate chain and identifies the client before the HTTP layer even begins.

Example flow

```
TLS handshake:
Server -> CertificateRequest
Client -> Certificate + CertificateVerify (signed with private key)
Server validates chain -> connection established as authenticated client
```

Strengths	Trade-offs
Very strong, phishing-resistant identity proof	Certificate distribution and rotation is operationally heavy
No password to steal or guess	Not practical for typical consumer-facing apps

Where you'll see it: Bank-to-bank payment rails, government systems, service mesh traffic between backend microservices.

6.9 SAML

An older, XML-based standard for enterprise single sign-on, letting a company's identity provider vouch for a user to many separate business applications.

How it works

1. User tries to access a service provider, for example Salesforce, without being logged in.
2. Service provider redirects the user to the company's identity provider, for example Okta or Azure AD.
3. User authenticates against the identity provider, often already logged in via single sign-on.
4. Identity provider builds a signed XML SAML assertion describing the user and their attributes.
5. Browser posts that assertion back to the service provider, which validates the signature and logs the user in.

Example flow

```
alice@company.com
```

Strengths	Trade-offs
Mature, widely supported in enterprise software	Heavy XML format, more complex to parse and secure than JSON
Centralizes identity across many business apps at once	XML signature wrapping attacks are a known historical risk

Where you'll see it: Corporate single sign-on into Salesforce, Workday, and other enterprise SaaS tools.

How this maps to Burp Suite

What you see in traffic	Likely method
Authorization: Basic ...	Basic auth
Set-Cookie / Cookie: sessionid=...	Session-based
Authorization: Bearer eyJ... (starts with eyJ)	JWT
Authorization: Bearer	Opaque token
Redirect to accounts.google.com/o/oauth2/...	OAuth / OIDC flow
Authorization: Api-Key ... or X-API-Key:	API key

For more practical breakdowns like this one, follow contentintech.com and [@contentintech](https://www.instagram.com/contentintech) on Instagram.