OWASP A07:2025 Authentication Failures

JOÃO PAULO BARRACA

This lecture covers the OWASP category that deals with how applications establish and maintain the identity of their users. The topic has been renamed over the years: it was A2 "Broken Authentication" in the 2013 and 2017 editions, became A07 "Identification and Authentication Failures" in 2021, and is A07:2025 "Authentication Failures" in the current edition (same position, #7, with a slightly shorter name that reflects the 36 CWEs mapped to it). The 2021 renaming was deliberately broader: it covers failures of identification (weak or missing account binding, poor registration and recovery flows) and not only the login form itself.

The material has a deliberate structure. We start with HTTP, because almost every authentication weakness on the web is a consequence of how a stateless protocol is made to carry state. We then recap authentication concepts, review the mechanisms that carry authentication state (URL parameters, hidden values, cookies, tokens), and finish with exploitation and prevention.

As background reading, keep OWASP ASVS (chapters V2 Authentication and V3 Session Management), the OWASP Web Security Testing Guide, and NIST SP 800-63B at hand. These three documents are referred to repeatedly, and they represent the closest thing to an industry consensus on what "good" looks like. When the lecture quotes a CWE number, you can look it up on cwe.mitre.org to see the formal weakness definition and real-world examples.

OWASP A07 – Authentication Failures

- Application functions related to **authentication** and **session management** are often implemented incorrectly

- Allow attackers to **compromise passwords, keys, or session tokens**, or to **exploit other implementation flaws**
  - Attackers may assume other users' identities temporarily or permanently.

- Known as A2:2017 "Broken Authentication", then A07:2021 "Identification and Authentication Failures"; A07:2025 "Authentication Failures"

OWASP (the Open Worldwide Application Security Project) publishes the Top 10 as an awareness document, not as a standard or a complete checklist. Categories are ranked using a mix of incidence data contributed by security vendors and a community survey. The category is not about one specific bug; it is a family of weaknesses that share the outcome: an attacker acts as somebody else.

It helps to separate three related functions. Identification is the claim ("I am alice"), authentication is the proof of that claim, and session management is how the system remembers that the proof was accepted, so the user does not have to repeat it with every request. Failures in any of the three produce the same impact: account takeover.

The statement about "temporarily or permanently" matters in practice. A stolen session token gives access only until it expires or the victim logs out. A stolen password, or a successful abuse of the recovery process, gives persistent access and can be used to lock the legitimate owner out by changing the credentials and the recovery address. A compromised privileged account (administrator, service account, developer) usually leads to complete compromise of the application and its data.

Notice that nothing here requires a vulnerability in the code in the classical sense. Many of the issues are design and policy decisions, such as accepting weak passwords, offering unlimited attempts or using guessable recovery questions. That is why this category is hard to fix with a patch and is better addressed with secure defaults and well-tested libraries.

OWASP A07 – Authentication Failures

- Prevalence is widespread
  - due to the operation of most identity and access controls.
- Session management is the bedrock of authentication and access controls
  - present in all stateful applications
- Attackers can detect broken authentication using manual means
- Attackers can exploit them using automated tools
  - There are extensive password lists and dictionary attack tools

Authentication flaws are widespread because every application with accounts needs identity logic, and a surprising number of teams still write their own. Login, registration, password reset, session creation and logout are five separate features, each with its own failure modes, and an attacker needs only one of them to be weak.

Session management is called the bedrock because every access control decision that follows depends on it. If the application believes the session identifier, then all authorization checks inherit the weakness. You can have perfect role-based access control and still lose everything if session identifiers are predictable.

Detection is mostly manual because the flaws are logical: you must observe how the application behaves when you try an unknown user, a wrong password, a reused token or a skipped step. A scanner can find a missing Secure flag, but it will not tell you that the password reset token is derived from the time of the request. Exploitation, on the other hand, is cheap to automate: once a flaw is understood, a script can try millions of combinations, and public wordlists (for example, the RockYou leak with more than 14 million passwords, or the much larger collections of breached credentials) make guessing efficient.

An important asymmetry for defenders: the attacker needs a single working credential, while the defender needs every account to be strong. Because of this, controls that do not depend on user behaviour (rate limiting, multi-factor authentication, breached password screening) are more effective than awareness campaigns.

A Web Request



User clicks on a link for http://server.com

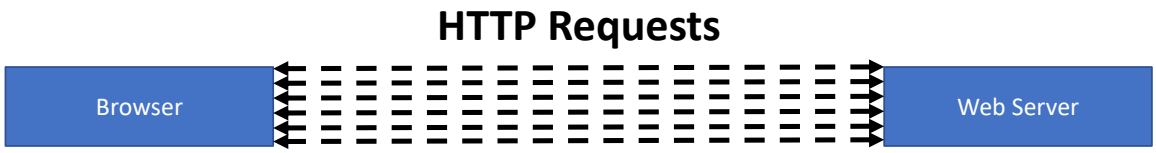
1. Browser asks DNS for the IP address of server.com
2. Browser connects to TCP port 80/443 of server.com
3. Browser sends a request with:
    1. Action: GET, POST, PUT, DELETE
    2. URL: http://server.com
    3. Headers: language, compression, user-agent…

Walk through what happens when a user clicks a link. The browser first resolves the host name using DNS (an unauthenticated protocol in most deployments, unless DNSSEC or DNS over HTTPS is in use, which is relevant to spoofing scenarios later). It then opens a TCP connection to port 80, or to port 443 if the URL uses https; in the latter case a TLS handshake follows and the server is authenticated through its certificate. Only then does the browser send the HTTP request.

An HTTP request consists of a request line (method, target and protocol version), headers, a blank line and an optional body. Methods express intent: GET retrieves, POST submits data, PUT replaces, DELETE removes, and there are others (HEAD, OPTIONS, PATCH). The Host header allows a single IP address to serve many sites. Other headers announce what the client accepts (Accept, Accept-Language, Accept-Encoding) and carry information about the client (User-Agent) and state (Cookie, Authorization).

Two points to retain. First, the URL in the slide is simplified: real URLs have a path, a query string and a fragment, and the fragment is never sent to the server. Second, everything in the request is produced by the client. The server knows nothing about the user unless the request itself contains something that identifies them; later sections show why that something is the critical element in authentication.

Multiple Web Request



- HTTP is stateless by design
- HTTP Requests are independent of each other
  - Each triggering an individual action
  - Usually **tokens** or **cookies** are included in requests/responses to keep state

This diagram stresses the lack of memory in HTTP: each request/response pair is an independent transaction. The server does not know whether two requests came from the same person, in what order they were issued, or whether the user has logged in. If the application needs to know, the client must prove its identity with every request.

This was a deliberate design choice that makes HTTP simple, scalable and cache-friendly: any server in a pool can answer any request, and intermediaries (proxies, CDNs) can serve repeated requests without involving the origin. The cost is that all state must be reconstructed from the request. There are only a few places where state can live: the URL (path and query parameters), the request body (form fields, JSON), headers (Cookie, Authorization, custom headers), or the connection itself (TLS client certificates, WebSockets).

For the server side there is also a design question: what does the token represent? Either it is an opaque reference (a random identifier) that the server resolves in a session store (memory, database, Redis), or it is a self-contained value (a signed token) that carries the state itself. The first approach allows immediate revocation and keeps data on the server; the second scales without shared storage, but is harder to revoke and exposes its content to the client. Many security problems come from mixing the two models or from using the wrong one for the requirements.

HTTP Communication

- HTTP is a standard Client-Server protocol

1. Client establishes a TCP connection with the server on port 80 (443 with TLS)
2. Client sends a HTTP request over that TCP connection
3. Server replies
   - Sends a response
   - HTTP 1.0: Closes the connection
   - HTTP 1.1: persistent by default; HTTP/2: one multiplexed connection

- Server only issues replies to requests. It may never contact clients directly

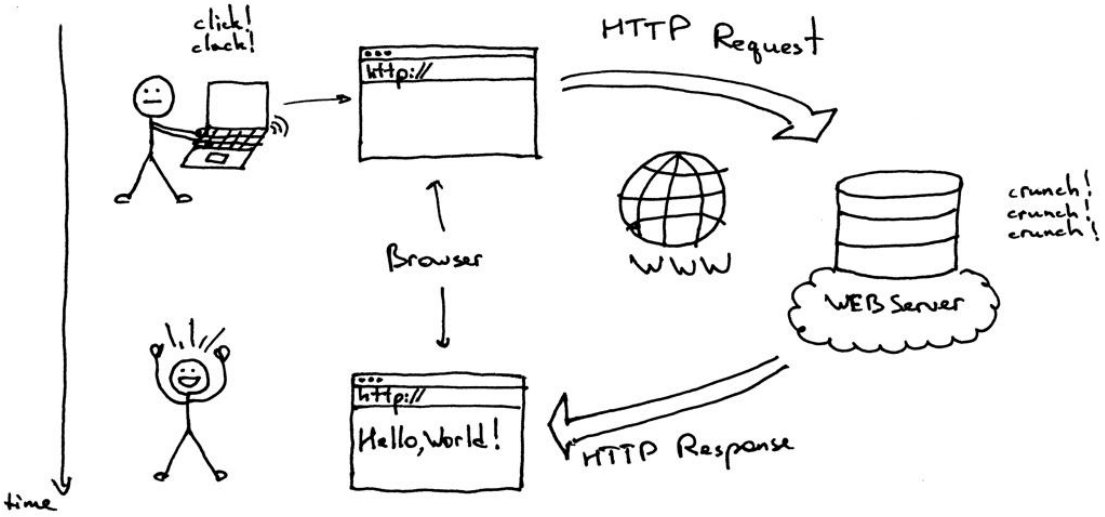
This slide describes the connection model. The client initiates a TCP connection (and TLS handshake when HTTPS is used), sends the request and the server sends back a status line, headers and the body. The status code tells the client what happened: 2xx success, 3xx redirection, 4xx client error (401 Unauthorized and 403 Forbidden are the ones that matter most for authentication) and 5xx server error. Note that 401 really means "unauthenticated" and 403 means "authenticated but not allowed", despite the unfortunate naming.

In HTTP/1.0 each request used a new connection. HTTP/1.1 made connections persistent by default (keep-alive), so that several requests reuse the same TCP and TLS session, saving round trips. HTTP/2 goes further and multiplexes concurrent requests as streams over a single connection. Notice that connection persistence is a performance feature and must not be confused with a session: the server cannot assume that the same TCP connection belongs to the same user, particularly behind proxies and load balancers that pool connections.

The last bullet describes the client-server model: the server is passive. It cannot push data to the browser spontaneously, which is why applications used polling or long polling for years to simulate notifications. This constraint is relevant for authentication because it limits what the server can do when it detects a problem: it cannot "log out" a client; it can only refuse future requests or invalidate the server-side state that the token refers to.

image: https://ruslanspivak.com

HTTP Communication



This diagram, taken from a well-known tutorial on building a web server, shows the lifecycle of a request at the socket level. It is useful to understand that web servers are ordinary programs that listen on a TCP socket, accept connections, read bytes, parse them according to the HTTP specification and write bytes back. There is no magic: a "web server" can be implemented in a few dozen lines in almost any language.

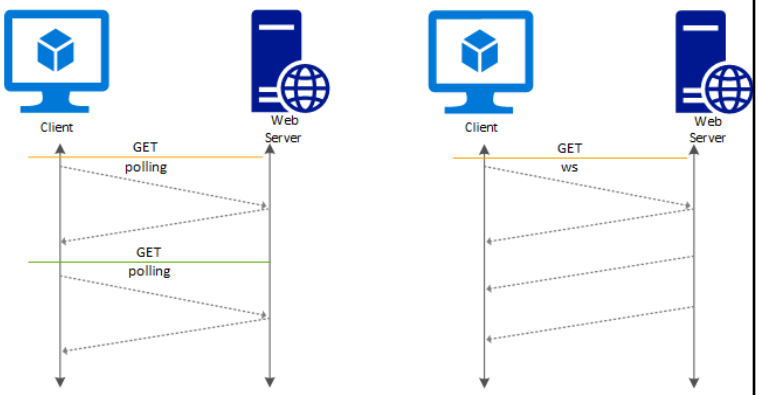
Two implications. First, a server must be robust to malformed, oversized or slow input: parsing code is a historical source of vulnerabilities, from buffer overflows to request smuggling, and different implementations disagree about corner cases (duplicate headers, odd whitespace, mixed line endings). Attackers use these differences to bypass security controls in front-end proxies. Second, because the server parses attacker-controlled bytes before any authentication takes place, every component involved in handling a request is part of the pre-authentication attack surface. That is an additional argument for keeping software up to date and for placing authentication as early as possible in the processing chain.

A good exercise is to run a minimal socket server in Python or Node and observe the raw bytes that a browser sends when you open the page, including the headers that the browser adds on its own (User-Agent, Accept, Accept-Encoding, Connection, Upgrade-Insecure-Requests, and, depending on the context, Referer, Origin and Sec-Fetch-* headers).

image: Microsoft.com

HTTP Communication

- Actually… servers can contact clients directly with WebSockets
  - Great for low latency asynchronous communications (e.g. VoIP, telemetry)
  - Challenging for security!
- Client upgrades connection to a WebSocket
- Any participant can send message
  - No polling is required
    Rarely logged by default
  - Client and server must know the message format



WebSockets (RFC 6455) start as an ordinary HTTP request with an Upgrade: websocket header. If the server agrees, it answers with 101 Switching Protocols and the connection stops being HTTP: from then on, both sides can send framed messages at any time over the same TCP connection. This is efficient for chat, collaborative editing, games, financial tickers or telemetry, because there is no repeated header overhead and no polling.

Security matters to consider. Authentication normally happens in the handshake (cookies are sent automatically by the browser, and that is exactly the problem: browsers do not enforce the same-origin policy for WebSockets). If the server does not check the Origin header, any web page the victim visits can open a WebSocket to the target site, using the victim's cookies, and read and write messages. This is called Cross-Site WebSocket Hijacking. Applications must validate Origin and should also use an explicit token (for example, a short-lived ticket obtained over HTTP and passed in the first message) instead of relying on ambient cookies alone.

Second, after the upgrade the message format is application-defined, so ordinary HTTP controls (WAFs, proxies, logs, CSRF protections, status codes) do not see or understand the traffic. Authorization must be checked on each message, not only when the connection is established, and long-lived connections must handle expiration and revocation: a session that has been terminated should also terminate the sockets associated with it. Finally, remember that WebSockets are one of the few ways in which a server can talk to a client without being asked.

HTTP Request

```
$ curl https://elearning.ua.pt  -D - -v


GET / HTTP/1.1
Host: elearning.ua.pt
User-Agent: curl/7.68.0
Accept: */*
```

HTTP Request

This is a real request produced by curl against the university's learning platform, with -v to show the headers and -D - to dump the response headers. Observe how short a request can be: a request line, a Host header, a User-Agent and an Accept header. Nothing else is required for HTTP/1.1 other than the Host header.

Everything is plain text and under the control of whoever writes it. The User-Agent identifies curl, but it is just a string and any other value could be sent; servers that use it for decisions (for example to block unwanted clients or to detect "mobile" devices) can be misled trivially. The same applies to Accept, Accept-Language and any other informational header.

In a request made by a browser for an authenticated user, you would additionally see a Cookie header (or an Authorization header) carrying the proof of identity. When you inspect such traffic with a proxy, you are looking at exactly what an attacker would need to steal in order to impersonate the user. That is why leaking request headers (for example in logs, in screenshots, in bug reports, in shared HAR files, in curl commands copied from the developer tools) is a real risk: treat them as passwords. A common source of incidents is a HAR export or a copied "curl" command pasted into a public ticket with live session cookies included.

Try it yourself with a site you are authorised to test, and compare the output of curl with the traffic generated by a browser.

HTTP Response

```
$ curl https://elearning.ua.pt  -D - -v

HTTP/1.1 200 OK
Date: Thu, 12 Nov 2020 17:01:16 GMT
Server: Apache
Set-Cookie: MoodleSession=qvnej3ar6u28ndar312jhg1veh; path=/
Expires: Mon, 20 Aug 1969 09:23:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Cache-Control: post-check=0, pre-check=0, no-transform
Last-Modified: Thu, 12 Nov 2020 17:01:16 GMT
Accept-Ranges: none
```

HTTP Response
With server timestamp
Cookie, Cache control

In the response, pay attention to the Set-Cookie header: this is the moment where the server creates a session and gives the client an identifier (MoodleSession here). Note that the cookie was set to an anonymous visitor, before any login. That is normal, since the application uses the session for preferences and for the login process itself; the important question is what happens to this identifier when the user authenticates (see session fixation later).

The remaining headers show how the server controls caching. Expires in the past, Cache-Control: no-store, no-cache, must-revalidate and Pragma: no-cache tell browsers and intermediaries not to keep copies. This matters for authenticated content: without those headers, a page with personal data can be stored in a shared proxy or remain accessible with the browser's back button after logout. The "post-check/pre-check" values are an old Internet Explorer extension and are not meaningful today.

The Server header reveals the software (Apache), which is a minor information leak that helps attackers to select exploits; many hardening guides suggest removing version details. Date and Last-Modified give the server's clock.

Also notice what is missing: this Set-Cookie has no Secure, HttpOnly or SameSite attribute (the capture is from 2020; modern browsers apply SameSite=Lax by default when the attribute is absent, but the other two are still not implied). Missing security headers such as Strict-Transport-Security or Content-Security-Policy are also common findings in assessments.

Anything can be a client

```
$ echo -ne 'GET / HTTP/1.1\r\nHost: elearning.ua.pt\r\nUser-Agent: Android
10\r\n\r\n' | ncat --ssl elearning.ua.pt 443

HTTP/1.1 200 OK
Date: Thu, 12 Nov 2020 17:20:12 GMT
Server: Apache
Set-Cookie: MoodleSession=ooma3far88iqh9nvssn598nsuu; path=/
Expires: Mon, 20 Aug 1969 09:23:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Cache-Control: post-check=0, pre-check=0, no-transform
Last-Modified: Thu, 12 Nov 2020 17:20:12 GMT
Accept-Ranges: none
```


This example removes the browser from the picture. With a shell one-liner and ncat (the Nmap project's netcat) we open a TLS connection to port 443 and write an HTTP request by hand, including a User-Agent claiming to be an Android phone. The server answers exactly as it would to a real browser, and even creates a session cookie.

The point is that the server has no means to verify who or what is on the other end. Headers like User-Agent, Referer, Origin, X-Forwarded-For or Accept-Language are claims made by the client. Security decisions based on them are decisions based on the attacker's own words. Typical bad examples: blocking access based on User-Agent, enforcing that requests "come from our app", checking the Referer for authorization, or trusting X-Forwarded-For for IP allow lists when no trusted proxy overwrites it.

The \r\n sequences are HTTP's line terminators, and the empty line (\r\n\r\n) marks the end of the headers. Forgetting them is the most common reason why a hand-written request hangs, because the server keeps waiting for the rest of the request. The same approach is used for testing how servers deal with malformed requests, a technique that is also used for request smuggling research.

Consider, as a defender, what you can really rely on. The IP address of the TCP peer is hard to spoof for full connections (but can be a proxy or a NAT shared by many users), TLS client certificates are cryptographically verifiable, and cookies/tokens are only as trustworthy as their secrecy and entropy.

Anything can be a client

- Many programs can communicate with HTTP servers
  - A socket is all that is required
- Even Bash can do it

Open TCP to 193.136.173.58:80 into FD5
Write to FD 5
Read from FD 5

```
$ exec 5<>/dev/tcp/193.136.173.58/80
$ echo -e "GET / HTTP/1.1\r\nHost: www.ua.pt\r\n\r\n" >&5
$ cat <&5

HTTP/1.1 301 Moved Permanently
Server: nginx/1.18.0
Date: Thu, 12 Nov 2020 17:26:58 GMT
Content-Type: text/html
Content-Length: 169
Connection: keep-alive
Location: https://www.ua.pt/
```


This example takes the idea of a hand-made request to the extreme: Bash can open TCP connections by itself using the special /dev/tcp pseudo-path. The command exec 5<>/dev/tcp/host/80 opens a bidirectional connection and assigns it to file descriptor 5; echo writes the request into it, and cat reads the response. No HTTP library, no curl, nothing but the shell.

Why does this matter in a lecture about authentication? First, because it reinforces that any program with a socket is a potential HTTP client. Malware, scripts, bots and attack tools do not need browsers. Rate limiting based on "browser-like behaviour" or JavaScript challenges can slow down attackers, but they are not authentication controls. Second, because this feature is a classical way to build reverse shells and to exfiltrate data from restricted systems that lack curl or wget; defenders should know that blocking binaries is not enough.

The response in the example is a 301 redirect to HTTPS, a very common behaviour. Note that in this exchange nothing was encrypted: any credential or cookie sent over port 80 can be read and modified by anyone on the path. That is the reason for HSTS (Strict-Transport-Security), which tells browsers to use HTTPS only for a site, and for redirecting plain HTTP only after the first, unprotected, request, which is a window that attackers (for example with sslstrip-style attacks) can exploit.

Note that the Host header is required for HTTP/1.1, because many sites share one IP address. If you change it, the server may return a different site, a property used in virtual host discovery.

Anything can be a client

- There is no client-side security model
- All parts of a request can be crafted
  - HTTP Headers, Methods, URLs
  - POST content can be manipulated freely
- Control must reside in the server-side context
  - Remember that developers are pushing content to the client? ☺
- The server cannot know if any client-side validation was performed
  - As long as the HTTP protocol is “generally” observed

This slide summarises the key security principle of the web: there is no client-side security model. Whatever runs in the browser (JavaScript validations, disabled buttons, hidden form fields, maxlength attributes, front-end role checks, minified code that "hides" API endpoints) is advisory and for user convenience. An attacker can bypass them by intercepting the request, using a different client or editing the page in the developer tools.

In authentication this appears in recurring mistakes: password rules enforced only in JavaScript; a hidden field such as role=user or isAdmin=false that the server believes; a "remember me" cookie that contains the user name and is accepted as-is; price, user id or account number sent from the client and not rechecked; "step 2" of a multi-step process (for example MFA) that can be accessed directly without completing step 1; or an API that enforces login in the user interface but not at the endpoint. All of them have the same root cause: the server trusts the client to follow the intended flow.

The defensive principle is to treat all input as hostile and to enforce every rule on the server: authentication, authorization on every request, input validation and business logic. Client-side checks are still useful to give quick feedback to users, but never as the control. This slide is also the bridge to the next topics: when the client keeps the state (cookies, tokens), it can modify that state, so the server must be able to detect modification, usually with a secret (random, unpredictable identifiers or cryptographic signatures).

Authentication

The next block recapitulates the vocabulary of authentication, most of which was introduced in earlier courses. It is repeated because inconsistent use of the terms is a source of design mistakes.

Authentication answers "who are you?", authorization answers "what are you allowed to do?", and accounting records what was done. A system can have strong authentication and no authorization, or the opposite. In this lecture the focus is the first, but the weaknesses in authentication directly feed into authorization because every access control decision uses the identity that authentication produced.

It is also useful to separate authenticating a user (a person, with a device, a browser, a password) from authenticating a service (a machine, with a key, a certificate or a token). Many incidents have been caused by service credentials: API keys committed to code repositories, cloud access tokens in environment variables, or default credentials in management interfaces. Rotation and secret management for non-human identities are as important as protection of human passwords.

Finally, remember that authentication is probabilistic: it provides a level of confidence about identity. NIST SP 800-63 formalises this as assurance levels, and risk-based systems adapt the level of proof to the sensitivity of the operation (viewing a balance versus transferring money).

Authentication - Recap

- Authentication aims to validate the identity of an entity
  - Entity may be user, system, software

- The basic process relies in the verification of some property of the authenticated entity by the authenticator
  - Something the entity has
  - Something the entity knows
  - Something the entity is



Authentication is the process of verifying a claimed identity. The "entity" may be a person, a device, a process or another service. The classical taxonomy groups the evidence into three factors: something you know (password, PIN, answer to a question), something you have (smart card, hardware token, phone with an authenticator application) and something you are (biometrics such as fingerprint, face or iris).

Each factor has typical weaknesses. Knowledge can be guessed, phished, observed, reused or leaked in a database breach. Possession can be stolen, cloned or intercepted (SIM swapping defeats SMS codes). Inherence cannot be changed if compromised and is often probabilistic (false accept/false reject rates), is impossible to keep secret (fingerprints on glass, faces on photos) and, in many systems, is validated locally by the device and not by the server, so what the server receives is a statement that the biometric check succeeded.

Notice also that the three categories are only counted as independent factors if they are really different. Two passwords, or a password and a security question, are two instances of the same factor and do not provide multi-factor security. The same applies to a password plus a code sent by e-mail when the attacker already controls that mailbox.

The verifier (the authenticator, in the slide's terms) must also be authenticated by the user in the opposite direction: that is the role of TLS server certificates and why phishing is effective when users cannot tell the real site from a fake one.

Authentication - Recap

- Somewhere the entity is (location)
- Someone nearby (neighborhood)
- Something the entity has done (past behavior)

- A combination of several
  - 2FA – Two-Factor Authentication (e.g. Password + one-time code)
  - MFA – Multi-Factor: 2 or more different factor types (e.g. Password + Smartphone + Fingerprint)

Beyond the three classical factors, systems may use contextual attributes. Location can be physical (GPS, IP geolocation, cell towers), or logical (the request comes from the corporate network or a VPN). "Someone close by" refers to proximity or social-graph evidence: a trusted device nearby over Bluetooth, or a user vouching for another, as in some account recovery schemes where trusted contacts help you regain access. Past behaviour includes the usual login times, devices, typical transactions and interaction patterns.

All of these are useful for risk assessment, but they are not strong proof: location can be spoofed, networks are shared, and behaviour changes. They are usually combined with other factors rather than used alone.

On combinations: Two-factor authentication (2FA) is the specific case of multi-factor authentication (MFA) with exactly two different factor types, for example a password plus a time-based one-time code (TOTP) or a hardware key. The factors must be independent: a password plus a security question is not 2FA, because both are knowledge, and a cookie is not a factor either: it is the result of an earlier authentication and is stored by the browser, so it only proves that authentication happened at some point. Treating a cookie as a factor is a common misconception. In practice, the cookie is what a session hijacker steals, and it is why MFA at login does not protect a session after it has been established.

Not all MFA is equal: SMS and e-mail codes can be phished or intercepted, TOTP can be relayed by a phishing proxy, and FIDO2/WebAuthn keys are bound to the real site and resist phishing.

Base HTTP methods

- Makes use of the Authorization header
  - Credentials are resent in every request and cached by the browser
  - Base64 is an encoding, not encryption: password exposed without TLS
  - Native browser prompt: no customization, no real logout

- Basic authentication through direct presentation of credentials
  - `Authorization: Basic base64(login:password)`

HTTP defines authentication in the protocol itself (RFC 7235 describes the framework and RFC 7617 describes Basic). When a resource is protected, the server answers 401 Unauthorized with a WWW-Authenticate header naming the scheme and the realm; the browser then asks the user for credentials, and resends the request with an Authorization header.

In Basic authentication the credentials are the string user:password encoded with Base64. It is an encoding, not encryption: anyone who sees the header can decode it with one command (echo ... | base64 -d). Without TLS, passwords travel in clear. Even with TLS, the browser sends them with every request to the protection space, which increases the exposure: they can end up in proxies' logs, in debug output or in an application's memory. Another limitation is that there is no logout: the browser caches the credentials for the session and there is no standard way to forget them except to close the browser; and the login form cannot be styled or extended with MFA, rate limiting messages or password recovery.

Its remaining advantages are simplicity and wide support, which makes it common for APIs, internal tools, CI systems and devices. If used, it must be over HTTPS only, with strong generated secrets (API keys rather than human passwords) and with rate limiting. Brute-force tools such as Hydra support HTTP Basic natively, which is often the quickest check in a penetration test on a management interface.

Note the user:password format: a colon in the user name breaks the syntax, which is one reason why it is forbidden in the specification.

Base HTTP methods

- Digest authentication
  - Server challenges (401) via WWW-Authenticate; client replies with a hash (MD5 by default, SHA-256 in RFC 7616)

```
Authorization: Digest username="Mufasa",
                      realm="testrealm@host.com",
                      nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                      uri="/dir/index.html",
                      qop=auth,
                      nc=00000001,
                      cnonce="0a4f113b",
                      response="6629fae49393a05397450978507c4ef1",
                      opaque="5ccc069c403ebaf9f0171e9517f40e41"
```


Digest authentication (RFC 7616, earlier RFC 2617) was created to avoid sending the password over the wire. The server issues a challenge containing a realm and a random nonce. The client computes a hash over the user name, realm, password, nonce, method and URI, and returns that hash as the response, together with its own client nonce (cnonce) and a counter (nc), which prevent precomputed and replay attacks. The example on the slide is the one from the RFC, and the highlighted response is the value computed by the client.

Why is it rarely used today? Historically it relied on MD5, which is considered broken for collision resistance and too fast for password protection; SHA-256 was added later but support is uneven. To verify the response, the server needs the password, or at least the hash of user:realm:password (HA1), stored in a form that is equivalent to the password for the purpose of authenticating to this server. This conflicts with modern password storage practice that uses salted, slow hashes (Argon2, bcrypt, scrypt, PBKDF2), which would make Digest impossible. Digest is also vulnerable to man-in-the-middle attacks, since an attacker may downgrade to Basic or may change the challenge, and it does not protect the content of the session, so TLS is still required.

The practical conclusion is that Digest improved a world of plain-text channels, but in a world with ubiquitous TLS, its benefit is small, and its disadvantages (storage, limited UI, no MFA) are significant. Applications normally use a form-based login with a session or a token instead.

Authentication Flow state

This section moves from the question "how is identity proven?" to "how is the proven identity remembered?". Once the user has authenticated, the application must connect the following requests to that result. Authentication happens once, but the requests continue for minutes or days. The mechanism used to carry this link is the real target in most attacks: why guess a password if you can copy the cookie that proves it was already entered?

The section addresses the main alternatives (URL arguments, hidden fields, headers, cookies and tokens), explaining their properties and limits. A useful way to compare them is to consider four questions. Who creates the value (server or client)? Does the browser send it automatically (cookies) or must code add it explicitly (headers, tokens in JavaScript)? Where is the state kept (server-side lookup or inside the token)? And how is the value protected against theft, guessing and tampering?

Also, think about the meaning of "logged in". For the server it is only the validity of a token at a certain point in time. There are no sessions in the HTTP sense, only conventions, so every property that we expect (expires after inactivity, ends at logout, is bound to the device, cannot be reused after a password change) must be implemented deliberately.

Sessions

- Most applications over HTTP need state for good purposes
  - User preferences
  - Navigation history
  - Authentication state
- Some use it for less noble purposes, usually compromising privacy
  - Track users across multiple sites for advertising purposes
  - Profile user behavior

State is not a bad thing in itself. A shopping basket, language preference, a multi-step form or an authenticated identity all need state between requests. HTTP being stateless means that it must be added by the application on top of the protocol, and the slide lists the typical benefits.

The "less noble purposes" are worth discussing with students because they affect the evolution of the mechanisms. Third-party cookies were used for cross-site tracking for decades, which led to regulations (ePrivacy and GDPR consent banners in Europe) and technical measures: Safari and Firefox block third-party cookies by default, and Chrome has introduced controls and Privacy Sandbox alternatives. Browsers now partition storage and apply SameSite=Lax by default. Side effects exist for authentication: embedded single sign-on and federated login flows that depended on third-party cookies had to be redesigned.

From a security view, any state kept for a user is also data that has to be protected, since it can identify, profile or be used to impersonate. Session data should be kept to the minimum needed, expire and be deleted when it is no longer required. Storing sensitive information in the client (even if signed or encrypted) leaks it to the user and to any attacker with access to the device.

The remark on WebSockets recalls that a persistent connection is the exception to the stateless model: the connection itself carries the context. Nonetheless, the authentication of that connection depends on one of the mechanisms we study here.

Flow State keeping mechanisms

- Referer header
- SESSION_ID, or SID or other custom headers
- Cookie
- JSON Web Token

These are the four families of mechanisms commonly used to link requests. Each has typical use cases and different security properties.

The Referer header is not a real state mechanism, and it must never be used as one: it is optional, informational and controlled by the client. It is included because it is frequently (mis)used in access control and in CSRF defences, and because it leaks information.

A session identifier sent as an explicit parameter or custom header (for example X-Auth-Token) is controlled by the application's code. It must be added to every request by JavaScript or by the HTML itself, and it does not suffer from the "ambient authority" problem of cookies (the browser never attaches it on its own), but it is more exposed to theft by script injection, and, if placed in the URL, it leaks through logs and referrers.

Cookies are standardised, are stored and sent automatically by the browser and have attributes that control their scope and protection. The automatic sending is convenient and also the cause of cross-site request forgery. JSON Web Tokens are a format, not a transport: they may be carried in a cookie or in an Authorization: Bearer header. Their distinguishing feature is that they are self-contained and signed, so the server may not need a session store.

There are other mechanisms not shown: TLS client certificates, hidden form fields and cryptographic request signing as used by some cloud APIs. The decision is a trade-off between security, usability and architecture.

Use of the URL

```
GET /internal/private.html?pass=secret&sid=234234 HTTP/1.1
Host: www.company.com
```

- Input encoded as part of the URL as Request Arguments

- GET is expected to be safe: no side effects on the server
  - Arguments control language, authentication, authorization

Putting authentication state in the URL means that it travels in the request line. The example is deliberately terrible: it contains a password (pass=secret) and a session identifier (sid=...) in the query string. It was common in early web applications and it can still be found in old code, in mobile app APIs and in "magic link" implementations.

The URL is treated by the web ecosystem as non-sensitive. It is stored in the browser history and bookmarks, may be synchronised between devices, appears in the address bar and in screenshots, is written in the access logs of web servers, proxies, load balancers, CDNs and WAFs, and is sent to other sites in the Referer header when the user follows a link or loads an external resource. A secret in a URL has, in effect, many copies in many systems with different retention and access policies.

The slide also reminds us about the semantic of GET: it should be safe, that is, a request that does not change server state, and idempotent. This is not a security mechanism, but a contract that browsers, crawlers, prefetchers and caches rely on. If a link such as /logout or /delete?id=7 performs an action, a prefetcher or an image tag on another site can trigger it. Contrary to a common belief, GET requests are not expected to have side effects: arguments in a GET should only select or filter what is retrieved (language, page, sort order).

Where tokens must appear in a URL (email verification, password reset), use single-use, short-lived values, and avoid loading third-party resources on that page.

Use of the URL

Should be avoided at all cost to transport state

- Arguments are visible in the browser
  - A use problem if your browser is visible: public presentation, remote lecture, over the shoulder eavesdropping
- Arguments may be logged by the webserver
  - Enable compromise if logs are accessed by an attacker
- SEO is broken: different users will have a different URL for the same resource
- Cache may be impacted: unique URLs limit the use of caches
- Leaks to third parties through the Referer header and browser history

This slide enumerates why URLs are unsuitable for authentication state. A few extensions help understanding.

Visibility: the browser displays the URL, so anyone who sees the screen (a shared projector, a recorded lecture, a screen-sharing session, a photograph, a screenshot posted to a forum) sees the secret. Users also copy and paste URLs in messages and emails to colleagues, unintentionally sharing their session: whoever opens the link is logged in as them. This has caused public leaks of private documents when links containing session or access tokens were shared.

Logging: web server logs are generally less protected than databases and are kept for long periods, shipped to log management platforms and read by many people. An attacker who obtains read access to logs (a path traversal, a misconfigured storage bucket, a compromised analytics tool) gets valid credentials. The slide's wording "compromise" is exactly that: a secondary vulnerability turns into account takeover.

SEO and caching: if every user receives a different URL for the same resource, search engines index duplicates (or session-specific pages) and caches cannot share content. Those are not security issues themselves, but they are why frameworks that used URL rewriting for sessions (for example the ;jsessionid=... suffix in Java servers) are now configured to disable it.

Finally, URLs leak through the Referer header: if a page with the token in the URL links to or embeds resources from another origin (an image, an analytics script), that origin receives the complete URL. Modern browsers reduce this by default with a strict-origin-when-cross-origin Referrer-Policy, but you should not depend on it.

Use of a POST request

```
POST /doLogin HTTP/1.1
Host: company.com
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows 10)
Referer: http://company.com/login
Content-Length: 34

username=john&password=supersecret
```

- URL visible in the browser: http://company.com/doLogin
- The body is hidden from the URL, but only TLS protects it

Moving the data into a POST body fixes the most obvious exposures of URL-based state: the credentials do not appear in the address bar, browser history, bookmarks, Referer headers or the default access logs of the web server (which record only the request line). This is why login forms use POST.

But it is important not to overstate the protection. The body is not encrypted, and, without TLS, it is readable by anyone on the network path. The example shows exactly that: username=john&password=supersecret in clear text. Also, the body may be logged by applications, debugging tools, API gateways and WAFs configured to record full requests; an infamous category of incidents is the accidental logging of passwords in plain text by the application itself or by monitoring tools (several large companies have disclosed such cases). Developers should mask sensitive fields in logs.

The headers in the example include Pragma: no-cache and Cache-Control: no-cache. They are meant to prevent the response from being reused without validation. The Referer is the login page and is optional. Content-Length must match the body: the value 34 corresponds to the 34 characters of the form-encoded data, which can be a good exercise to verify.

Remember also what the URL shows after login: the action URL (/doLogin) is visible but harmless. A frequent error is a redirect after a successful login to a URL containing a parameter such as ?next=... which can be abused for open redirects and phishing, so the target must be validated against a whitelist of internal paths.

GET vs POST

- **GET** is used to REQUEST information
  - Can be resent by browsers
  - May be logged, cached, bookmarked, kept in the browser history
  - Should not change server-side state (no side-effects)
    - Frequently it will change state, or create logs

- **POST is used to SUBMIT data / change state**
  - Responses are not cached by default; not bookmarked or kept in history
  - Body usually not logged by default
  - Body is not in the URL, but is visible in browser tools and proxies
  - Is expected to change server-side state (has side effects)

This comparison shows the intended semantics of the two most common methods (defined in RFC 9110). GET is safe and idempotent: it asks for a representation of a resource and should not change the state of the server. POST submits data to be processed and is neither safe nor idempotent, so browsers warn before resubmitting a POST on page reload.

Security-relevant consequences. Browsers and intermediaries may cache, prefetch and repeat GET requests, so a GET that performs an action is dangerous (logout links, "delete" links, unsubscribe links, one-click confirmations). Many CSRF issues exist because state-changing operations accept GET. Conversely, POST responses are not cached by default and the browser does not keep POST bodies in the history.

It is common to hear that POST is "more secure" than GET. That is only true with regard to the URL-related leaks (history, logs, Referer, bookmarks). The POST body is as visible to the user, to a proxy or to an attacker on the network as the query string, and it is as easy to modify and replay. It is not hidden from the user, who can see it in the developer tools. Also, other methods exist: PUT, PATCH and DELETE in REST APIs follow similar considerations, and some applications accept method overriding headers (X-HTTP-Method-Override) which can bypass controls that only inspect certain methods.

The final remark in the slide, that GET requests frequently change state or create logs, is a reminder that "should not" is not "does not". Authorization and CSRF protections need to be applied based on what an endpoint does, not on the method it was documented to accept.

Referer Header

- The **Referer** request header contains the address (URL) of the page that linked to the requested resource.

- **Referer** header allows servers to identify where people are visiting them from
  - may use that data for analytics, logging, or optimized caching
  - Sometimes (wrongly) used for access control

- **Fully client controlled**: can be forged, removed or trimmed (Referrer-Policy)

The Referer header (the misspelling is in the original specification and has been preserved ever since; the related policy header is correctly spelled Referrer-Policy) tells the server which page the request came from. When you click a link on page A that leads to site B, B receives A's URL. The same happens for images, scripts, style sheets and other subresources loaded by A.

Legitimate uses: analytics (which sites send traffic), logging, caching hints, and some defence-in-depth checks such as comparing Referer or Origin against the expected origin for state-changing requests (a CSRF mitigation, preferably using Origin or Sec-Fetch-Site headers and tokens). Problematic uses: access control, as the slide points out. For example, "only serve this download if the Referer is our own site" is a hotlink protection, not a security control.

Leakage is the other side of the header. Full URLs are sent to third parties, which may include sensitive information such as session identifiers, password reset tokens, internal host names, search terms, or the identifier of a document. This is a reason to avoid secrets in URLs. Browsers now default to a more restrictive behaviour (strict-origin-when-cross-origin: only the origin is sent to other origins, and nothing is sent when going from HTTPS to HTTP), and sites can set Referrer-Policy: no-referrer or same-origin, or use rel="noreferrer" on links.

Finally, the value is fully under client control: it may be missing (privacy extensions, policy, direct navigation), truncated, or forged by any non-browser client. A missing Referer is not evidence of an attack nor of its absence.

Referer Header

- **First hit: No Referer**

```
GET https://elearning.ua.pt/ HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:82.0) Gecko/20100101 Firefox/82.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Cookie: _ga_RWZB1HRVYE=GS1.1.1605202432.1.1.1605202861.0; _ga=GA1.1.857190253.1605202434;
_gid=GA1.2.1334581424.1605202436; _hjTLDTest=1; _hjid=25f0d0c2-0616-4e02-86b2-ba6a336f5a99;
_hjFirstSeen=1; _hjAbsoluteSessionInProgress=0
Upgrade-Insecure-Requests: 1
Host: elearning.ua.pt
```


This capture, obtained with an intercepting proxy, shows the first request to the site when the user types the address or follows a bookmark. There is no Referer header because there is no referring page. It is a good illustration of the idea that the header is only added when a navigation or subresource load is triggered from another page, and can be absent in many legitimate cases (typing the URL, bookmarks, links from native applications, HTTPS to HTTP transitions, links with rel=noreferrer, strict policies).

Observe the other headers. The Cookie header already carries several identifiers, but they are not session cookies: _ga, _gid, _hjid and others belong to Google Analytics and Hotjar, third-party analytics services used to track visitors and to record behaviour. They are an example of the "less noble purposes" of state, and they appear in every request to the domain. Besides the privacy aspects, they enlarge the headers and, if a site were to be affected by a cross-site scripting vulnerability, the scripts of these third parties execute with the same privileges as the application's own code; supply-chain compromises of such scripts have been used to steal payment data and credentials.

The Upgrade-Insecure-Requests header means that the browser prefers HTTPS and will accept a redirect to it. Accept, Accept-Language and User-Agent are informational. Also note that the request was captured in clear text because the proxy sits between the browser and the server; this is possible because the user installed the proxy's certificate authority in the browser, which is also why compromising a user's trust store is so powerful.

Referer Header

- **Subsequent request**

```
GET https://elearning.ua.pt/theme/adaptable/style/print.css HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:82.0) Gecko/20100101 Firefox/82.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Referer: https://elearning.ua.pt/
Cookie: _ga_RWZB1HRVYE=GS1.1.1605202432.1.1.1605202861.0;
_ga=GA1.1.857190253.1605202434; _gid=GA1.2.1334581424.1605202436; _hjTLDTest=1;
_hjid=25f0d0c2-0616-4e02-86b2-ba6a336f5a99; _hjFirstSeen=1; _hjAbsoluteSessionInProgress=0;
MoodleSession=bcsqms7e7h324mmv1s46favccl
Host: elearning.ua.pt
```


Here the browser is loading a style sheet linked from the home page, so the request now includes Referer: https://elearning.ua.pt/. The header is automatically added by the browser; the page author did nothing. Compared with a request for the home page, the cookie list now also includes a MoodleSession cookie. That is the session identifier created by the server in the first response, now returned by the browser on every request to the site, including images, scripts and style sheets.

This is the cookie mechanism at work, and it shows two properties that matter for security. First, the cookie is sent automatically with every request for a resource of that site, even those triggered by another site (subject to SameSite rules). This automatic behaviour is what makes CSRF possible. Second, it gets copied in all places where requests are recorded: proxy logs, debugging tools, HAR files and screenshots, which is why you should be careful when sharing such captures. The session shown here is from a lab capture in 2020 and is no longer valid, but in a real incident response a cookie in a ticket attachment can be as valuable as a password.

Observe also the Accept header for the CSS (text/css,*/*;q=0.1): browsers announce what type of resource they expect. The Referer tells the server which page requested the asset; it is used for hotlink protection of images and for analytics, and it is trivially forged.

If you try this yourself, compare what the browser developer tools show with what a proxy shows; the proxy sees the exact bytes, while the browser tools reformat the headers.

Referer Header

```
GET /internal/private.html HTTP/1.1
Host: www.company.com
Referer: http://www.company.com/loggedin/
```

- Expected meaning: User accessing `/internal/private.html`, and came from `/loggedin` therefore it was authenticated
- In reality
  - Referer is set by the client: it may be omitted, removed by policy, or forged
  - Meant for informational use (analytics, logs), misused for access control (CWE-293)
  - Trusts client-controlled data in a security decision

This slide describes a frequent vulnerability pattern, classified as CWE-293 (Using Referer Field for Authentication). The developer reasons: the internal page is only linked from the area that requires login, so a request whose Referer is that area must come from an authenticated user. The conclusion does not follow. The attacker can fabricate the header in any raw request (curl -H "Referer: http://www.company.com/loggedin/" ...), with a proxy such as Burp, or with a one-line script, and gain access without authenticating.

Do not confuse this with a time-of-check/time-of-use problem, which is a different class of weakness (a race between a check and the use of a resource, for example a file whose permissions are checked and then opened). Here the defect is that the access decision is based on data that the attacker controls: it is an authentication/authorization bypass by trusting client-supplied input. In general, any decision derived from client-controlled headers (Referer, Origin without verification, User-Agent, X-Forwarded-For, Host, custom headers like X-Is-Admin or X-Internal-Request) deserves suspicion.

The correct design is to check, on every request, a server-side verifiable credential: a session identifier that maps to an authenticated session, or a signed token. The route should be protected by a server-side check, independently of how the user got there. As a defensive measure you may use Origin and Sec-Fetch-* headers to complement CSRF protection, but never as the only authentication.

Also note that the header is optional: legitimate users with privacy settings or strict policies may not send it, causing false denials.

SESSION ID

- A value, set by the server in the HTML/Javascript
  - Kept manually by the HTML/JS logic
  - Browser is unaware of it
- URLs in the HTML include the SESSION ID

```
<a href=http://company.com?PHPSESSID=value>resource</a>
```

- SESSION ID added to requests
  - Header or explicit argument in GET actions
  - Header or body in POST actions

This is the approach used before cookies became generalised and still used by some mobile apps, APIs and legacy systems: the server generates a session identifier (SID) and the application code is responsible for carrying it. In the HTML, every link and form includes the identifier, either as a URL parameter (the example shows a PHP-style session name, PHPSESSID) or as a hidden form field; in JavaScript-heavy applications, the code adds it to a header or to the body of each request.

The weaknesses are practical. The identifier ends up in URLs, with the problems discussed for URL-based state: history, logs, Referer, shared links. It is easy to forget to add it to one link, which results in the user being logged out or, worse, a new anonymous session being created. And it enables session fixation in its simplest form: if the application accepts a SID supplied in the URL and uses it to create a session, an attacker can send the victim a link containing a known SID (explored later in the lecture).

PHP's session.use_trans_sid option automatically rewrites URLs to include the SID when cookies are unavailable; it is disabled by default and should remain so, together with session.use_only_cookies=1 and session.use_strict_mode=1 (which rejects uninitialised identifiers). Java servlet containers had a similar behaviour (;jsessionid), also best disabled.

Note the difference regarding browsers: the browser is unaware of the SID, which means that it does not add it automatically to cross-site requests. That removes the "ambient authority" of cookies (and so CSRF) but does not eliminate theft.

Cookies (RFC 6265)

- ASCII text created by the server and sent to the client
  - HTTP Header - Set-Cookie: NAME=VALUE; attributes
- Stored in the clients' cookie jar
  - A file or simple database
  - The client may freely delete (or edit) cookies
- Client resends the **`Cookie`** header to servers
  - in every request made for which there is a compatible cookie
  - Format is: **`Cookie: NAME=VALUE; NAME2=VALUE2`**

Cookies were introduced by Netscape in the mid-1990s and are currently specified in RFC 6265 (an update, usually called 6265bis, adds SameSite, prefixes and other modern behaviour). A cookie is a small name=value pair that the server sets in a Set-Cookie response header; the browser stores it in its cookie jar and automatically includes it in the Cookie request header of later requests to the matching host and path.

The cookie is fully under the client's control: users can see, modify, delete and copy cookies using browser tools or extensions, and any non-browser client can send arbitrary Cookie headers. The server must therefore assume that any cookie can be altered or invented. This is why the sensitive part of the cookie should be an unguessable identifier (or a signed/encrypted value) rather than something like role=admin or user_id=42. A real example of the opposite: authentication cookies containing base64-encoded user names or numerical identifiers, which allowed attackers to switch accounts by changing the number.

Format: the Cookie header carries all matching cookies in one line, separated by semicolons: Cookie: a=1; b=2. The server cannot see the attributes (Path, Domain, Secure, HttpOnly, SameSite, expiry) of the cookies it receives, only names and values, which has consequences: it cannot know if two cookies with the same name from different scopes are being sent, a fact used in cookie tossing attacks.

Size and number are limited (around 4 KB per cookie and a few dozens per domain), and cookies are sent in every request, including those for images and scripts, which affects performance and exposure. Keep them small and prefer server-side storage for anything else.

Cookies (RFC 6265)

- Server can keep context using the cookie provided
  - Receives a Cookie from the client
  - Fetches context (session)
  - Provides a customized answer
- Used as a **bearer token** enabling authentication of later requests
  - When set as the result of an authentication process
  - Allow obtaining the identity associated with the request
- Losing a Cookie opens the door to *impersonation*

This slide shows the usual way in which the cookie supports a session. The server creates a random identifier when the user first arrives (or when they log in), stores the session data on the server (memory, database, a cache such as Redis) under that identifier, and gives the identifier to the browser as a cookie. When a later request arrives, the server uses the cookie value to find the session, recovers the identity and other context, and generates the answer for that user.

The key security property is that the session identifier works as a bearer credential: whoever presents it is treated as the user, with no further proof. The server does not verify that the request comes from the same browser, device or person. That is the meaning of the last bullet, "Losing a cookie opens the door to impersonation": theft of the identifier by network sniffing, XSS, malware, log leakage, or physical access to the device gives the attacker the user's session for as long as it is valid.

This is also why the lifetime and invalidation policies are important. Sessions should end at logout (by destroying the server-side record, not only by deleting the cookie in the browser), after inactivity (idle timeout) and after a maximum duration (absolute timeout). When a user changes the password or the MFA settings, other sessions should be revoked. Frameworks usually provide this; the risk is custom implementations that forget one of the steps.

Compare with the alternative where the cookie is the data itself (client-side sessions, for example signed cookies in some frameworks): this avoids the server store, but the content is visible (unless encrypted) and cannot be revoked individually without extra state.

Cookies (RFC 6265)

- Cookie scope and lifetime are set by the server in the client response

```
Set-Cookie: <cookie-name>=<cookie-value>
Set-Cookie: <cookie-name>=<cookie-value>; Expires=<date>
Set-Cookie: <cookie-name>=<cookie-value>; Max-Age=<non-zero-digit>
Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>
Set-Cookie: <cookie-name>=<cookie-value>; Path=<path-value>
Set-Cookie: <cookie-name>=<cookie-value>; Secure
Set-Cookie: <cookie-name>=<cookie-value>; HttpOnly
Set-Cookie: <cookie-name>=<cookie-value>; SameSite=Strict
Set-Cookie: <cookie-name>=<cookie-value>; SameSite=Lax
Set-Cookie: <cookie-name>=<cookie-value>; SameSite=None; Secure
```


Attributes tell the browser how long a cookie lives and where it is sent. Expires and Max-Age define persistence: without either, the cookie is a session cookie and is deleted when the browser closes (though modern browsers with session restore may keep it for much longer). Domain widens the scope to subdomains: omitting it restricts the cookie to the exact host that set it, while Domain=example.com makes it available to all subdomains, which expands the attack surface if any subdomain is vulnerable or controlled by a third party (subdomain takeover). Path restricts to a URL prefix, but it is not a security boundary, because scripts on the same origin can read cookies set for other paths.

Three attributes are security controls. Secure: the browser only sends the cookie over HTTPS. HttpOnly: the cookie is not accessible through document.cookie, which prevents simple theft by injected JavaScript (it does not stop the script from making authenticated requests as the user). SameSite: Strict never sends the cookie on cross-site requests; Lax sends it on top-level navigations with safe methods (the default in current browsers when the attribute is missing); None sends it always but requires Secure, and is needed for legitimate cross-site use such as embedded widgets or federated login flows.

Prefixes provide additional guarantees: a cookie named __Secure- must be set with Secure, and __Host- also requires Path=/ and no Domain, which pins the cookie to the exact host and prevents sibling subdomains from overwriting it. For session cookies the recommended combination is __Host-name=value; Secure; HttpOnly; SameSite=Lax; Path=/, with an appropriate lifetime.

Cookies (RFC 6265)

- Client -> Server

  No cookie sent

- Server -> Client

```
Set-Cookie: MoodleSession=0r6mroovg98o338clahfd177g0; path=/
```

- Client -> Server

```
Cookie: MoodleSession=0r6mroovg98o338clahfd177g0
```


This is a concrete trace of the cookie lifecycle, using a real session cookie from the university's platform. In the first request the browser has no cookie for the site, so none is sent. The server answers with Set-Cookie and a randomly looking value. From that moment, every request to that host includes the Cookie header with that value, and the server uses it to find the session.

Several security questions can be asked about this trace. How long is the identifier, and where does its randomness come from? The value shown has 26 characters from a base-36-like alphabet, i.e. about 130 bits if generated by a good random number generator, which is adequate; the minimum recommended by OWASP is 64 bits of entropy, and 128 bits is a comfortable target. Was the cookie sent with Secure and HttpOnly? Here not, and the missing flags would be flagged in an assessment. Does the identifier change when the user authenticates? If it does not, the application is vulnerable to session fixation. Does it stay valid after logout? Test by logging out and replaying the old cookie from curl; a vulnerable site accepts it, which means that logout only deleted the cookie from the browser.

A practical exercise: log in on a test site, copy the cookie from the developer tools, and use curl with -H "Cookie: ..." to access an authenticated page from the command line. This demonstrates without any exploit that possession of the cookie is equivalent to being logged in, and it motivates every protection discussed in the rest of the lecture.

JWT - JSON Web Tokens

- Concatenation of 3 texts
  - **base64url(header) + '.' + base64url(payload) + '.' + base64url(signature)**
  - signed with HMAC or asymmetric crypto (RSA, ECDSA); payload is encoded, not encrypted

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

payload = {"loggedInAs" : "admin","iat" : 1422779638}

signature=HMAC-SHA256(secret,b64url(header)+'.'+b64url(payload))
```


A JSON Web Token (RFC 7519) is a compact, URL-safe string with three parts separated by dots: a header that declares the type and the signature algorithm, a payload with claims (who the subject is, who issued the token, when it expires, and any application data), and a signature. The first two parts are encoded using base64url (the URL-safe variant, without padding), not plain base64. In the common case (JWS), the token is signed but not encrypted: anyone who obtains it can decode and read the claims. Do not put secrets or unnecessary personal data in the payload. Encrypted tokens (JWE) exist but are less common.

The signature protects integrity and authenticity: with HMAC (HS256) the same secret is used to sign and verify, so every service that verifies tokens can also forge them; with asymmetric algorithms (RS256, ES256) the issuer signs with a private key and any service can verify with the public key, which fits distributed architectures and third-party identity providers better.

Standard claims include iss (issuer), sub (subject), aud (audience), exp (expiration), nbf (not before) and iat (issued at, a Unix timestamp as in the example). Verification must check the signature and the claims: expiry, issuer and audience. Skipping the audience check allows a token issued for one service to be used on another.

The attraction of JWTs is statelessness: the server needs no session store, which helps scaling and microservices. The price is revocation: a valid token remains valid until it expires, unless the server keeps a denylist, which reintroduces state. Short lifetimes are therefore essential.

JWT - JSON Web Tokens

- Provide mechanisms for token refresh, limiting impact due to a lost token
- Access Token – JWT Token that authorizes the user – very limited lifespan
  - Is used in every request and has higher exposition
- Refresh Token – Random Token only to refresh Access Token
  - Only used to refresh the Access Token
  - Longer lifetime
  - Should be single-use (rotated on refresh) and revocable
- After all tokens expire, the authentication process must be restarted

```
payload = {"loggedInAs" : "admin","iat" : 1422779638}
```


Because JWTs are self-contained and hard to revoke, the usual pattern is to combine a short-lived access token with a longer-lived refresh token. The access token (minutes, typically 5 to 15) is sent with each API request and, if stolen, is useful to an attacker only for a short window. The refresh token is kept more carefully, is sent only to the authorisation endpoint to obtain new access tokens, and lives for days or weeks. When it expires, the user must authenticate again.

Refresh tokens are higher-value targets because they are long-lived, so they need extra protection. They are usually random opaque strings stored in the server (hashed) so that they can be revoked individually, associated with a device or client, and rotated: each use returns a new refresh token and invalidates the old one. If an old token is presented again, it is a sign of theft (reuse detection), and the entire token family should be revoked. In browsers, keep them in HttpOnly, Secure, SameSite cookies limited to the refresh path, instead of localStorage.

Where should the access token be stored in a browser application? There is no perfect answer. Memory-only storage protects against persistence but is lost on reload; localStorage is readable by any script (XSS gives theft); an HttpOnly cookie is not readable by scripts but is sent automatically, which brings back CSRF concerns that must be addressed with SameSite and anti-CSRF tokens. Patterns such as a backend-for-frontend that holds the tokens and gives the browser an ordinary session cookie are increasingly recommended by the OAuth security guidance for browser-based applications.

The payload at the bottom of the slide repeats the example claim set, which makes clear that it is readable, and that the "admin" claim is only as trustworthy as the signature check.

Exploitation and Prevention

The second half of the lecture turns to attacks and defences. A helpful framework for the weaknesses that follow is to follow the lifecycle of a credential: how it is created (registration, default accounts, recovery), how it is stored (server-side hashing, client-side storage), how it is transmitted (TLS, tokens in URLs), how it is verified (rate limits, MFA, constant-time comparison, error messages) and how it ends (logout, expiration, rotation, revocation). A weakness at any stage lets an attacker obtain or bypass a credential.

Also separate two attacker positions. Remote, unauthenticated attackers target the login endpoint, the recovery flows and the session identifiers; they are limited by network visibility and rate limiting. Attackers with a foothold (a stolen database, a compromised workstation, an XSS vulnerability, a position on the network) already have secrets or traffic and try to convert them into access. Defences must be designed for both: online defences (throttling, MFA) and offline ones (strong password hashing, short token lifetimes, HttpOnly cookies, TLS).

For each issue presented, try to answer three questions: what does the attacker gain, what does it cost the attacker, and which control changes that equation most. Often the cheapest effective control is a design decision, such as not using passwords at all, rather than a patch.

What makes an application vulnerable?

- Credentials can be guessed or overwritten as a result of weak account management functions
  - Default passwords, weak passwords (low entropy), predictable passwords
  - Broken account creation/recovery process

- Allow automated guessing: brute-force, dictionary, credential stuffing
  - Brute force: test the key space by blind variation of characters
  - Dictionary / stuffing: common passwords or pairs leaked elsewhere

Weak credentials are still the most common root cause of account compromise. Default passwords (admin/admin, vendor-installed accounts on routers, cameras and management consoles) are the basis of botnets such as Mirai, which recruited IoT devices using a list of about sixty factory credentials. Weak passwords follow patterns: dictionary words with a number, keyboard walks, seasons and years, or the name of the service. Password policies that force composition rules (one capital, one digit, one symbol) lead users to predictable transformations such as Password1!.

Entropy is the key concept. A randomly chosen eight-character password from 94 printable characters has about 52 bits of entropy, but human-chosen ones have far less, and a passphrase of four random words from a 7776-word list (Diceware) has around 51 bits while being easier to remember. NIST SP 800-63B recommends allowing long passphrases and checking against lists of compromised and common passwords, instead of composition rules.

Broken account creation and recovery flows are the other main branch: registration that allows impersonation (registering an existing e-mail with a trailing space or different capitalisation), recovery that uses weak questions, or administrative "reset password" endpoints that do not verify the requester.

The second bullet groups the automated guessing attacks, which are explained in detail in their own slide. The "key space" language comes from cryptography: the attacker is trying to find the secret by testing candidates, and the cost is the number of candidates divided by the attack rate. Defenders raise the cost per guess (slow hashes, rate limits) or reduce the attacker's useful candidate set (block common passwords).

What makes an application vulnerable?

- Credentials are stored in a vulnerable format
  - Clear text, reversible encryption, fast unsalted hashes (MD5, SHA-1), low PBKDF2 iterations
  - Attacker can conduct an offline attack
    - Offline attacks are dangerous:
      - The victim doesn't know the resources of the attacker
      - The attack is silent to the victim and can take days-years
- Credentials are sent over unencrypted connections
  - Or authentication driven tokens (cookies)
- Recovery or Multi-Factor is broken/missing

Storage and transmission are the other two places where credentials fail. If the database stores passwords in clear text, any read access (SQL injection, backups, a misconfigured replica, an insider) discloses all accounts at once, and because users reuse passwords, also their accounts on other services. Reversible encryption is nearly as bad: the key must be available to the application, so an attacker who compromises the application obtains it too. Fast, unsalted hashes (MD5, SHA-1, SHA-256 on their own) can be attacked with rainbow tables and GPUs at billions of guesses per second. A single modern GPU can test tens of billions of MD5 hashes per second.

Offline attacks are the most dangerous because they happen after a data breach and with no interaction with the system: no rate limit, no logs, no alerts. The attacker works with their own hardware for as long as they want, and the defender may not even know that the database has leaked. The only protection is the work factor and unique salt already stored with each hash, and the strength of the original password. Public breach corpora show the consequence: the weaker half of most leaked databases is cracked in hours.

Unencrypted transport allows interception of credentials and tokens by anyone on the path: public Wi-Fi, a rogue access point, a compromised router, or an ISP. The mention of "authentication driven tokens" means that cookies are as sensitive as passwords. HTTPS must cover the entire session, not only the login page, and should be enforced with HSTS.

The final bullet concerns recovery or MFA: if recovery is weaker than login (for example, answering a security question or receiving an SMS), the attacker will choose the recovery path. The overall security is that of the weakest path to the account.

Credential guessing attacks

- Brute force
  - Tries every combination: only feasible for short or low-entropy secrets
  - Online is slow and noisy; offline against stolen hashes is fast
- Dictionary and credential stuffing
  - Dictionary: common passwords and their mutations
  - Stuffing: leaked username/password pairs replayed on other sites – relies on reuse
  - Automated with botnets and rotating proxies
- Password spraying (CAPEC-565)
  - One or few common passwords tried against many accounts
  - Stays below per-account lockout thresholds

This slide separates four attacks that are often mixed up, because the countermeasures differ.

Brute force tries candidates systematically (all strings up to some length). Against an online login it is slow and noisy, so it is usually applied to short or numerical secrets (PINs, four-digit codes, short OTPs, weak reset tokens). Against stolen hashes it is limited only by hardware, which is why hash cost parameters matter. Typical tools: Hydra, Medusa, Burp Intruder, ffuf for HTTP; hashcat and John the Ripper for offline hashes.

Dictionary attacks try likely passwords from wordlists (RockYou, SecLists, lists derived from breaches), often with mangling rules (capitalise, append digits, leetspeak substitutions). They are far more efficient than brute force because human choices are concentrated.

Credential stuffing takes username/password pairs leaked from one service and tries them on others. Since most people reuse passwords, success rates of 0.1% to 2% are typical; with millions of pairs that means thousands of accounts. It is industrialised: automation frameworks, residential proxy networks to evade per-IP limits, and CAPTCHA solving services. Defences are MFA, breached password screening, device fingerprinting and bot detection, and monitoring for spikes in failed logins with many distinct usernames.

Password spraying inverts the order: one or a few very common passwords (Spring2026!, Welcome1) against many accounts, keeping the number of attempts per account below the lockout threshold. This is a favourite against corporate single sign-on portals and VPN gateways. Defences: detect many accounts failing from the same source or with the same password, apply global and per-IP limits, and ban common passwords. Identified as CAPEC-565.

What applications should do

Avoid passwords as much as possible (CWE-309)

- Passwords are prone to have low entropy, follow patterns
  - If required: enforce length, block breached/common passwords (CWE-521); avoid composition rules
- Users frequently reuse credentials among different services
- Servers must store only a salted, slow hash – never the password
  - Clients should use password managers or passwordless methods

The best way to protect a password is not to have one. CWE-309 (Use of Password System for Primary Authentication) captures that passwords are a weak primary mechanism: humans cannot memorise many strong unique secrets, reuse them, are tricked by phishing, and the server must hold verification data that attackers want to steal. Where feasible, delegate authentication to a trusted identity provider, use passkeys (WebAuthn/FIDO2) or use hardware-bound credentials, and keep passwords as a fallback.

If passwords remain, follow current NIST SP 800-63B guidance, which differs from many legacy policies. Require a minimum length (at least eight characters, and in practice more, like 12 or longer), allow long passphrases (up to 64 characters or more) and all printable characters including spaces and Unicode, do not impose composition rules, and check candidates against lists of breached and common passwords and context-specific words (user name, service name). Do not use security questions or password hints. Allow pasting so that password managers work. Offer a strength meter that reflects real guessability, such as the zxcvbn estimator.

Regarding storage: the server should keep only a verifier, never the password. The slide mentions that clients may store passwords, which should be understood as password managers and the browser's credential store, which are acceptable, preferably with the master secret protected by device security, rather than the application keeping the password in cookies or local storage, as some "remember me" implementations did. A "remember me" feature should use a separate, random, revocable token bound to the user and device, stored hashed on the server.

Reuse across services is the central risk, which is why breached password screening, such as the Pwned Passwords k-anonymity API, is valuable: only the first five characters of the hash leave your system.

What applications should do

Use secure storage (CWE-257)

- Do not store passwords in clear, even if the domain is “secure”
  - E.g. database requires authentication to be accessed
- Use a slow, salted password hash (CWE-916)
  - Argon2id, scrypt, bcrypt or PBKDF2, with a unique random salt
  - Tune cost (iterations/memory) so each hash takes ~100s of ms
- Direct access to storage may reveal secrets
  - Directly or through brute force

Password storage must resist an attacker who has the complete database. Use a password hashing function designed to be expensive: Argon2id (the current recommendation; memory-hard, which hurts GPUs and ASICs), scrypt, bcrypt, or PBKDF2 with a high iteration count when FIPS compliance requires it. General-purpose hashes (MD5, SHA-1, SHA-2) are too fast for this purpose. Each password must have a unique random salt (at least 16 bytes), which prevents the use of precomputed tables and makes identical passwords produce different hashes. Many libraries store the algorithm, parameters and salt in a single string, which also makes upgrades easier.

Parameters should be tuned to the hardware so that one verification takes in the order of 100 milliseconds to a second; the OWASP Password Storage Cheat Sheet currently suggests, for example, Argon2id with 19 MiB of memory, 2 iterations and one degree of parallelism as a minimum, or bcrypt with a cost of 10 or more (noting its 72-byte input limit), and PBKDF2-HMAC-SHA256 with several hundred thousand iterations. Costs should be reviewed periodically and hashes upgraded transparently when the user next logs in. An optional pepper, a secret key held outside the database (HSM or secret manager), adds protection if only the database is stolen.

CWE-257 describes storing passwords in a recoverable format, and CWE-916 describes using a hash with insufficient computational effort. Notice that this also affects features: if you can send a user their password by e-mail, you are storing it recoverably, and you must change the design to send a reset link instead.

Also compare passwords with a constant-time function and avoid revealing which part (user or password) failed. For the note about "domain is secure": database access control is not a reason to skip hashing; defence in depth assumes the perimeter will eventually fail.

What applications should do

Don’t force periodic password rotation; require a change when compromise is suspected (NIST 800-63B)

- Rotation is still important for long-lived secrets: API keys, tokens, certificates
  - Stolen/discovered secrets will be rendered useless
  - Doesn’t depend on users good practices (it’s imposed by system)
- Forced frequent rotation is counter-productive: users create “algorithms”
  - 01MocKingBird2020, 02MocKingBird2020...
  - Frequent expiration will impact usability and increase the security burden

Password rotation is a good example of advice that was wrong for years. For decades, organisations forced users to change passwords every 60 or 90 days. Research (for example, studies on university accounts showing that most new passwords could be guessed from old ones within a few attempts) showed that forced expiration leads to predictable transformations such as incrementing a number, and increases the number of passwords written down or forgotten, with help-desk and recovery abuse as consequences. NIST SP 800-63B (since 2017) and the UK NCSC advise against periodic changes for user-memorised passwords, while requiring a change if there is evidence of compromise, such as a detection in a breach list or suspicious activity.

That does not mean that rotation is useless. It is valuable for secrets that are not memorised by humans and that can be rotated automatically: API keys, service account passwords, database credentials, signing keys, TLS certificates (now with lifetimes of 90 days or less in many cases) and session/refresh tokens. Automated rotation limits the time window for a stolen secret, does not depend on user behaviour, and a secrets manager makes it routine. Short-lived credentials (for example, cloud tokens that expire in an hour) are the extreme form of this idea.

Applications should offer a way for users to change their password, require the current password for it, invalidate other sessions after a change, notify the user by a trusted channel, and check the new one against the same rules and lists as for registration.

Be careful with the CWE references: CWE-263 refers to password ageing with long expiration, and it is better regarded as a weakness of systems that never expire non-human credentials, not as a requirement for human password rotation.

What applications should do

Rate limit authentication functions (CWE-307)

- Online guessing will be dramatically slowed
  - Even a small delay of hundreds of milliseconds may be useful
- Monitoring authentication functions allows detecting attack attempts
  - Prefer progressive delays over hard lockout, which enables denial of service
  - Password spraying evades per-account limits (CAPEC-565): also limit per IP and globally

Rate limiting and monitoring address online guessing, and CWE-307 (Improper Restriction of Excessive Authentication Attempts) is the weakness when they are absent. The design goal is to make large-scale guessing impractical without giving the attacker a way to lock out legitimate users.

Common techniques: increasing delays after failed attempts (exponential back-off) per account and per source, limits per IP address and per network range, global counters that detect distributed attacks, CAPTCHA or proof-of-work challenges after a threshold, and temporary rather than permanent lockouts. Hard lockout after N failures stops guessing against one account, but gives the attacker a denial of service: they can lock any user by submitting wrong passwords. It also does not stop password spraying, where each account sees only a few attempts. A better practice is to throttle per account with notifications, apply stricter checks on unknown devices, and keep a trusted device path available for the legitimate owner.

Responses must be uniform: the same message and similar timing for unknown users and wrong passwords, otherwise attackers can enumerate valid accounts first (see the section on enumeration). Remember that limits must be applied to every authentication endpoint, including APIs, mobile endpoints, password reset, MFA verification and "check if account exists" calls, because attackers choose the least protected one. A frequent defect is that the web login is protected and the mobile API is not.

Monitoring means logging failures with enough context (user, source, user agent, outcome) while never logging the password itself, and alerting on anomalies such as many failed logins across many accounts, or a successful login after many failures. These events feed into incident response and into automated blocking.

What applications should do

Use Multi Factor Authentication (CWE-308)

- One attack rarely compromises every factor
  - Password: phishing, eavesdropping or database leak
  - Smartphone: remote compromise or physical steal
  - Hardware token: physically steal the token
  - SMS codes are weak (SIM swap): prefer TOTP or FIDO2/WebAuthn
- If it is a usability issue, use progressive multi factors
  - Risk-based: ask for an extra factor when context changes (new device, IP, location)
- Favor multi-factor authentication in recovery processes

Multi-factor authentication is the single most effective control against credential-based attacks. Microsoft's analysis of compromised accounts has suggested that MFA blocks over 99% of automated account compromise attempts, and Google's research on account takeover found that SMS codes blocked all automated bots and 96% of bulk phishing, but only 76% of targeted attacks, while security keys blocked all three. CWE-308 describes the weakness of relying on single-factor authentication for critical functions.

The idea is that the attack needed to get one factor does not give the others. A phishing site or database leak gives the password, but not the phone or the key; stealing the phone does not give the password. But MFA methods differ in strength. SMS and voice codes are vulnerable to SIM swapping, SS7 weaknesses and interception by malware; e-mail codes depend on the mailbox security; TOTP applications are better but can be phished in real time by a reverse proxy (Evilginx-style attacks) that relays the code; push notifications suffer from "MFA fatigue", in which attackers spam approval requests until a user accepts, which was used in several high-profile intrusions, mitigated by number matching. FIDO2/WebAuthn security keys and passkeys are bound to the site's origin and therefore resistant to phishing.

Usability matters: if MFA is too intrusive, people look for ways around it. Risk-based or progressive approaches ask for an additional factor only when the context changes, as on the slide: new device, new network, unusual location or sensitive operation. The cookie remembered on a trusted device is a convenience marker, not a factor, and it needs the protections discussed for sessions.

Recovery is the usual back door: if a user can disable MFA with only an e-mail link, the attacker will do that. Recovery codes, a second device, or a waiting period with notifications are better options.

What applications should do

- For reference: NIST SP 800-63B – Authenticator Assurance Levels
  - AAL1: single factor is acceptable (e.g. password)
  - AAL2: two distinct factors, approved cryptography
  - AAL3: hardware cryptographic authenticator, resistant to phishing

NIST SP 800-63B, part of the Digital Identity Guidelines, defines three authenticator assurance levels (AAL) that organisations can select according to risk. At AAL1 some confidence is provided that the claimant controls an authenticator; single-factor authentication is permitted (for example a password), and it is for low-risk applications. At AAL2 high confidence is required and two distinct factors are mandatory, using approved cryptography; in the latest revision (SP 800-63-4) a phishing-resistant option must be offered at AAL2, and phishing resistance is required at AAL3. At AAL3, very high confidence is required: authentication with a hardware-based cryptographic authenticator that provides verifier impersonation resistance, normally with a multi-factor cryptographic device such as a FIDO2 security key or smart card, and the key must not be exportable.

The levels offer a vocabulary for deciding which mechanisms are acceptable for which risk, in a way that regulators, auditors and procurement can reference. For example, a public forum might be at AAL1, a banking transaction or administration console at AAL2, and the control of an organisation's identity provider or a privileged government system at AAL3.

The document also holds many practical rules that we have used in this lecture: no periodic rotation of memorised secrets, no composition rules, use of breached password lists, rate limiting of failed attempts (no more than 100 consecutive failures per account is a stated upper bound), and requirements for session management and reauthentication. The related documents are 63 (enrolment and identity proofing) and 63C (federation). If you design or evaluate an authentication system, this is one of the best references to read in full.

Account recovery and user enumeration

- User enumeration (CWE-204)
  - Different messages, status codes or timing for unknown user vs wrong password
  - Affects login, registration and password reset: use generic responses
- Weak recovery flows
  - Security questions with guessable or public answers
  - Predictable, reusable or non-expiring reset tokens
  - Reset link built from the Host header (poisoning)
- Safer recovery
  - Random, single-use, short-lived token (stored hashed)
  - Notify the user and invalidate active sessions after a reset

Account recovery is an alternative way to obtain access to an account, and therefore it has to be at least as strong as the login itself. Real-world takeovers frequently abuse recovery and not the login: public examples include answers to security questions found on social media, password reset e-mails intercepted via a compromised mailbox, telephone-based social engineering, and SIM swaps to receive reset codes.

User enumeration (CWE-204, Observable Response Discrepancy) is the first step in many attacks. The login says "user not found" versus "wrong password"; the registration form says "e-mail already registered"; the reset form says "no account with this address"; or the response time differs because a hash is computed only for existing users. Attackers use this to build target lists for spraying and phishing. Countermeasures: identical messages and status codes ("if the account exists, we sent an e-mail"), similar timing (for example by hashing a dummy value for unknown users), and rate limiting. For registration, where uniqueness must be shown, send an e-mail to the address instead of showing the result on the page.

Reset tokens should be random (at least 128 bits from a CSPRNG), single use, short-lived (minutes to an hour), stored hashed in the database, and tied to the account. Weak designs include tokens derived from timestamps or user IDs, MD5 of the e-mail address, or sequential numbers. Another classical flaw is Host header poisoning: the application builds the reset link from the Host header of the request, so an attacker triggers a reset for the victim with a header pointing to a domain they control, and the victim's click delivers the token to them. Build links from a configured base URL.

After a successful reset, invalidate existing sessions and tokens, notify the user, and avoid logging the person in automatically without MFA.

Delegated and passwordless authentication

- OAuth 2.0 and OpenID Connect
  - OAuth 2.0 delegates authorization; OIDC adds identity (ID token is a JWT)
  - Pitfalls: missing state/PKCE, loose redirect_uri, token leakage
- Passwordless: WebAuthn / FIDO2 passkeys
  - Public-key crypto: the private key never leaves the authenticator
  - Bound to the origin: resistant to phishing and replay
  - The server stores no shared secret
- Single Sign-On
  - A compromised identity provider account opens every service: enforce MFA
  - Validate signature, issuer, audience and expiry of every token

Many modern applications do not manage passwords at all. They delegate authentication to an identity provider (IdP) such as an enterprise directory, Google, Microsoft or a national eID, using protocols such as SAML (older, XML-based, common in enterprises) or OpenID Connect (OIDC). It is important to understand that OAuth 2.0 is an authorisation framework: it allows an application to obtain limited access to resources on behalf of the user. OIDC adds an identity layer on top, by returning an ID token (a JWT) that states who authenticated and how. Using a plain OAuth access token as proof of identity is a known mistake.

Common pitfalls include: not checking the state parameter (CSRF on the login callback), not using PKCE for public clients such as mobile and single-page applications (authorisation code interception), accepting any redirect_uri or wildcard patterns (open redirect leading to token theft), using the implicit flow, not validating issuer, audience, expiry, nonce and signature of tokens, and trusting email claims that the IdP has not verified. The OAuth 2.0 Security Best Current Practice and OAuth 2.1 consolidate the recommendations.

Passwordless authentication with WebAuthn and FIDO2 (passkeys) replaces the shared secret by a public key pair: during registration the authenticator creates a key pair for the specific site (origin), and the server stores only the public key. During login the server sends a challenge and the authenticator signs it after a local user verification such as a fingerprint or PIN. Because the credential is scoped to the origin, a phishing site on another domain gets nothing, and a database leak exposes no secrets. Passkeys synchronised through platform providers improve usability, with some trade-offs in recovery and trust in the provider.

Single Sign-On concentrates risk: one compromised account can open many services, so the IdP

deserves the strongest protection and MFA.

Token exploitation

- Client may freely manipulate tokens to trigger an attack
  - Break the authentication process, Enumerate users, Bypass authentication
- Cookies
  - If contain sensitive information (passwords) – CWE-256
  - If they have low entropy
  - If they have a structure that is processed in the server
- JWT
  - If server improperly verifies signature and allows the “none” method
    - Server must enforce the expected algorithm, never trusting the token’s alg header
  - Weak HMAC secret allows offline brute force and forging of tokens

When the client holds state, the client can attack it. Token exploitation means modifying, forging or replaying the values that the server accepts as proof of identity or authorisation. The checklist is useful for testing: decode every cookie and token, look for readable structure (base64 or hex of usernames, roles, timestamps), try changing fields, check whether the server validates a signature or a MAC, and gather many samples to analyse their randomness (Burp Sequencer is made for this).

Cookies that contain sensitive information, such as passwords or personal data, expose them to the user and to anyone who steals the cookie (CWE-256 refers to plaintext storage of a password). Cookies with low entropy, or with sequential or time-based values, can be guessed. Cookies that are processed by the server as structured data, such as serialised objects, are a deserialisation risk and may lead to remote code execution in addition to authentication bypass.

JWT-specific attacks are well known. The "none" algorithm: the header says alg=none, and a vulnerable library accepts the token without verifying any signature, so the attacker can edit the claims freely. Algorithm confusion: a server expecting RS256 receives a token with alg=HS256 and uses the public key (which is, by definition, public) as the HMAC secret, so the attacker can sign tokens. The defence is to never let the token choose its algorithm: configure the expected algorithm and keys in the server and reject everything else. Weak HMAC secrets can be brute-forced offline from a single captured token with hashcat, so use random secrets of at least 256 bits. Header parameters such as kid, jku and jwk have also led to key injection and path traversal bugs, so they must be handled with a strict allow list.

Always use maintained libraries, verify exp, iss and aud, and keep access tokens short-lived.

Session Hijacking

- Web applications use session ids, cookies and tokens as credentials
  - Stealing this credential will result in session hijacking
  - SESSION IDs and tokens reside in RAM
  - Cookies are stored, and present in backups
  - BAD: sometimes use IP Address as SID (CWE-291)

- Multi-factor authentication may limit exploitation
  - Cookie from a different IP address? Require re-authentication
  - Cookie from a different browser? Require re-authentication

Once a user has authenticated, the session identifier or token is the credential, and stealing it is usually easier than breaking the password or the second factor: it also bypasses MFA, because the proof was already given. This is the logic behind many real attacks, including cookie-stealing malware (infostealers) that collect browser cookie databases, which are sold in bulk and used to access corporate cloud accounts without ever seeing a password or an MFA prompt.

Where does the credential exist? Session IDs and tokens live in server memory or in a session store, in the browser's memory, and, in the case of persistent cookies, on disk, where they may be present in backups, synchronised profiles, and forensic images. Anyone with access to the device profile can take them. Tokens in localStorage or in URLs have additional exposure paths.

Binding a session to context is a way to reduce the value of a stolen token. Using the client's IP address as the session identifier, or as a requirement, is poor design (CWE-291, Reliance on IP Address for Authentication): addresses are shared behind NAT, change when users move between Wi-Fi and mobile networks, and can be spoofed or reused. Using IP changes as a risk signal is more defensible: when the context changes strongly (different country, different browser family) require re-authentication or MFA instead of silently invalidating or accepting. Device-bound credentials such as DPoP or mTLS-bound tokens and Device Bound Session Credentials are emerging to make stolen cookies useless on another machine.

Key controls: short idle and absolute timeouts, regenerate on privilege changes, re-authenticate for sensitive actions (change e-mail, change password, payments), allow users to see and revoke active sessions, and monitor for anomalies.

Session Hijacking – Sniffing/Interception

- Sessions can be stolen from Cookie repository
  - If device is compromised
  - Read by injected scripts (XSS) if not HttpOnly
  - If they do not expire (CWE-613)

- Browser can be led to provide the Cookie/Token to a malicious server
  - Attacker spoofs the DNS reply for http://company.com and provides the address of the malicious server
  - MiTM attacks with non secure (no TLS) services

This slide lists the ways in which an attacker may capture session credentials in transit or at rest. Cookie repositories can be read by malware or by anyone with access to the unlocked device or its backups. Sessions that never expire (CWE-613, Insufficient Session Expiration) turn any past theft into long-term access; this includes persistent cookies lasting months and sessions that survive logout or password change. Cross-site scripting is the most common web route: if the cookie is not HttpOnly, a script injected in the page can read document.cookie and send it to the attacker. Even with HttpOnly, the script can perform actions as the user, but cannot export the session for use elsewhere.

The network attacks depend on missing TLS. On an open Wi-Fi network, a passive observer sees the cookies of any plain HTTP request (the Firesheep extension in 2010 made this a mainstream issue and pushed large sites to adopt HTTPS everywhere). An active attacker can also mount man-in-the-middle attacks. In the slide's DNS scenario, the attacker responds to the victim's DNS query faster than the real resolver (spoofing) or poisons a cache, and the browser connects to the attacker's server. If the site is served over HTTP, or the cookie lacks the Secure flag, the browser sends the cookie to the attacker. With HTTPS and a valid certificate this attack fails, because the attacker cannot present a certificate for the domain.

Defences: HTTPS everywhere with HSTS (and preloading), the Secure attribute, HttpOnly, short lifetimes, DNSSEC and encrypted DNS where available, certificate transparency monitoring, and a strong Content Security Policy to reduce XSS impact. Also avoid mixed content, since a single HTTP resource may expose cookies not marked Secure.

Session Hijacking – Brute Force

- SIDs and Cookies must have high entropy (CWE-331)
  - Must come from a cryptographically secure random generator (≥128 bits)
  - Caveat: calculating a hash from a timestamp is a bad pattern (CWE-330)
    - Timestamp is predictable
- Otherwise attacker may brute force values of active sessions
  - Send multiple requests with varying SID/Cookies until access is granted
- Same can be done for username/passwords
  - Passwords are weak links
  - User enumeration will reduce the attack time (CWE-204)
  - Seen in many products (see CVEs)

If the identifiers are guessable, an attacker does not need to steal them: they can try values until one works. This works because sessions of other users are valid at the same time, and a random guess has a chance of matching one of them. The probability depends on the identifier space and the number of active sessions: with N active sessions among 2^b possibilities, each guess succeeds with probability N/2^b. With 128 bits, this is negligible for any realistic N and request rate; with 32 bits, it is feasible.

The classical failures are not obvious random sequences. Incrementing counters; time-based values (a hash of the timestamp, as in the slide, has only a few million possibilities per hour); values derived from the username, IP address or other predictable data; or non-cryptographic random generators (rand(), Math.random(), java.util.Random, or PRNGs seeded with the time), whose output can be predicted after observing a few values. The correct approach is to use the platform's cryptographically secure random number generator (/dev/urandom, secrets in Python, crypto.randomBytes in Node, SecureRandom in Java) with at least 128 bits, as OWASP recommends (at least 64 bits of entropy as a minimum). A UUIDv4 contains 122 random bits and is acceptable only if the library draws them from a CSPRNG, while other UUID versions (time-based or name-based) are predictable and must not be used. A hash does not add entropy: it merely transforms the input.

Burp Sequencer or similar tools estimate the entropy of captured tokens statistically. The slide notes that the same technique applies to usernames and passwords; user enumeration lets attackers reduce the search space to valid users. In this context, CWE-204 is the right reference for the observable discrepancy, as CWE-200 is a general information exposure.

Session Fixation

- A new SID must be issued on login and privilege change
  - **The old one must be invalidated (CWE-384)**
  - Never accept SIDs from the URL or unknown values (strict session mode)
- Attacker may force a predictable SID and wait for authentication
  - SID will be kept after authentication, granting access to the attacker
  - Hey Alice, check this https://server.com?SID=KNOWN_TO_ATTACKER
- Detected by observing the Cookie/SID before and after authn

In session fixation the attacker does not steal the victim's session; they make the victim use a session that the attacker already knows. The attacker obtains or creates a session identifier, gets the victim to use it (a link, a cookie set from another site, or an injected header), waits for the victim to log in, and then uses the same identifier, now associated with an authenticated session. It works when the application keeps the same identifier across the transition from anonymous to authenticated, or accepts identifiers that it did not generate.

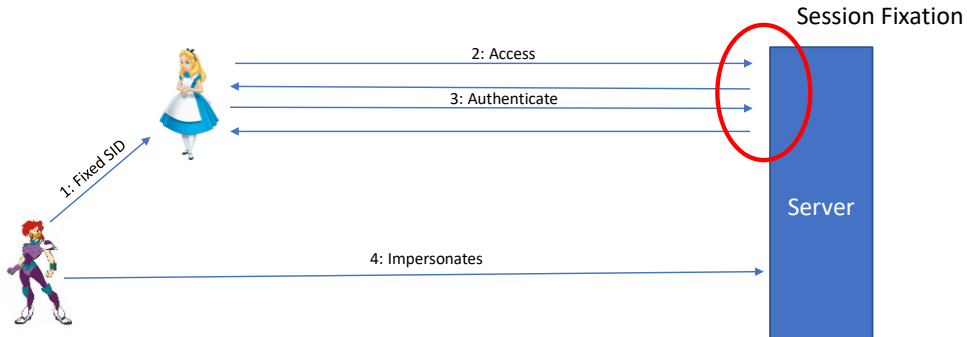
The fix is simple and mandatory: issue a new identifier whenever the privilege level changes (login, logout, role change, MFA completion), and destroy the old session. Frameworks provide this (session_regenerate_id(true) in PHP, request.changeSessionId() in Servlets, cycle_key in Django, and so on), but it needs to be called, and custom login code often forgets. In addition: accept session identifiers only from cookies and not from URL parameters; reject identifiers that the server did not issue (strict mode, so that a client cannot choose its own); use cookies with the __Host- prefix so that sibling subdomains cannot set them; and consider sending a fresh anti-fixation token.

Detection is done by comparing the session identifier before and after authentication in a test. If it does not change, the application is vulnerable. This can be done in a few seconds with the browser developer tools or a proxy, and it is a standard item in the OWASP Web Security Testing Guide (testing for session fixation, WSTG-SESS-03). CWE-384 is the corresponding weakness.

The "secondary cookie" idea that was common in older guidance binds a random value to the session, but it is a mitigation and not a replacement for regenerating the identifier.

Session Fixation - Scenarios

- Freely controlled SID
  - Attacker says: Hey Alice, check this https://server.com?SID=KNOWN_TO_ATTACKER
  - If Alice accesses the URL and logs on
  - The attacker can use the SID to impersonate Alice



The first scenario is the simplest and shows the entire pattern. The application accepts a session identifier chosen by the client, typically from a URL parameter. The attacker crafts a link with a value that they know (step 1), the victim opens it (step 2), and the server starts a session with that identifier. The victim then logs in (step 3), and since the identifier is not regenerated, the session now has the victim's identity, and the attacker, who has the identifier, can use it (step 4).

This was real in early versions of PHP and Java applications that supported URL-based sessions, and examples have been reported in banks and webmail services. The attack requires that the attacker can deliver the link (phishing, a message, an advertisement) and that the victim logs in afterwards. It does not require breaking any cryptography or network position.

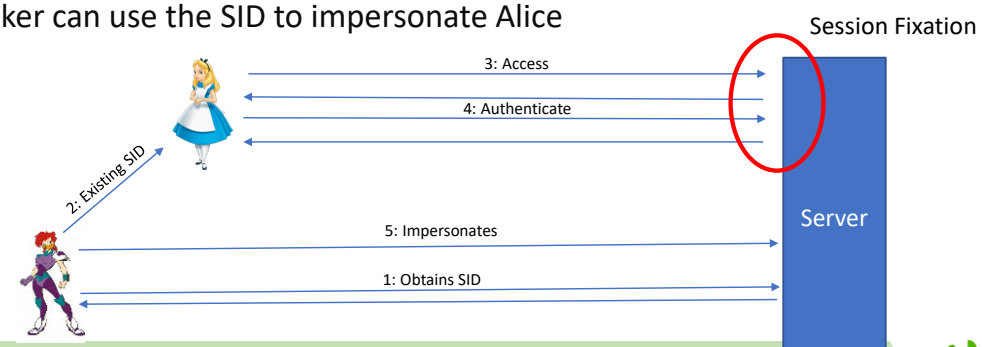
Practical notes. It is not necessary that the identifier be passed in the URL: any place where the attacker can influence the victim's cookie works, such as a vulnerable XSS that sets document.cookie, a response splitting vulnerability, or a related subdomain (see the third scenario). Browsers also have the "cookie jar overflow" and cookie tossing techniques that let one origin overwrite a cookie for another.

A test: visit the login page anonymously, note the session cookie value, authenticate, and look at the cookie again. Then try setting your own made-up value before authenticating: if the server accepts it and keeps it after login, the application accepts attacker-chosen identifiers and doesn't rotate them.

Mitigation is the combination described for session fixation in general: strict mode (the server only accepts identifiers that it created), and regeneration at login.

Session Fixation - Scenarios

- Pre-Generated SID
  - Attacker obtains SID from server
  - Attacker says: Hey Alice, check this https://server.com?SID=EXISTING_SID
  - If Alice accesses the URL and logs on
  - The attacker can use the SID to impersonate Alice



In this variant the application does not accept arbitrary identifiers. The attacker first interacts legitimately with the server to obtain a valid, server-generated identifier (step 1): many applications create a session for anonymous visitors, so a simple request is enough. Then the attacker plants this valid identifier in the victim's browser (step 2), for instance through a link when URL sessions are accepted. After the victim authenticates (step 4), the session associated with this identifier is upgraded to the victim's identity without changing the identifier, and the attacker (step 5) just needs to use the identifier that they already hold.

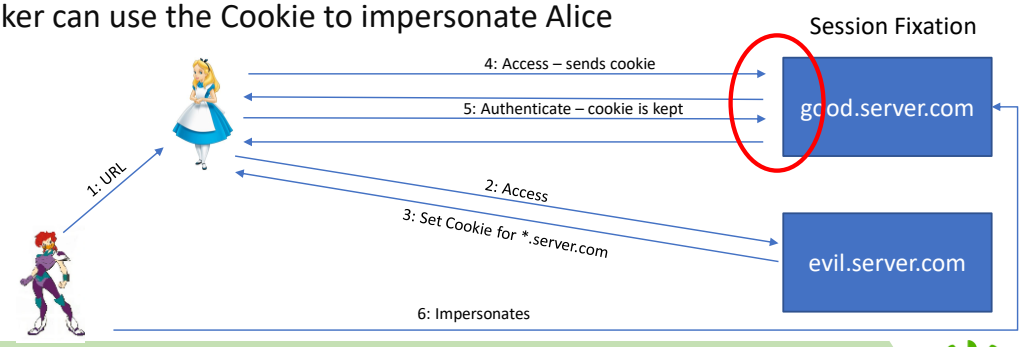
This shows why "strict mode" alone is not enough: the identifier is genuine and was issued by the server. What breaks the attack is the regeneration of the identifier at authentication. It also shows a subtle point: the attacker must keep the session alive until the victim logs in, so idle timeouts on anonymous sessions limit the window. Servers that create sessions for every anonymous request also expose themselves to resource exhaustion, which is why sessions are often created lazily, only when something needs to be stored.

The diagram uses a cast of three actors, with the attacker on the left, the victim and the server. The red circle highlights the point where trust is misplaced: the server associates authentication with an existing, attacker-known identifier.

For testing, repeat the observation of the cookie before and after login, but also try using the identifier obtained in one browser in another browser after the login completes in the first. If the second browser gets access, the cookie is a bearer credential and nothing else is checked, which is expected; the fixation question is only whether you could have known the value before the victim logged in.

Session Fixation - Scenarios

- Cross-Domain Cookie
  - Attacker creates evil.server.com and Alice has account at good.server.com
  - Attacker says: Alice, check this http://evil.server.com and provides a cookie *.server.com
  - If Alice accesses the URL and logs on (The cookie is provided)
  - The attacker can use the Cookie to impersonate Alice



The third scenario exploits cookie scoping rules. Cookies are keyed by name, domain and path, but not by the exact origin that set them: a cookie set with Domain=.server.com by evil.server.com is sent to every host under server.com, including good.server.com. If the attacker controls any subdomain (a forgotten or taken-over subdomain, a user-content site, a vulnerable sibling application, or a hosting service that gives customers subdomains), they can set a cookie that will be sent to the target application. This is known as cookie tossing or cookie injection. The browser does not tell the server which host created a cookie.

Combined with session fixation, it means that the attacker plants a session identifier that they know, wait for the victim to log in to good.server.com, and then use it. Even if the victim's secure cookie is a different one, the attacker may be able to shadow it: when two cookies share a name, the browser sends both, with the more specific path first, and server behaviour in choosing one is not defined, which gives the attacker leverage.

Network attackers can do the same even over HTTPS sites: a plain HTTP response from any subdomain, or an on-path attacker on an HTTP connection, can set cookies that the browser will later send to the HTTPS site, because cookies do not provide integrity with respect to the scheme (a limitation called "weak integrity" in RFC 6265). The mitigation is the __Host- cookie prefix (the browser refuses cookies with the prefix unless they were set by the exact host over HTTPS, with Path=/ and no Domain attribute), HSTS with includeSubDomains, regeneration of the session on login, and good subdomain hygiene (DNS inventory, removal of dangling records).

The numbered steps in the diagram follow the attack order: the victim is lured to the attacker's site, which sets the cookie, then proceeds to the real application.

Protecting sessions

- Cookie attributes
  - Secure (with HSTS): sent only over HTTPS
  - HttpOnly: unreadable by JavaScript, limits theft through XSS
  - SameSite=Lax/Strict: limits cross-site sending (CSRF); __Host- prefix pins to host
- Session lifecycle
  - New SID at login; idle and absolute timeouts
  - Logout destroys the session on the server, not only the cookie
- Token storage in the browser
  - Avoid localStorage for tokens: readable by any script on the page
  - Never place tokens in URLs

This slide gathers the controls that make session credentials harder to steal, abuse and prolong. Cookie attributes first: Secure prevents transmission over plain HTTP, and should be paired with HSTS so that the browser never attempts HTTP; HttpOnly blocks script access and reduces the impact of XSS on cookie theft; SameSite=Lax (or Strict for highly sensitive applications) prevents the browser from attaching the cookie to most cross-site requests, which is the primary defence against CSRF, together with anti-CSRF tokens for state-changing operations; the __Host- prefix prevents overwriting by sibling domains.

The lifecycle is the other half. Generate a new identifier at login and whenever privileges change. Apply an idle timeout (for example 15 to 30 minutes for sensitive applications, longer for low risk ones) and an absolute timeout, so that a stolen session cannot be kept alive forever by periodic requests. Provide a visible logout that destroys the session on the server, deleting the server-side record and, for tokens, adding them to a denylist or relying on short expiry. A logout that only deletes the cookie leaves a valid credential in the hands of whoever copied it. Also invalidate other sessions when the password changes, and show users their active sessions and recent logins.

For browser storage of tokens: localStorage and sessionStorage are readable by any JavaScript running on the origin, including injected scripts and compromised third-party libraries. HttpOnly cookies are not readable, but are sent automatically. Pick based on the threat model, and prefer architectures where the browser holds only an HttpOnly session cookie and the backend holds tokens. Never put tokens in URLs, because they leak through logs, history and the Referer header. Finally, add a Content Security Policy and Trusted Types to reduce the chance that injected script can run at all.

Testing and summary

- Testing authentication
  - OWASP WSTG (Authentication, Session Management) and ASVS V2/V3
  - Burp Suite (Intruder, Sequencer), Hydra, ffuf, jwt_tool
- Key takeaways
  - Never trust the client: the server owns identity and state
  - Slow salted hashes, MFA, rate limiting, no forced rotation
  - Random, protected, regenerated and expiring session identifiers
- References
  - OWASP Top 10, Authentication and Session Management Cheat Sheets
  - NIST SP 800-63B; RFC 6265 (cookies); RFC 7519 (JWT)

To close the lecture, here are the practical elements. For testing, the OWASP Web Security Testing Guide has dedicated chapters on authentication (credential transport, default credentials, lockout, bypass, weak password policy, weak recovery) and session management (cookie attributes, fixation, exposed variables, CSRF, logout, timeout). The ASVS provides verifiable requirements at three levels, which are used both to design and to assess applications. Typical tooling: Burp Suite (Repeater for manual tampering, Intruder for guessing, Sequencer for token randomness), OWASP ZAP, Hydra or ffuf for guessing, jwt_tool for token attacks, and hashcat for offline cracking in lab settings. Only test systems for which you have explicit authorisation; unauthorised guessing against real services is illegal and, in the case of credential stuffing, causes real harm.

The takeaways summarise the design logic: the client is untrusted, so every decision (who the user is, what they can do, whether a token is valid) happens on the server using secrets the attacker cannot guess or forge. Credentials should be made difficult to obtain (MFA or passwordless, TLS, HttpOnly cookies), difficult to guess (entropy, rate limiting, breached password screening) and of little use if stolen (slow salted hashes, short lifetimes, regeneration, revocation).

Suggested practice: set up a deliberately vulnerable application (OWASP Juice Shop, DVWA, WebGoat or the PortSwigger Web Security Academy labs on authentication) and reproduce each attack in this lecture: enumerate users, brute-force a login with rate limits, forge a JWT, and test session fixation. For further reading, use the references listed: the OWASP Cheat Sheet Series (Authentication, Session Management, Password Storage, Forgot Password), NIST SP 800-63B, RFC 6265 and its successor draft for cookies, and RFC 7519 together with RFC 8725 (JWT best current practices).