Authentication Mechanisms and Protocols

SIO

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

João Paulo Barraca

Authentication (Authn)

Proof that an entity has an attribute it claims to have

- Hi, I'm Joe
- Prove it!
- Here is my proof, calculated with Joe's credentials that I've agreed with you
- Proof accepted/not accepted

- Hi, I'm over 18
- Prove it!
- Here is a claim issued by a competent authority, which I can also prove that I'm the owner
- Proof and claim accepted/not accepted

Authentication is the mechanism that turns a claim about an identity into a verified fact. Whenever an entity asserts something about itself - "I am Joe", "I am over 18" - the counterparty can only act on the basis of a proof that it can check. The definition here is deliberately broad: authentication is not limited to people; it is the act of proving that one has a given attribute, and the attribute can be an identity, a role, an age, or any other claim. The two examples on the slide already contain the whole lecture in miniature: the first is a direct proof with a shared secret, the second is a proof backed by a third-party claim, and every protocol that follows is a variation on one of these two patterns.

The first example (a person proving who they are) uses credentials the claimant and the verifier previously agreed on: a secret only the genuine Joe would know. Note the phrasing - the proof is computed with the credentials, which means the verifier never needs to see the secret itself. That distinction is the seed of everything in this lecture: good protocols authenticate without ever transmitting the credential.

The second example is more subtle: proving a fact you cannot derive yourself, like an age limit, requires a claim issued by a competent authority. Now there are two proofs to check - that the claim is genuine, and that you own it. This is the structure behind identity cards, certificates and all attribute-based systems: a trusted third party binds the attribute to you, and you authenticate by showing both the binding and your ownership of it. It is also the origin of the privacy question that runs through the whole lecture: how do you prove ownership of a claim without revealing the secret that proves it, and without leaking the claim itself to anyone who should not see it?

Authn: Proof Types

- Something we know
  - A secret memorized (or written down…) by Joe
- Something we have
  - An object/token solely held by Joe
- Something we are
  - Joe's biometrics

Multi-factor authentication

- Use of several, different proof types
- 2FA = Two Factor Authentication

Risk-based MFA

- Variable MFA
- Higher attack risk, more factors or less risky factors
- Lower attack risk, less or easier factors

The three classic categories - something you know, something you have, something you are - organise nearly every authentication mechanism. A password is a knowledge factor; a token, card or phone is a possession factor; a fingerprint or face is a biometric factor. The categories matter because they fail differently: a password can be guessed or phished, a token can be cloned or stolen, a biometric can be spoofed or leaked. Knowing the category tells you the threat model.

Multi-factor authentication combines different categories, which is much stronger than using the same category twice - two passwords or two cards is still only one factor. 2FA is simply MFA with exactly two factors. The point of the word "multi" is diversity of failure modes, not counting. The diversity requirement is why a smart card plus its PIN counts as two factors (possession and knowledge) while a card alone, or a PIN alone, does not: the factors must fail for independent reasons.

Risk-based (adaptive) MFA adjusts the number and strictness of the factors to the current risk: a login from a known device on a trusted network may pass with a single factor, while an unusual location or a high-value operation can trigger additional, harder factors. This is how large systems balance security with user friction - the friction follows the risk.

When you design or audit a system, start by classifying each credential it accepts into one of the three categories, and then ask what the recovery path is when that factor fails: a forgotten password is reset, a lost token is reissued, and a compromised biometric has no clean answer at all. The category also predicts the attack: phishing targets knowledge factors, theft targets possession factors, and spoofing targets biometric ones. Keep the classic caveat in mind as well: knowledge factors get written down and reused, possession factors get lent out over the phone, and biometric factors cannot be revoked, so no single category is sufficient on its own for accounts that matter.

Authn: Goals

- Authenticate interactors
  - People, services, servers, hosts, networks, etc.
- Enable the enforcement of authorization policies and mechanisms
  - Authorization ≠ authentication
  - Authorization requires authentication
- Facilitate the exploitation of other security-related protocols
  - e.g., key distribution for secure communication

The first and most direct goal is authenticating the parties that take part in an interaction - but "interactors" is broader than human users: hosts, services, servers and even networks authenticate each other. Machine-to-machine authentication is now the majority of authentication traffic, and it is usually the least visible part of security engineering.

The second goal is the link between authentication and authorization: you cannot enforce an authorization policy on an entity whose identity is not established. Authentication is the input to authorization, never a substitute for it - the slide's "authorization is not authentication" is the key sentence. A system that authenticates perfectly but authorizes nothing has solved only half the problem. The dependency is one-way and it should stay that way: an authorization decision can be denied for reasons unrelated to identity, such as time of day or resource state, but it can never be granted before the identity is established.

The third goal is enabling other security protocols, above all key distribution: secure communication protocols need to know who they are exchanging keys with before they can use them. Without authentication, a key-exchange protocol cannot tell whether it is talking to the right peer or to a man in the middle.

A useful reading habit for the rest of the lecture: whenever you meet a new protocol, ask which of these goals it serves and which it assumes. Audit trails, for instance, are only as trustworthy as the authentication that produced the log entries - an attacker who can authenticate as someone else can pollute the very records you would use to investigate them. The key-distribution goal is equally pervasive: session keys are only secret if they were exchanged with the right peer, so every secure-communication protocol in the second half of the lecture is, at its core, an authentication protocol with a key exchange attached.

Authn: Requirements

- Trustworthiness
  - How good is it in proving the identity of an entity?
  - How difficult is it to be deceived?

- Secrecy
  - No disclosure of secret credentials used by legit entities
  - Other performance metrics defined in NIST SP800-63-4

Trustworthiness asks how good a mechanism is at proving identity, and how hard it is to deceive. This is the intuition behind Levels of Assurance: not every mechanism is equally trustworthy, and the level of trust required depends on what is at stake. A LoA statement makes that quality explicit and comparable, so that a requirement like "AAL2" means the same thing to the buyer and the seller. The two sub-questions are complementary and both measurable: the first asks how well the mechanism separates the genuine claimant from an impostor, the second asks how much effort, money and skill an attacker needs to make the system say yes to the wrong person. A mechanism that proves identity perfectly but is trivial to deceive by replay has not passed the second question, and one that resists attackers but constantly rejects its own users is useless in practice.

Secrecy is the other face of the same coin: the mechanism must not leak the secret credentials of legitimate users. A protocol that defeats an attacker but hands the real password to an eavesdropper has failed this requirement even if the attacker is stopped. Both requirements are checked in the design of every protocol later in this lecture. Secrecy has two dimensions: the channel must not reveal the credential to eavesdroppers, and the authenticator must not store it in a form that survives a database compromise.

The mention of further performance metrics points at a trio that every deployment weighs: throughput (how many authentications per second the verifier sustains), latency (how long a user waits) and usability (how many legitimate users are locked out). A mechanism that is trustworthy and secret but so slow or awkward that users bypass it fails in the field rather than in the lab. When you compare mechanisms, score them on all of these axes plus the attack cost - not on cryptographic novelty alone.

Assurance levels by NIST SP 800-63-4

- IAL 1-3 (Identity Assurance): how strictly the identity is verified when the account is created
- AAL 1-3 (Authentication Assurance): how strongly the user must authenticate in each session
- FAL 1-3 (Federation Assurance): how strongly the federation transaction communicates attributes/events

NIST SP 800-63-4 assurance levels	Level 1 Some confidence	Level 2 High confidence	Level 3 Very high confidence
Identity proofing IAL - at enrolment	**Remote or in-person proofing** Evidence validated; claimed attributes checked	**Stronger evidence** Verified against authoritative sources; applicant matched to the evidence	**Supervised proofing** In person (or supervised remote) with a trained agent; biometric collected
Authentication AAL - every session	**Single factor allowed** *e.g.,* password	**Two distinct factors** A phishing-resistant option must be offered *e.g.,* password + TOTP; synced passkey	**Phishing-resistant hardware key** Private key cannot be exported *e.g.,* FIDO2 security key + PIN; smart card
Federation FAL - IdP to RP	**Signed assertion** IdP signs it, audience-restricted to the RP *e.g.,* OIDC ID token, SAML	**+ injection protection** RP registered with the IdP in advance; assertion bound to the request	**+ holder-of-key** User also proves to the RP possession of a key bound to the assertion
Example service	*Forum or newsletter account*	*Online banking, tax or e-government portal*	*Privileged admin access, health records*
IAL, AAL and FAL are chosen independently	Higher assurance - more cost and friction for the user		

Special Publication 800-63 is the reference framework that turns "strong authentication" into measurable levels. It splits the problem in two: how the identity was verified when the account was created (Identity Assurance, IAL), and how the user is authenticated in each subsequent session (Authentication Assurance, AAL). The two are independent - you can have a well-verified identity with weak session authentication, and vice versa. The current revision is SP 800-63-4 (2025); besides IAL and AAL it also defines FAL (federation assurance level), which grades the strength of federated assertions.

IAL 1 to 3 covers identity proofing: from self-asserted identity (IAL1, "say who you are") to in-person verification of government-issued documents (IAL3). The higher the IAL, the harder it is for an attacker to create a fraudulent account in the first place, which matters because many attacks start with account creation rather than session takeover.

AAL 1 to 3 covers per-session authentication: AAL1 is a single factor such as a password; AAL2 requires two factors, for example a password plus an OTP or a biometric; AAL3 requires a hardware-based, non-exportable, phishing-resistant authenticator, for example a device-bound FIDO2 security key; synced passkeys reach AAL2, not AAL3.

The practical value of the framework is comparability: a procurement document or a compliance requirement can say "AAL2" and both sides know exactly what is being demanded. It also gives a ladder for improvement - systems can state their current level and plan the next step instead of debating adjectives. It also gives a common language for incident response: when a breach happens, the question is not how strong the password policy was but which level the session authentication was at when the account was taken over.

Authn: Requirements

- Robustness
  - Prevent attacks to the protocol data exchanges
  - Prevent on-line DoS attack scenarios
  - Prevent off-line dictionary attacks
- Simplicity
  - It should be as simple as possible to prevent entities from choosing dangerous shortcuts
- Deal with vulnerabilities introduced by people
  - They have a natural tendency to facilitate or to take shortcuts
  - Deal with phishing!

Robustness targets three attack classes. Attacks on the protocol's data exchanges (man-in-the-middle, replay), online denial-of-service (flooding the authenticator, or triggering account lockouts), and offline dictionary attacks, where a stolen credential database is attacked at leisure, without any detection by the system, by an automated process that guesses many candidate secrets against the stored values. The offline dictionary case deserves special attention because it is the most common real-world incident: the attacker never touches the live system, they sit in front of a stolen hash database at their own pace, and the only defence is making each guess expensive.

Simplicity is a security requirement, not a convenience: every complication in a protocol is an opportunity for an implementation bug or for users to take shortcuts. This is why modern authentication keeps pushing complexity into well-tested hardware (tokens, secure elements, platform authenticators) rather than into clever protocol tricks that humans must understand to use safely.

The human factor is the most exploited of all: people reuse passwords, write them down, and click on convincing phishing pages. A robust protocol must assume that the human side will do something careless - phishing resistance is the clearest example of designing around that reality instead of against it.

A practical way to apply these requirements is to walk through a protocol asking three questions: can an attacker modify or replay the exchanges, can they lock users out by flooding the authenticator or triggering lockouts, and can they attack the stored credentials offline at their own pace? The third question is the one most often forgotten at design time, because it involves no interaction with the system at all - which is also why its damage is discovered latest, when the database is already stolen. Simplicity, meanwhile, is the requirement that protects all the others: every optional branch and every configuration knob is a place where two implementers can disagree, and a disagreement in authentication usually ends with the weaker option being enabled.

Authn: Entities and deployment model

Entities

- People
- Hosts
- Networks
- Services/servers

Deployment model

- Over time
  - Only when the interaction starts
  - Continuously along the interaction
- Directionality
  - Unidirectional
  - Bidirectional (mutual)

Authentication applies to many kinds of entities. People are the obvious case; hosts, services/servers and networks also need to prove who they are, which is why protocols like TLS and SSH authenticate both sides, and why infrastructure identity management (identities for servers, devices and services) is a growing field of its own. The growth of infrastructure identity is driven by the machine economy: a modern organization has many more machine identities than human ones, and the authentication machinery for devices must handle scale, rotation and revocation without any human in the loop.

The deployment model has two dimensions. Timing: authentication can happen once at the start of an interaction, or continuously throughout it - re-authentication, step-up authentication for sensitive operations, and session monitoring are all forms of the continuous model. Direction: unidirectional (only the server proves itself, as in an ordinary HTTPS session) or bidirectional/mutual (both sides prove themselves, as with client-certificate TLS or SSH).

The choice of model should follow the risk: a web session that only reads public content needs little, while a banking operation or a root-level SSH session benefits from mutual, continuous authentication. Most incidents happen because a high-value operation reuses a low-assurance session.

When you read a requirement such as "only authenticated hosts may talk to the payment service", unpack it against this slide: which entities, at what moments, and in which direction? The answers change the design completely - a unidirectional check at connection start is a different product from continuous mutual authentication, and it is sold, deployed and attacked differently. The mutual case also has a cost asymmetry worth remembering: it requires both sides to hold credentials, which doubles the provisioning and revocation work, and that operational cost - not cryptography - is what most often decides how strong the deployed authentication actually is.

Authn interactions: Basic approaches

- Direct approach
  1. Provide credentials
  2. Wait for verdict

  – Advantage: no computations by the presenter
  – Disadvantage: credentials can be exposed to malicious validators

- Challenge-response approach
  1. Get challenge
  2. Provide a response computed from the challenge and the credentials
  3. Wait for verdict

  – Advantage: credentials are not exposed to malicious validators
  – Disadvantage: requires computations by the presenter

Every authentication protocol belongs to one of two families. In the direct approach the presenter simply sends the credential and waits for a verdict; in the challenge-response approach the presenter first receives a random challenge and then sends a value computed from the challenge and the credential. The whole lecture can be read as a tour of these two families and their descendants. The two families are not historical stages but living siblings: direct-approach mechanisms still appear inside trusted boundaries, and challenge-response dominates everywhere the channel cannot be assumed safe.

The trade-off is exactly the one on the slide. The direct approach is computationally trivial for the user, but it exposes the credential to any entity that can see the transmission - so it is only safe over an already-protected channel. Challenge-response never transmits the credential itself: an eavesdropper sees a one-off challenge and a one-off response, neither of which is reusable.

The price of challenge-response is computation by the presenter, which is why it usually needs a device or an application to do the work. Most modern protocols - TLS, SSH, cellular networks, FIDO2 - are challenge-response for precisely this reason, and the remaining direct-approach protocols survive where the channel is already trusted. The split is also visible in the user experience: a direct approach is one keystroke, a challenge-response is a device interaction, and the ecosystem's steady migration from the first to the second is the migration from passwords to tokens.

Authn of subjects: Direct approach w/ known password

- A password is checked against a value previously stored
  - For a claimed identity (username)

- Personal stored value:
  - Transformed by a unidirectional function
  - Windows: NT hash (MD4, no salt)
  - UNIX: DES hash + salt
  - Linux: yescrypt or SHA-512 + salt
    - hash is configurable (formerly MD5)

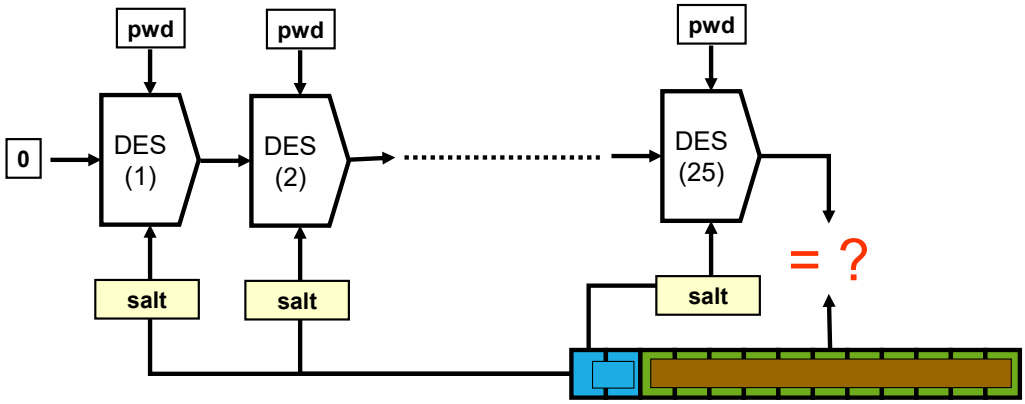
Optimal scenario
- Complex, slow password transformations
- Argon2id, scrypt, PBKDF2 with high cost

The simplest authentication mechanism in existence: the user types a password, and the system compares it against a value stored for the claimed identity. The crucial detail is that the system never stores the password itself - it stores the output of a one-way transformation, so that even a stolen database does not directly reveal the passwords of the users. The claimed identity, the username, is the lookup key, and the stored value is the only thing the system ever compares against: the password itself is a transient that should exist only in the user's head and, briefly, in memory.

The stored value should be the result of a deliberately slow transformation. A fast hash (a plain SHA-256, for instance) is a liability here: an attacker who steals the database can try millions of candidates per second. Modern key-derivation functions - PBKDF2, scrypt, Argon2 - exist precisely to make each guess expensive; the slide's "optimal scenario" box points at exactly this. This is the direct connection to the earlier lecture on digests and key derivation: a password hash is a key-derivation function with a work factor, and the parameters chosen there are exactly the parameters that determine how long an offline attack takes.

Operating systems have historically used different schemes (Windows stores the NT hash, an unsalted MD4 of the password; classic UNIX used DES with salt, and Linux used MD5 with salt, now yescrypt or SHA-512-crypt by default, with the algorithm configurable per user). The exact algorithm matters less than the principle behind it: slow, salted, per-user, and one-way. The per-user configurability is what allows a migration: new users get the strong algorithm, old hashes are re-derived on the next successful login, and the weak scheme fades out without a mass password reset.

Authn of subjects: Direct approach w/ known password



$$\text{DES hash} = DES_{pwd}^{25}(0)$$

$$DES_k^n(x) = DES_k(DES_k^{n-1}(x))$$

Salt (12 bits) swaps 12 bit pairs of the DES E expansion

The figure shows how the classic UNIX DES password hash works: the first 8 characters of the password, 7 bits each, form a 56-bit DES key that is used in all 25 iterations, repeatedly encrypting an all-zero block; the 12-bit salt perturbs DES's E expansion (swapping bit pairs), so standard DES hardware cannot be used directly. The output is 64 bits, stored as 11 characters preceded by the 2 salt characters. It looks baroque because it was engineered in the 1970s to be slow enough to resist brute force on the hardware of the time - the slowness is the feature. Reading the formula on the slide: each of the 25 encryptions uses the same password-derived key and takes the previous output as its input, so the whole password influences the result, and the final output is the hash stored in the password file.

The 25 iterations were the "slowness" mechanism. Today that slowness is negligible: commodity hardware tests billions of candidates per second, and the 12-bit salt plus the 8-character limit make precomputation attractive. The scheme is legacy, and it survives in this deck as the historical baseline. The 8-character limit is the most painful legacy detail: it silently caps the entropy of the password, so even a strong modern password would have been truncated to its first eight characters under this scheme.

It remains a good teaching example because it contains all the ingredients of a password hash - a one-way transform, a salt, and iterations - in their minimal, pre-modern form. Everything that follows is the same recipe with bigger parameters and more honest engineering.

Password storage: modern key-derivation functions

- DES-based crypt is legacy
  - Fast, 12-bit salt, passwords truncated to 8 characters
- Use a slow, salted KDF: Argon2id, scrypt, PBKDF2
  - A unique random salt per user defeats precomputed (rainbow-table) attacks
  - Memory-hard functions (Argon2, scrypt) resist GPU/ASIC attacks
- Tune the cost to the highest the server tolerates
  - e.g., OWASP baseline: Argon2id, 19 MiB, 2 iterations, 1 thread
  - Store algorithm + parameters + salt + hash, never the password

The DES scheme just shown is the historical starting point; modern systems replace it with dedicated key-derivation functions: PBKDF2 (RFC 8018), scrypt, and the Argon2 family, of which Argon2id is currently the recommended default in most password-storage guidelines. Argon2id is a hybrid of Argon2i (data-independent memory access, side-channel resistant) and Argon2d (data-dependent memory access, GPU-resistant).

All of them take the password, a unique per-user salt, and a work factor (iteration count, memory cost, or both) and stretch the computation. The salt kills precomputed rainbow-table attacks - each user's hash chain must be computed from scratch - and the work factor makes online and offline guessing economically painful. Scrypt and Argon2 add a memory component, which defeats GPU/ASIC parallelism that pure iteration counts do not.

Deployment guidance: pick a KDF, a random 128-bit (or longer) salt per user, and tune the cost parameters to the highest cost the server tolerates (the OWASP baseline for Argon2id is 19 MiB of memory, 2 iterations and parallelism 1), which typically means well under one second per check. Store the algorithm name, parameters, salt and hash together - never the password itself, and never a fixed parameter set that cannot be raised when hardware improves. The cost setting is a policy decision, not a physical constant: it should be re-tuned every few years as hardware gets faster, and it should be measured on the actual authentication server, not on a developer's laptop.

Authn of subjects: Direct approach w/ known password

- Advantage
  - Simplicity!

- Problems
  - Usage of weak passwords
    - Enable dictionary attacks
  - Transmission of passwords along insecure communication channels
    - Eavesdroppers can easily learn the password
    - e.g., telnet, rlogin, FTP, PAP



Rank	Password	Count of password uses
1	123456	21,627,656
2	admin	21,030,012
3	12345678	8,274,408
4	123456789	5,673,712
5	12345	3,950,777
6	password	3,545,119
7	Aa123456	2,520,728
8	1234567890	1,418,939
9	Pass@123	1,210,039
10	admin123	1,087,247

https://nordpass.com/most-common-passwords-list/

The password's great virtue is simplicity, and its great vice is the same: it is so simple that humans choose terribly. The two structural problems on the slide - weak passwords enabling dictionary attacks, and transmission over insecure channels - are both consequences of using a human-memorized secret as a cryptographic key.

The Top Ten list shown (SplashData, 2017 - the last edition published) is a historical snapshot rather than a current ranking, but its content has barely changed in a decade: sequential digits, the literal word "password", keyboard patterns, sports teams, "iloveyou". The same entries topped the charts in 2007, 2012 and 2017, which says something about human behaviour rather than about security technology. The list also has a methodological point: it is a snapshot of breached or leaked credential data, not a survey of what people choose in general, so it measures the population of attacked accounts rather than the population of all users.

The lesson is not "use a better password" - it is that memorized secrets cannot be expected to carry the full weight of authentication on their own. That is the motivation for everything that follows in this lecture: biometrics, one-time passwords, tokens, passkeys, and protocols that keep the secret out of the channel entirely.

The transmission problem deserves a note of its own: the password itself can travel safely if the channel is already encrypted, which is why a password form over HTTPS is not broken by eavesdropping - the residual risks are the endpoint, the user being phished, and reuse across sites. What passwords cannot survive is the combination of human choice and offline attack: the verifier's stored value becomes a guessing target the moment it leaks, and the guessing is limited only by the work factor and the users' imagination. That asymmetry - trivial to use, expensive to store safely, impossible to un-learn - is what keeps passwords in place for low-value accounts and pushes high-value systems toward the factors that follow.

Password policies: NIST SP 800-63B

- Favor length over complexity
  - At least 15 characters (8 if part of MFA); accept 64+ and any character
  - No composition rules (e.g., mandatory digits or symbols)
- Do not force periodic password changes
  - Require a change only on evidence of compromise
- Block known-bad passwords and limit guessing
  - Check against breached and common-password lists
  - Rate-limit failed attempts; allow password managers (paste)

The Top Ten list shows why password rules matter, but the classic rules (mandatory symbols, a change every 90 days) made things worse: users respond with predictable patterns such as Password1! followed by Password2!. The SP 800-63B guidelines, in their 2025 revision (800-63-4), now forbid both composition rules and forced periodic changes.

Length is what actually increases the search space, so the minimum is 15 characters when the password is the only factor, and 8 when it is part of a multi-factor authenticator. Verifiers must accept long passwords (at least 64 characters) and any printable character, including spaces, so that passphrases and password managers work.

The effective controls are on the verifier side: reject passwords found in breach corpora and common-password lists, rate-limit failed attempts to defeat online guessing, and allow pasting so that users can use password managers, which is what makes long, unique, per-site passwords practical.

The change from the classic advice is worth pausing on, because most people were trained under the old rules. Forced periodic changes measurably weakened security: users pick Password23! after Password22!, and the increment is predictable, so the requirement manufactured weak passwords at scale. The guidelines invert the logic - the verifier watches for evidence of compromise and acts then, which puts the effort where the actual risk is. Rate-limiting is the online counterpart of the slow hash: the first defends against guessing from outside, the second against guessing from a stolen database, and a deployment needs both.

For practice, the checklist reduces to: allow long passphrases and password managers, block the few thousand most common and breached passwords, and never send a change request that the user did not initiate - phishing pages mimic change forms precisely because users are trained to type passwords into them.

Authn of people: Direct approach with biometrics

- People get authenticated using body measures
  - Biometric samples
  - Fingerprint, iris, face geometry, voice timbre, handwriting, vein matching, etc.
- Measures are compared with personal records
  - Biometric references (or template)
  - Registered in the system with a previous enrolment procedure
- Identification vs authentication
  - Identification: 1-to-many check for a match
  - Authentication: 1-to-1 check for a match

Biometric authentication replaces "something you know" with "something you are": the system measures a physical or behavioural characteristic - fingerprint, iris, face geometry, voice, handwriting, even vein patterns - and compares the fresh measurement against a template stored at enrolment. The comparison is a similarity decision, not an equality check.

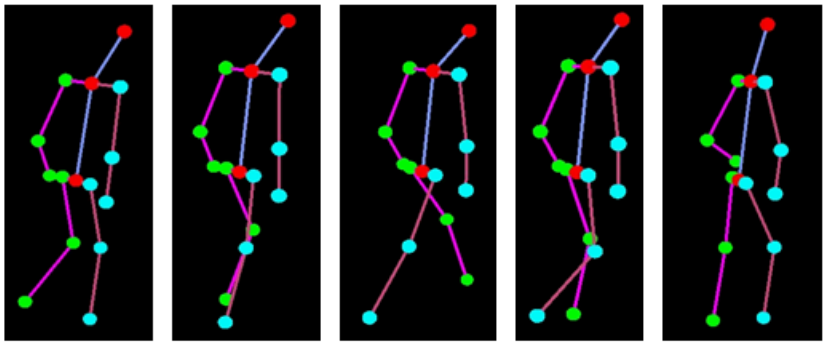
Two details matter. First, the template is not the raw sample: no system should store the actual fingerprint image; enrolment processes the sample into a compact, non-reversible template. Second, a verification threshold decides the trade-off between rejecting legitimate users and accepting impostors - every biometric system has a false-accept and a false-reject rate, and moving the threshold trades one for the other.

The slide also draws an important distinction. Identification is one-to-many: "who is this?" - matching one sample against a whole database. Authentication is one-to-one: "are you the person whose template this is?" - matching against a single record. Identification is computationally heavier and raises much sharper privacy questions, which is why most authentication systems do only one-to-one matching. The privacy point matters beyond engineering: a one-to-many database of templates is a surveillance asset, which is why the template store itself must be treated as sensitive data, often with its own encryption and access control.

The threshold decision is the practical heart of every biometric deployment, and it is context-dependent: a phone unlock tolerates more false rejects (you simply retry) but must never accept an impostor, whereas high-throughput gates tune the other way. The rates are also unequal in cost: a false reject annoys one legitimate user, a false accept silently grants an attacker whatever the biometric guards. Enrolment quality matters as much as the sensor - a template captured once, badly, degrades every later verification - and the template store must be protected like a password database, because a leaked template can be replayed to any matching sensor.







These images show examples of the biometric modalities introduced earlier - different body measures that can serve as credentials. The common thread is that each modality is measured by a dedicated sensor: optical scanners for fingerprints and irises, cameras for faces, microphones for voice. The quality of the capture device is a first-order security property, because every error in capture shows up as an error in the decision.

Notice what none of these images is: a secret. Biometric data is public in a sense - you cannot stop a camera from capturing your face - so the security comes from the difficulty of reproducing the sample with enough fidelity to pass the verification threshold, and from protecting the stored template. A password you can keep dark; a face you cannot.

Take the gallery as a survey of sensor engineering rather than a menu of equal options. Fingerprints and iris are the most established, with decades of accuracy data; face recognition is the most convenient and the most attacked, because cameras are everywhere and photographs of you are everywhere; voice and handwriting are behavioural as much as physical, so they vary with the user's state - a cold, fatigue, a different pen. Vein patterns are the quiet overachiever: internal to the body, hard to capture without cooperation, and stable over a lifetime, which is why they appear in payment systems.

For each modality the same two questions apply: how hard is it to counterfeit the sample (a printed fingerprint fools a weak reader, a photograph fools a naive face system), and what does the stored template leak if stolen? A reader that only measures, and a verifier that only compares templates, keeps the security in the match quality - which is exactly why spoofing and template theft are the two attack surfaces to keep in mind for every image here.

Authn of people: Direct approach with biometrics

- Advantages
  - People do not need to use memory, or carry something
  - Just be themselves
- People cannot choose weak passwords
  - In fact, they don't choose anything
- Authentication credentials cannot be transferred to others
  - One cannot delegate one's own authentication

The appeal of biometrics is that the credential is inescapable: the user does not need to remember anything or carry anything, and there is nothing to lose, forget or share. "Just be yourself" is the strongest possible statement of user experience, and it is why biometrics won the consumer market - phone unlock, laptops, payment terminals. The just-be-yourself property is also a supply-chain property: the credential cannot be shared, which sounds like a strength, but it means that every device that accepts the biometric becomes a gatekeeper with permanent access, and the lifecycle of those devices, theft or resale or loss, is now a security issue that passwords never had. It is also why consumer biometrics are usually paired with a fallback factor: the phone still asks for a PIN if the fingerprint reader fails, and that fallback is an admission that the biometric alone is not the whole story.

Two points deserve emphasis. "People cannot choose weak passwords" is literally true - there is no choice at all, which removes the entire class of poor-password attacks that made the Top Ten lists. And the credential cannot be delegated: you cannot lend your fingerprint to a colleague the way you can lend a password or a card, so biometric authentication binds the person to the session in a way no other factor does. The non-transferability is the mirror image: a password can be revoked and reissued, a card can be reprogrammed, but the biometric is fixed for the lifetime of the body, so the entire security model has to assume that the credential itself cannot change, which in practice means the systems around it, the templates, the devices, the thresholds, must be the ones that are managed and updated.

Authn of people: Direct approach with biometrics

- Problems
  - Biometric methods are still imperfect
    - In many cases it can be fooled with ease (Face Recognition, Fingerprint)
  - People cannot change credentials
    - If the credentials or templates are stolen
  - Credentials cannot be transferred between individuals
    - If it is required in extraordinary scenarios
  - Can pose risks to individuals
    - Physical integrity can be compromised by an attacker in order to acquire biometric data
  - It is not easy to implement in remote systems
    - It is mandatory to have secure and trusted biometric acquisition devices
  - Biometrics can reveal other personal secrets
    - Medical conditions

The problems are structural, not engineering details. First, biometric templates cannot be revoked and reissued: if a password is compromised you change it; if a fingerprint template is stolen there is no "reprint" of the finger. Systems mitigate this by storing derived, non-reversible templates and binding them to device-specific secrets, but the asymmetry with passwords remains and should always be stated.

Second, the capture device must be trusted - a remote system is only as secure as the sensor that feeds it - and spoofing attacks are real: printed fingerprints, photos or deepfakes of faces. Liveness detection (pulse, gaze, challenge-response behaviour) is part of any serious deployment, and it adds cost and friction.

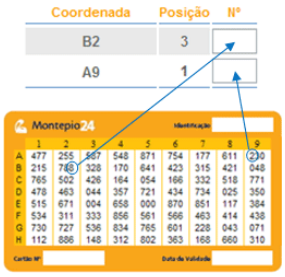
Third, biometrics are not neutral: they can leak other personal information (some iris and face analyses can reveal medical conditions), and in hostile scenarios an attacker can coerce or physically harm the person to harvest the biometric data - a threat class that has no equivalent for passwords or tokens. The coercion case is the one that has no engineering fix: an attacker who can hold the person can get the fingerprint or the face, which is why high-security designs keep a knowledge factor as a fallback that cannot be extracted by force.

The remote-deployment problem in the list has the most direct engineering consequence: if the sensor cannot be trusted, the whole decision is worthless, so serious designs move the matching into a secure element next to the sensor or bind the sensor to the verifier cryptographically - the same reasoning that later reappears in device-bound passkeys. The "cannot change credentials" bullet pairs with the leak problem in an unpleasant way: a stolen template is permanently valid wherever it is accepted, so the mitigations are architectural (never store raw samples, never let one template work everywhere) rather than operational. When you weigh biometrics for a system, price in these irreversibilities first.

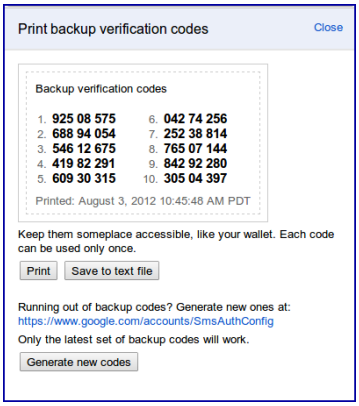
Authn of subjects: Direct approach w/ one-time passwords

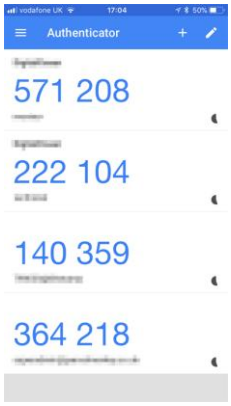
- One-Time Passwords = Secrets that can be used only once
  - Pre-distributed directly, or the result of a generator function

- Example: Bank codes, Google Backup Codes, Authenticators



https://net24.bancomontepio.pt/Net24-Web/func/general/cmcPHelp.jsp





One-time passwords are secrets designed to be used exactly once, which limits the damage of any single exposure to a single transaction. They come in two forms: pre-distributed lists (bank codes, the Google backup codes - a printed sheet of one-time passwords) or values generated on the fly by a shared function, which is what tokens and authenticator apps implement. The two families differ in their threat model: the pre-distributed list is only as safe as the paper it is printed on, and the user's discipline in using each code once, while the generated family is only as safe as the generator's secret and its synchronization.

The example on the slide (the Montepio "cartao-matriz" page) is a classic pre-distributed list: the customer receives a card of one-time codes and types one per operation. It is a possession factor - the card holds the secret - which makes it a genuine second factor when combined with a knowledge factor.

The value of OTPs is that replay is impossible: even a perfectly captured code is worthless the next time, and the attacker's window of usefulness is at most one transaction. This property is what makes one-time secrets usable even over channels that are not fully trusted. The bounded-exposure property is what makes one-time secrets composable: they can sit next to long-lived secrets in the same protocol and not become a weakness, which is why they appear in so many of the mechanisms that follow.

A useful way to hold the family together: pre-distributed OTPs are a possession factor you must keep safe and consume in order; generated OTPs are a shared-secret design where the generation rule is the credential. The bank-code card works anywhere because the codes are independent; the generator works indefinitely because the rule produces new values - but both fail if the holder loses the list or the device, which is why backup codes exist for the backup codes. Note also what OTPs do not fix: they bind nothing to the session, so whoever tricks the user into revealing the current code in real time inherits the session, a limit that motivates the phishing-resistant designs later in the lecture.

Authn of subjects: Direct approach w/ one-time passwords

- Advantages
  - Eavesdropping is harmless: a used OTP cannot be replayed (but OTPs can still be phished and relayed in real time)
  - Can be chosen by the authenticator, which may enforce a given complexity
  - Can depend on a shared password

- Problems
  - Interacting entities need to know which password to use on each occasion
    - Implies some form of synchronization (e.g., index, coordinates)
  - Individuals may require additional resources to store/generate the passwords
    - Sheet of paper, application, additional device, etc.

The advantages on the slide are subtle and important. Because a one-time value is useless after one use, it can travel over an unencrypted channel without revealing a reusable secret - the exposure is bounded by design. And because the authenticator chooses or generates the values, it can enforce a complexity that humans would never pick for themselves.

The main problem is synchronization: both sides must agree on which password is valid right now. That agreement is usually a counter (event-based) or a clock (time-based), and drift between the two sides breaks authentication - which is why real systems accept a small window of adjacent values rather than a single exact one. The synchronization window is where the security meets the user: a window of one on either side means a code is valid for roughly two periods, which is long enough for clock drift and short enough that the replay risk stays inside one human transaction.

The secondary problem is user burden: each person needs a place to keep the list or a device that generates the values - a paper sheet, an application, a hardware token. This friction is what pushed the ecosystem from OTP lists toward push-based notifications and finally FIDO2 methods, where the user only confirms. The device question is also a supply-chain question: a hardware token that generates the values must be provisioned with its secret over a channel that is at least as trustworthy as the one the OTP is protecting.

The synchronization problem explains a lot of product design you see daily: the accept-window on authenticator apps, the "next code" button, and the recovery flow when a phone is replaced. Each is an engineering answer to the same question - how do two parties that rarely talk agree on where they are in the sequence? For your own designs, the rule of thumb is that any state held by two parties drifts, so plan the resynchronization before the first desynchronization happens, and treat the state (counter, clock, index) as security-sensitive data: an attacker who can reset it can force replay windows open.

YubiKey



- Personal Authentication Device
  - USB, Bluetooth and/or NFC

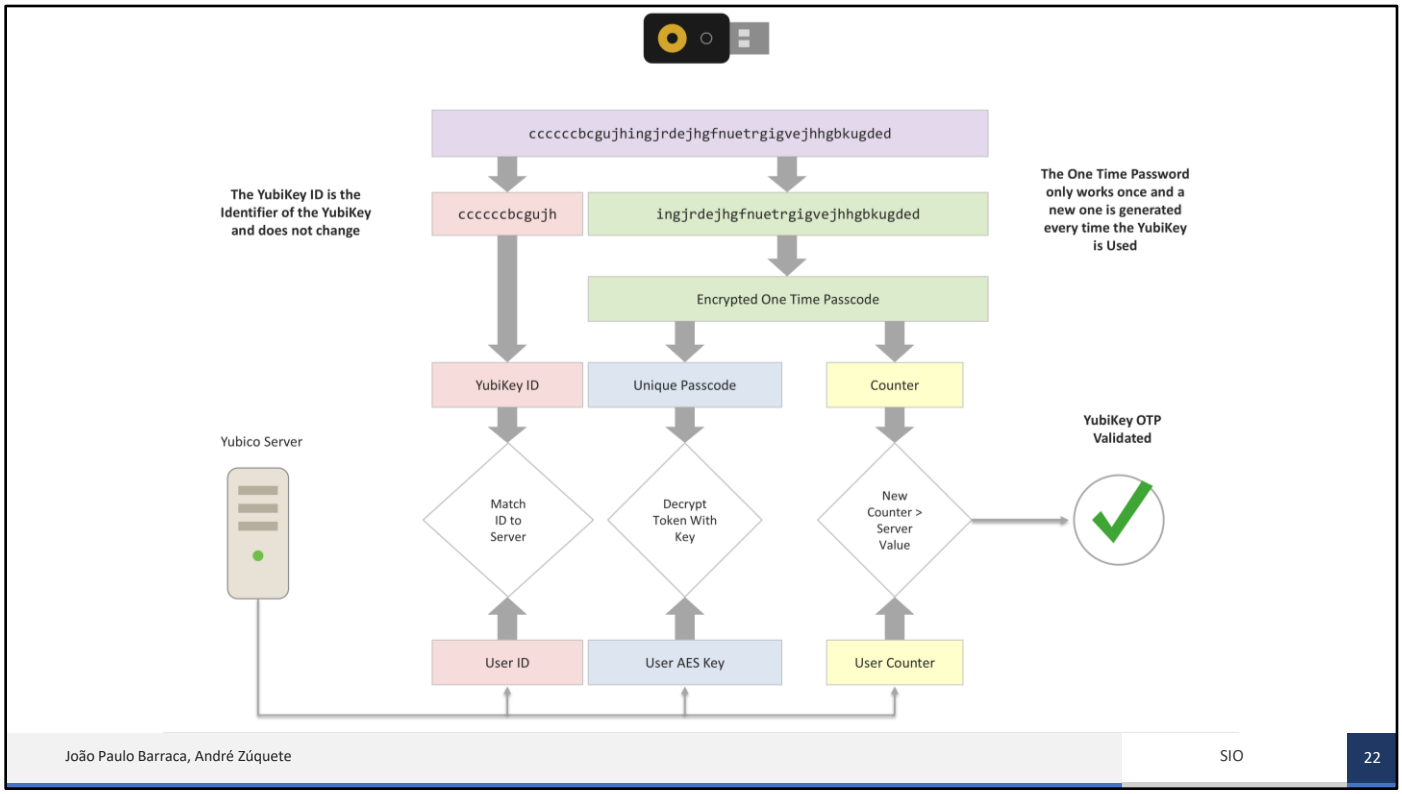
- Touch generates a 44-character OTP
  - Emulates a USB keyboard (besides an own API)
  - Supports HOTP (events) and TOTP (time-based, via a companion app)
  - Challenge-response (HMAC-SHA1) can require a button touch
  - Also FIDO U2F/FIDO2, PIV smart card and OpenPGP

The Yubikey is the canonical example of a personal hardware authentication device: a small USB token (newer versions also speak Bluetooth and NFC) that acts as a possession factor. When activated for a service it generates a long per-service credential; the 44-character value on the slide is a token-derived code of that kind. The 44-character value is a deliberate design: it is long enough to be unguessable, short enough to type, and unique per service, so a leak on one site does not help on any other.

Operationally it emulates a USB keyboard - it simply types the code into whatever field is focused - which makes it work with any system that accepts one-time passwords, in addition to its own API and its native FIDO2 mode. It supports HOTP (event-based) and TOTP (time-based) codes, several algorithms including AES-256, and a physical button that must be touched to produce a response.

The button is a nice security detail: the challenge-response exchange requires deliberate user consent, so a token left plugged into a PC cannot be abused silently by malware that does not have the user's finger. Confirmation and possession are two separate checks, even though they live in the same device. The same button pattern appears in every FIDO2 token: the cryptographic operation and the user confirmation are one gesture, and that gesture is what separates a token from malware that can merely read it.

The device is worth studying as a design pattern, not only as a product: put the secret in hardware that cannot export it, require a physical gesture for every use, and speak as many standard protocols as possible so the same token serves second factors, password-less logins and even encryption at once. The keyboard-emulation trick is the compatibility story - the token works with any login form that accepts a long code, no driver and no network - while FIDO2 is the security story, because it binds responses to the website's origin. The lesson generalizes: a personal authentication device succeeds when it lowers the user's effort while raising the attacker's, and both happen here.



The image shows a personal authentication device of the kind described earlier - a small token that holds the secret and performs the computation locally. Devices like this move the entire "something you have" factor into hardware that the user carries, and into a chip that is substantially harder to extract than a secret stored in an application.

The practical effect of such devices is that the secret never appears on the screen or in the clipboard in a reusable form: each code they emit is one-time, and in FIDO2 mode they never expose a static secret at all. The token becomes a gate with a fingerprint, and the user becomes the confirmation step.

The photograph shows a hardware token of the class just described: a personal authentication device whose entire value is that it holds a secret the user cannot read out, and computes with it locally. Physically it is a small dongle with a touch area; internally it is a secure element with per-protocol secrets provisioned at manufacture over an authenticated channel.

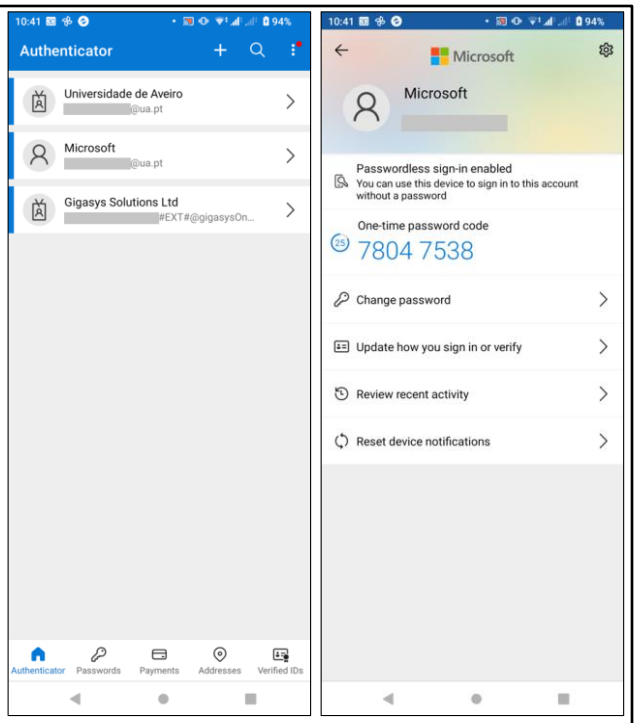
Keep in mind what such a device changes in the threat model. Phishing stops working in its FIDO2 modes, because the response is bound to the genuine origin; replay stops working because challenges are fresh; database theft stops working because the verifier stores only public keys. What remains is the physical layer: the token can be lost (hence every service must offer a second enrolled token or a recovery path), stolen (hence the touch requirement, which a thief at a distance cannot satisfy), or cloned only through attacks on the chip itself, which are the expensive kind. That residual profile - everything cryptographic solved, everything physical remaining - is the honest summary of hardware tokens.

HOTP / TOTP

- Shared secret between authenticator and user

- HOTP (HMAC-based One-Time Password)
  - Each Interaction generates a new code
    - Counter-based HMAC (RFC 4226)
  - Result is converted to human-readable text
  - Counter's synchronization is fundamental

- TOTP (Time-based One-Time Password)
  - HOTP with 30s time steps (RFC 6238)
  - Time synchronization is fundamental



HOTP (RFC 4226) is the standard counter-based one-time password: a shared secret and a counter feed an HMAC, and the result is truncated into a short human-readable code - typically six digits. The counter must advance in lockstep on both sides; if the token and the server drift apart, authentication fails until they are resynchronized, which is the "desynchronization" problem mentioned on the slide. The truncation step is a usability decision with a security cost: six digits means a one-in-a-million space per step, which is acceptable because the code also has to be current, but it is the number that makes shoulder-surfing a realistic threat.

TOTP (RFC 6238) replaces the counter with time: the "counter" is the current Unix time divided into fixed steps, normally 30 seconds. This removes the need to track usage - any party that knows the current time can compute the current code - at the price of requiring clock synchronization between the device and the server. The 30-second step is a standard value, not a physical limit: some deployments use 60 seconds to reduce synchronization pressure, at the price of a wider replay window.

Both are the basis of the authenticator apps most people already use. The 30-second step plus a small accepted window (typically one step in either direction) is what makes TOTP tolerant of phones whose clocks drift a few seconds - and it is also what bounds the replay window of a captured code to about a minute.

One detail ties the two together: in both schemes the shared secret is static and the freshness comes entirely from the counter or the clock. That is why the provisioning step - scanning the secret into the app as a QR code - is the most security-critical moment in the whole scheme, and why storing it in the cloud backup of an authenticator app quietly changes the threat model. When comparing them for a deployment: the counter version wins on systems without reliable clocks and on air-gapped equipment; the time version wins on user experience, because the clock does the work and the user never resynchronizes anything.

MFA attacks and phishing resistance

- Real-time phishing (adversary-in-the-middle proxies)
  - Relays password + OTP and steals the resulting session cookie
- MFA fatigue (push bombing) and SIM swapping
  - Use number matching for push; avoid SMS OTP for valuable accounts
- Phishing-resistant MFA binds the proof to the origin or channel
  - FIDO2/WebAuthn (passkeys), smartcards with TLS client authentication

One-time passwords defeat replay, but not phishing. An adversary-in-the-middle phishing kit sits between the user and the real site: the user types the password and the current OTP into the fake page, the proxy forwards both immediately, and the attacker keeps the authenticated session cookie. The OTP was used only once - by the attacker.

Push-based MFA introduced a different weakness: an attacker who knows the password sends approval requests until a tired user taps accept. Number matching, where the user must type a number shown on the login screen, stops blind approvals. SMS codes add the risk of SIM swapping, where the attacker convinces the operator to move the victim's number to a new SIM.

The structural fix is phishing-resistant MFA: the authenticator signs a challenge bound to the origin of the site (FIDO2/WebAuthn) or to the TLS channel (smartcard client authentication), so a response produced for a fake site is useless on the real one. This is why the strongest assurance level in the 800-63 framework demands a phishing-resistant authenticator.

The adversary-in-the-middle case is worth tracing end to end: the user browses a convincing copy of the login page, the kit proxies every field to the real site in real time, the user completes the second factor, and the attacker's automated client lands on the genuine site with a fresh session - all within seconds, well inside the code's validity window. No amount of factor count helps, because the user themselves performed the authentication on the attacker's behalf. That is why phishing resistance is a property of the credential binding, not of user vigilance.

For your own accounts, the practical hierarchy is: device-bound passkeys and smartcards first (origin-bound, phishing-resistant), authenticator apps second, push approvals with number matching third, and SMS last - not because SMS codes are weak cryptography, but because the phone-number account is the weakest link in that chain. Standards frameworks echo the same ordering, which is why the top assurance level demands this phishing-resistant class of authenticator.

Challenge-Response approach

1. The authenticator provides a challenge
   - A nonce (number used only once)
   - Usually random
   - Can be a counter
2. The authenticated entity transforms the challenge
   - Using a secret (or private key) related to what the authenticator knows
   - The result is sent to the authenticator
3. The authenticator verifies the result
4. Calculates a result using the same method and challenge
   - Or produces a value from the result and evaluates if it is equal to the challenge, or to some related value

The challenge-response protocol is the workhorse of modern authentication. The authenticator issues a challenge - a nonce, ideally a fresh random value that is never reused - and the authenticated party responds with a value computed from the challenge and their credential, using a transformation method both sides share. The choice between a random nonce and a counter is a policy decision: a random value is stronger because it cannot be predicted, a counter is cheaper because it is a single increment, and both satisfy the core requirement that the challenge is fresh and unique per exchange.

The verification on the authenticator side can be direct - recompute the expected result with the same method and challenge, and compare - or indirect, when the transformation is reversible: the authenticator undoes the transformation and checks that it lands back on the original challenge. Both patterns appear in the protocols later in this lecture.

The choice of nonce is a security-critical detail. A predictable or repeated challenge lets an attacker replay an old response or precompute the whole response space, which is why the slide stresses "not once used". Fresh randomness is what separates a challenge-response protocol from a replayable exchange.

When you read a protocol description, identify the three roles immediately: who issues the challenge, who proves, and who verifies - and then check what the verifier must store, because that storage is where the design's weaknesses concentrate (passwords to hash, secrets to protect, or nothing but public keys). The two verification styles on the slide also map to the two big families: recomputation suits one-way constructions such as hashes and signatures, while reversal suits encryptive ones - and the difference is not cosmetic, because reversible constructions leak equivalence classes and are the usual home of the classic attacks on shared-secret schemes.

Challenge-Response approach

- Advantages
  - Authentication credentials are not exposed
  - An eavesdropper will see the challenge and the result
    - but cannot compute it without the secret
- Problems
  - Authenticated entities must have the capability of calculating results to challenges
    - Hardware token or software application
  - The authenticator may need to keep shared secrets (in clear text)
    - Secrets can be stolen
    - Individuals may reuse secrets in other systems, enabling lateral attacks
  - May be possible to calculate all results to a single (or all) challenge(s)
    - Can reveal the secret used
  - May be vulnerable to dictionary attacks
    - Authenticator should NEVER issue the same challenge to the same user

The central advantage is confidentiality of the credential: an eavesdropper on the channel sees only a challenge and a response, both one-off values, and learns nothing about the secret that produced them. This is what makes challenge-response usable over insecure channels in the first place.

The cost is capability: the authenticated party needs something that can perform the computation - a hardware token or a software application - and the authenticator must store the shared secret or a verification value derived from it. If the authenticator's database is stolen, the attacker inherits everything in it, and users who reuse secrets across systems hand the attacker a lateral-movement path for free.

The deeper risks are cryptanalytic: if the transformation is weak, the attacker can compute responses to any challenge from a stolen database, or run dictionary attacks against the stored values. The last point on the slide - never issue the same challenge twice to the same user - is the minimal defence against replay, and it is a hard requirement, not a recommendation.

The database point deserves the emphasis it rarely gets: challenge-response protects the channel, not the verifier's storage. If the authenticator stores the shared secret itself, a leaked database is a credential forge for every user in it, and the protocol's elegance on the wire is irrelevant. The fixes are the ones from password storage - store a one-way transformation instead of the secret, or move to asymmetric credentials where the verifier stores nothing sensitive at all - and the protocols that follow in this lecture are, to a first approximation, a tour of those fixes in increasing order of strength. The lateral-reuse problem, meanwhile, is invisible to the protocol itself: only deployment discipline, monitoring or user education can catch it. And whenever a scheme claims resistance to dictionary attacks, check whether the claim covers an attacker who already holds the verifier's database - several historical schemes fail exactly there.

Authn of Subjects: Challenge-Response with Smartcards

- Authentication Credentials
  - Having the smartcard (e.g., the Citizen Card)
  - The private key stored inside the smartcard
  - The PIN code to access the key
- The authenticator knows
  - The user's public key
- Robust against:
  - Dictionary attacks
  - Offline attacks to the database
  - Insecure channels





The smartcard realises challenge-response in the most complete form: the credential is a three-legged combination - possession of the card, the private key stored inside its chip (which cannot be extracted), and the PIN that unlocks the key. All three must be present, which makes the card a two-factor device: something you have plus something you know.

The authenticator only needs the user's public key, so the server-side database holds no secret worth stealing - a dramatic improvement over storing password hashes. The consequence, listed on the slide, is robustness against dictionary attacks, offline attacks on the database, and insecure channels: the three classic threats to the direct approach all disappear at once.

The Portuguese Citizen Card is the concrete example: one chip carries the citizen's identity, a bank card, a digital signature certificate and a TLS certificate - multiple credentials in a single tamper-resistant device, each with its own PIN and its own policy.

The three-legged credential is the design to imitate whenever the stakes justify a card: possession without the PIN is inert, the PIN without the card is unguessable in practice, and both together still prove nothing unless the chip cooperates - so an attacker needs the object, the number and the silicon. The public-key-only verifier has a second benefit worth spelling out: because the server holds no secret, its database can be replicated across sites without multiplying risk, and compromising one replica compromises nothing. Note also the division of labour that makes this workable: the card only signs what it is asked to sign, and all policy - which keys may sign what, and when - lives with the issuer and the verifier.

Authn of Subjects: Challenge-Response with Smartcards

- Challenge-Response Protocol
  1. The authenticator generates a challenge
  2. Smartcard signs the challenge with the owner's private key
     - Stored in the smartcard, protected by the PIN code
     - (often described as "encrypting with the private key")
  3. The authenticator verifies the signature with the public key
     - If the signature is valid for the challenge, authentication succeeds
     - Proves possession of the private key without revealing it

The protocol is a textbook asymmetric challenge-response. The authenticator generates a random challenge; the card signs the challenge with its private key (the slide calls this encrypting with the private key, a loose description that only holds for RSA, and even there the padding schemes differ from those used for encryption), and the authenticator verifies the signature with the public key. The protocol is minimal by design: three messages, no setup beyond the key pair, and the whole exchange takes milliseconds, which is what makes it practical for a device that has to answer to a network on every attachment.

Because the private key never leaves the chip, the attacker's options are exhausted by the three legs: steal the card (the PIN still protects it), guess the PIN (bounded attempts, then the card locks), or break the cryptography (infeasible at the key sizes in use). Every leg is defended, and the attacker needs all of them. The PIN attempt limit is a specific counter to the most likely attack: an attacker who has stolen the card can try PINs offline at the speed of the chip, and the lockout policy bounds that search to a handful of guesses before the card is unusable.

Signing the challenge is the correct operation: it produces a signature that can be verified with the public key and checked offline, which is how the same card also performs digital signatures on documents - authentication and signing share one key pair and one mechanism.

The "often described as encrypting with the private key" caveat is the kind of detail that separates correct from merely plausible implementations: real signature schemes (RSASSA-PSS, ECDSA, Ed25519) are different algorithms with different padding and different failure modes, and treating a signature as a decryption has produced real breaks. For building mental models: the card proves possession of the private key by producing a value only it could have produced, verifiable by anyone holding the public key and the same challenge - and the challenge's freshness is what makes yesterday's proof worthless today.

Authn of Subjects: Challenge-Response with other tokens

- FIDO2 tokens (FIDO Alliance)
  - Web Authentication (WebAuthn) specification
  - Client-to-Authenticator Protocol (CTAP)
  - Client generate service specific asymmetric keys



https://www.inovex.de/de/blog/fido2-webauthn-in-practice/

  - Security
    - Private keys never leave the authenticator
    - Phishing-resistant (origin-bound); tokens can be stolen
    - No replay attacks
    - Token certification levels
  - Privacy
    - Credentials are unique per website
    - Tracking is not possible (different web sites, different public keys for the same token)
    - Biometric data, when used, never leaves the user's device

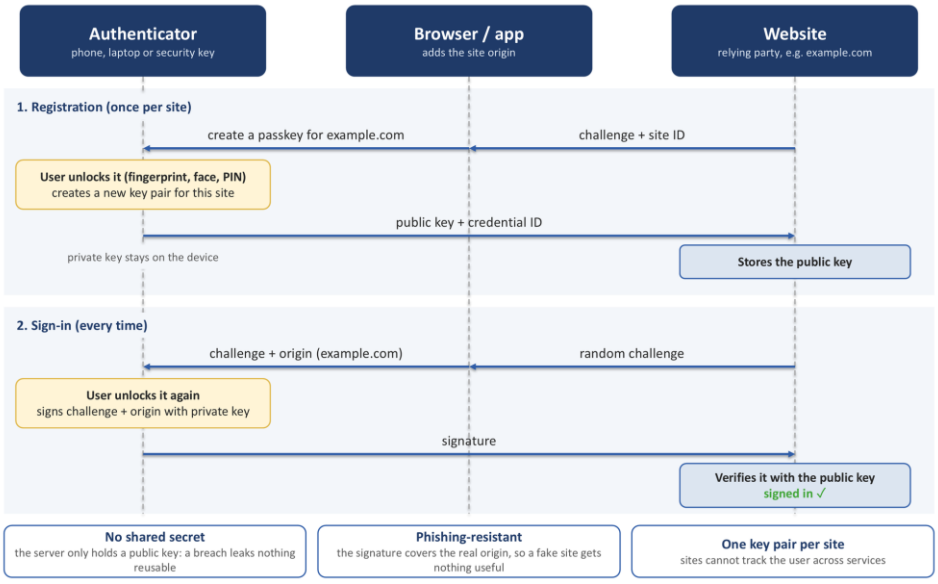
FIDO2 is the modern consolidation of the challenge-response idea. A FIDO2 authenticator (a hardware token, or a platform built-in such as a phone's fingerprint reader) generates a key pair per relying party: the private key stays on the device, and only the public key is registered on the server. The server holds a public key - there is nothing in its database worth stealing.

The two protocols are complementary. WebAuthn is the web-standard API that browsers use to talk to authenticators; CTAP (Client to Authenticator Protocol) is the low-level protocol between the client and the device, over USB, NFC or Bluetooth. Browsers, operating systems and tokens all speak the same language, which is what made passkeys a cross-ecosystem standard.

The security properties on the slide follow directly from the design. The private key never leaves the authenticator (synced passkeys are copied between the devices of the same user through an end-to-end-encrypted cloud backup) and the server stores only the public key, so there is no password database to steal and no password to phish; the origin of the request is bound into the authentication, which defeats phishing; and per-site key pairs make cross-site tracking with the same token impossible - the same physical token presents a different identity to every website.

The privacy consequence is worth stating plainly: no two websites can correlate you through your authentication device, because the public key they hold is unique to them. For the first time at scale, the authentication mechanism is also an anti-tracking mechanism. The property also has an operational side: because the token presents a different key to every site, the loss of one site's public key has zero value to an attacker, and the token can be revoked per site without touching the others.

Authn of Subjects: Challenge-Response with other tokens



FIDO2 is the modern consolidation of the challenge-response idea. A FIDO2 authenticator (a hardware token, or a platform built-in such as a phone's fingerprint reader) generates a key pair per relying party: the private key stays on the device, and only the public key is registered on the server. The server holds a public key - there is nothing in its database worth stealing.

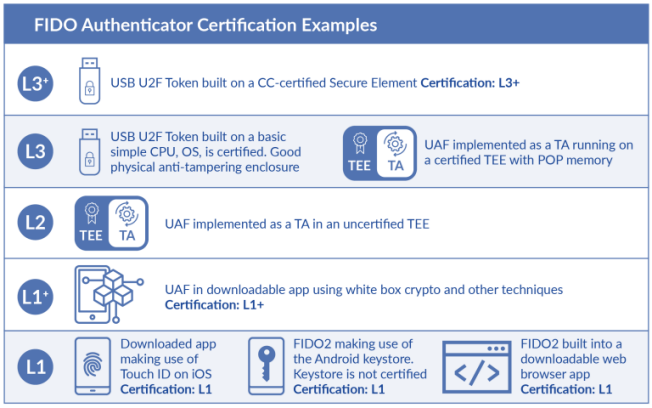
The two protocols are complementary. WebAuthn is the web-standard API that browsers use to talk to authenticators; CTAP (Client to Authenticator Protocol) is the low-level protocol between the client and the device, over USB, NFC or Bluetooth. Browsers, operating systems and tokens all speak the same language, which is what made passkeys a cross-ecosystem standard.

The security properties on the slide follow directly from the design. The private key never leaves the authenticator (synced passkeys are copied between the devices of the same user through an end-to-end-encrypted cloud backup) and the server stores only the public key, so there is no password database to steal and no password to phish; the origin of the request is bound into the authentication, which defeats phishing; and per-site key pairs make cross-site tracking with the same token impossible - the same physical token presents a different identity to every website.

The privacy consequence is worth stating plainly: no two websites can correlate you through your authentication device, because the public key they hold is unique to them. For the first time at scale, the authentication mechanism is also an anti-tracking mechanism. The property also has an operational side: because the token presents a different key to every site, the loss of one site's public key has zero value to an attacker, and the token can be revoked per site without touching the others.

• FIDO Alliance certification program for FIDO2 products

- Authenticators are tested for CTAP2/WebAuthn conformance and interoperability
  - hardware tokens and platform built-ins
- Certification levels L1/L2/L3 add security, tamper-resistance and side-channel tests
- The list of certified products is public - check it before deploying tokens



Certification is the ecosystem's answer to a question every deployer eventually faces: how do I know a token from an unfamiliar vendor implements the protocols correctly and protects its secrets? The FIDO Alliance runs the program that answers it: authenticators - hardware keys and platform built-ins alike - are put through conformance and interoperability runs against the CTAP2 and WebAuthn specifications, and only those that pass appear on the public list of certified products.

The levels have a concrete meaning. L1 certifies functional conformance: the device speaks the protocols and passes the functional check suites. L2 adds security requirements and a review of the implementation, and L3 adds physical security - tamper resistance and resistance to side-channel analysis - evaluated against defined attack scenarios. The level is a purchase criterion, not a decoration: a deployment that requires hardware-backed, phishing-resistant authentication should demand the matching level explicitly, because a device that merely "speaks FIDO2" may implement none of the protections the level implies.

The practical habit to take away: before buying tokens or enabling platform authenticators at scale, check the certification status and level on the public listing, and re-check when firmware updates ship - certification is granted per product and version, and the gap between a certified reference and a cheap clone is exactly the gap this program exists to close. In short: the listing turns vendor claims into third-party evidence, and consulting it is a five-minute step that closes an entire class of supply-chain surprises.

Authn of Subjects: Challenge-Response with Shared Secret

- Authentication Credentials
  - Password selected by the individual

- The authenticator knows
  - Bad approach: the shared password
  - Better approach: A transformation of the shared password
    - The transformation should be unidirectional (Argon2id, scrypt, PKBDF2)

This is the bridge between the direct and the challenge-response approaches: the credential is still a password (or a shared key) that the individual knows, but it is never transmitted - it is used locally to transform the challenge. The channel only ever carries one-off values.

The authenticator must hold something to verify against. Storing the shared password in the clear is the "bad approach" the slide names: anyone who reads the database has the password itself, and can authenticate as anyone. The better approach is to store a one-way transformation of the password, so a database leak reveals hashed values that still have to be attacked - a delay that can be the difference between an incident and a breach.

The difference from the smartcard case is where the secret lives: on the server, in a form that can at least be protected, rather than on an immutable chip. All of CHAP, MS-CHAP and the SIM-card protocols that follow are instantiations of this design, and their security depends on the strength of the transformation and the protection of the database.

The "bad approach / better approach" contrast is the whole design space in two words. With the clear password stored at the verifier, every reader of the database instantly becomes every user; with a one-way transformation, the leak becomes an offline guessing problem whose cost is the work factor - the same reasoning as password storage, now applied to a protocol that never sends the password anyway. The subtle point is that the transformation must be applied to the stored verification value, not improvised in the response: schemes that hash the password once and use the hash as the shared secret inherit a new weakness - anyone who steals the hash can authenticate with it directly, without ever learning the password. This is why shared-secret challenge-response is a transitional technology: it fixes the channel but not the storage, and the endgame - a verifier that stores nothing sensitive - arrives only with asymmetric credentials.

Authn of Subjects: Challenge-Response with Shared Secret

- Basic Challenge-Response Protocol
  - The authenticator generates a challenge
  - The individual calculates a transformation of the challenge and the password
    - result = hash(challenge || password)
    - or... result = encrypt(challenge, password)
  - The authenticator recomputes (or reverts) the process and checks if the values match
    - result == hash(challenge || password)
    - or .... challenge == decrypt(result, password)
  - Examples with shared passwords: CHAP, MS-CHAP v1/v2, S/Key
  - Examples with shared keys: SIM & USIM (cellular communications)

The basic protocol pairs a shared secret with a fresh challenge. The user computes result = hash(challenge || password) - or, with a symmetric cipher, encrypts the challenge with the password - and the authenticator, which knows the password (or its hash), recomputes the expected value and compares.

The hash variant is one-way, so a stolen database reveals at most hashed passwords that must still be attacked; the encryption variant is reversible, which is a trade-off some legacy protocols accept. CHAP and MS-CHAP v1/v2 are the password-based examples; the SIM and USIM cards of cellular networks are the shared-key examples, with the key preloaded at manufacture. The cellular examples are the shared-key family at scale: the key is installed once at manufacture, the phone and the home network are both managed, and the protocol has run on hundreds of millions of devices for decades, which is a strong argument for the design.

The construction challenge || password (or its HMAC form) is what makes each response unique per challenge: the same password produces a different result every time, so a captured response cannot be replayed later. One-time-ness comes from the challenge, not from the secret.

The two formulas on the slide are worth comparing as a designer. hash(challenge || password) keeps the proof one-way and lets the verifier store a hash instead of the password; encrypt(challenge, password) is reversible, so the verifier must hold the password itself, and a weak cipher turns the response into a guessing oracle. Real protocols therefore prefer HMAC-style constructions over a derived key, and MS-CHAP's history shows what happens otherwise: its single-key, DES-based response was broken exactly at the stored-secret level. The cellular examples - SIM and USIM - are the same construction with a hardware-protected key and operator-chosen algorithms: an idea whose strength is decided entirely by the quality of the algorithms and the protection of the key.

PAP and CHAP (RFC 1334, 1992, RFC 1994, 1996)

- Protocols used for PPP (Point-to-Point Protocol)
  - Unidirectional authentication
    - The authenticator authenticates users, but users do not authenticate the authenticator

- PAP (Password Authentication Protocol)
  - Simple presentation of a UID/password pair
  - Insecure transmission (in clear text)

- CHAP (Challenge-Handshake Authentication Protocol)

```
Aut → U  : authID, challenge
U   → Aut: authID, MD5(authID, secret, challenge), identity
Aut → U  : authID, OK/not OK
```

  - The authenticator can request further authentication at any time

PAP (Password Authentication Protocol) and CHAP (Challenge-Handshake Authentication Protocol) are the authentication options of the Point-to-Point Protocol, the workhorse of dial-up and still the basis of many PPP-based links, including VPN concentrators. Both are unidirectional: the authenticator verifies the user, but the user has no equivalent check on the authenticator - a design choice that simplifies the protocol and leaves the user exposed to a fake authenticator. The unidirectionality is a deliberate simplification for a point-to-point link: the authenticator is the trusted end, the access concentrator, and the protocol assumes the user has no independent way to verify it, an assumption that is acceptable on a dial-up line and not on an open network.

PAP is the minimal option: the user sends their user ID and password in clear text. It exists mainly for compatibility with systems that cannot do anything else, and the slide is right to call it insecure - it should only be used on a link already protected by encryption.

CHAP replaces the clear password with a challenge-response: the authenticator sends a challenge, the user responds with MD5(Identifier || secret || challenge), and the authenticator compares with its own copy. The authenticator may re-issue challenges at any time during the session, which also catches a link that has been hijacked after the initial authentication - a property PAP structurally lacks.

The historical framing explains their persistence: both were designed for a point-to-point link whose two ends were a modem and an access concentrator, so the "channel" was assumed to be a dedicated wire - an assumption that connections across the open internet did not inherit. The MD5 in CHAP is also a period detail: it was a reasonable digest choice in 1996, but the construction stores a password-equivalent secret at the verifier, so a database leak is fatal regardless of the hash's later reputation. Modern deployments keep the PPP framing but run it inside an encrypted tunnel, which repairs the cleartext problem and leaves the design lesson: protocol reuse is safe only when the assumptions it was built on travel with it.

GSM Subscriber authentication

- Uses a secret shared between the HLR/AuC and the subscriber's SIM
  - Uses 128-bit shared key (not an asymmetric key pair, Ki)
  - Key is **stored in the SIM card**
  - SIM card is **unlocked by a user PIN**
  - SIM card answers challenges using the shared key
- Uses (initially secret) algorithms:
  - A3 for authentication
  - A8 to generate the session key
  - A5 is a stream cipher for communication
- A3 and A8 executed by the SIM, A5 executed by the baseband
  - A3 and A8 can be chosen by the operator

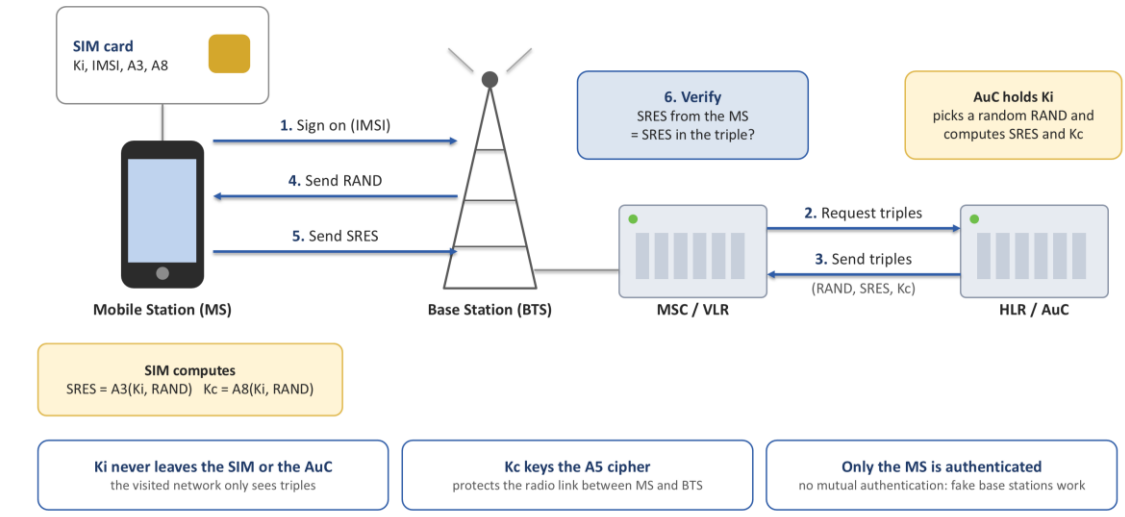
GSM (2G cellular) authenticates the subscriber to the network with a shared 128-bit key Ki stored in the SIM card, which is unlocked by a PIN. The network challenges the SIM; the SIM answers using Ki. Note the direction: in 2G the network authenticates the phone - the phone has no equivalent check on the network, a weakness later fixed in UMTS. The direction of authentication is a design decision with real consequences: the network proves nothing to the phone, so a fake base station can harvest the SIM's responses, and the later fix, mutual authentication, is a direct answer to this gap.

The three algorithms split the job: A3 computes the authentication response (SRES), A8 derives the session ciphering key (Kc), and A5 is the stream cipher that protects the over-the-air traffic. A3 and A8 run inside the SIM, A5 in the baseband processor; the operator chooses which algorithm set to deploy, which is both a flexibility and a risk.

The original algorithm sets were later found weak - A5/1's cipher was publicly broken and the 64-bit session key was short by modern standards. It is a standing reminder that "the operator can choose" means the security floor is set by the weakest deployment in the network. The initially-unknown phrase on the slide is a historical footnote: the algorithms were proprietary at launch, and their later public analysis is what found the weaknesses, a reminder that closed algorithms are not a security feature.

For the reader, this slide is a case study in separation of duties inside a phone: the SIM is a tamper-resistant element that owns the identity (the key Ki) and the subscriber's cryptographic answers, while the baseband processor owns the radio and the session cipher. That split is why the PIN protects the card but not the conversation, and why cloning attacks target the card's algorithms rather than the handset. It also previews the architectural pattern - credentials in a secure element, algorithms chosen by the operator - whose strengths and weaknesses the 3G design that follows addresses one by one.

GSM Subscriber authentication



- RAND, random value (128 bits)
- SRES = A3 (Ki, RAND) (32 bits)
- Kc = A8 (Ki, RAND) (64 bits)

Ki never leaves the SIM and the home authentication centre, so the home network precomputes "triplets" (RAND, SRES, Kc) and hands them to the visited MSC/VLR, which can then authenticate the phone without knowing Ki. The mobile switching centre (MSC) requests one or more triples from the home location register / authentication centre (HLR/AuC), which generates a random RAND and runs the subscriber's key Ki through A3 and A8 to produce the expected response and the session key.

The COMP128 implementation of A3/A8, recommended by the GSM association, is a proprietary hash-like compression function, not a block cipher; one call yields both the 32-bit response and the 64-bit session key. COMP128-1 was broken in 1998, which allowed Ki to be extracted and SIMs to be cloned. When the MSC later receives the phone's SRES, it simply compares it against the stored SRES - the home network did the cryptographic work in advance, and the visiting network never sees Ki.

Precomputation is the pattern: it moves heavy cryptography off the mobile path and onto the home network, at the price of storing triples that must be consumed before they can be reused - a stored SRES for a given RAND is a replayable response, so the network rotates challenges aggressively.

The numbers are the design's own critique: a 32-bit response is small enough to precompute against, and a 64-bit session key gave the A5 stream cipher a brute-forceable core - both choices made for 1980s silicon and never revised in the standard's lifetime. The triplet distribution also created a structural risk that the next generation fixed: a visited network holding many valid triples can authenticate the phone but also replay them, so triples had to be managed as one-time material and consumed in order. When you compare this with the vector-based design that replaced it, watch how each weakness here maps to a mechanism there: weak algorithms become published ones, and replayable triples become single-use, network-bound vectors.

UMTS Subscriber authentication

- Adds Mutual Authentication
  - The phone checks the network with AUTN, whose MAC = f1(K, SQN‖RAND‖AMF) can only come from someone who knows K
- Adds Replay Protection
  - Sequence number (SQN). The USIM rejects vectors that aren't fresh.
  - If the counters drift apart, a resync procedure (AUTS) fixes it
- Stronger keys (128bits)
  - CK and IK
- Integrity protection
  - IK protects signalling messages (f9), so injected or modified commands are detected
- More robust cryptography
  - Published and reviewed: KASUMI-based f8/f9, later SNOW 3G; MILENAGE (AES-based) for f1–f5

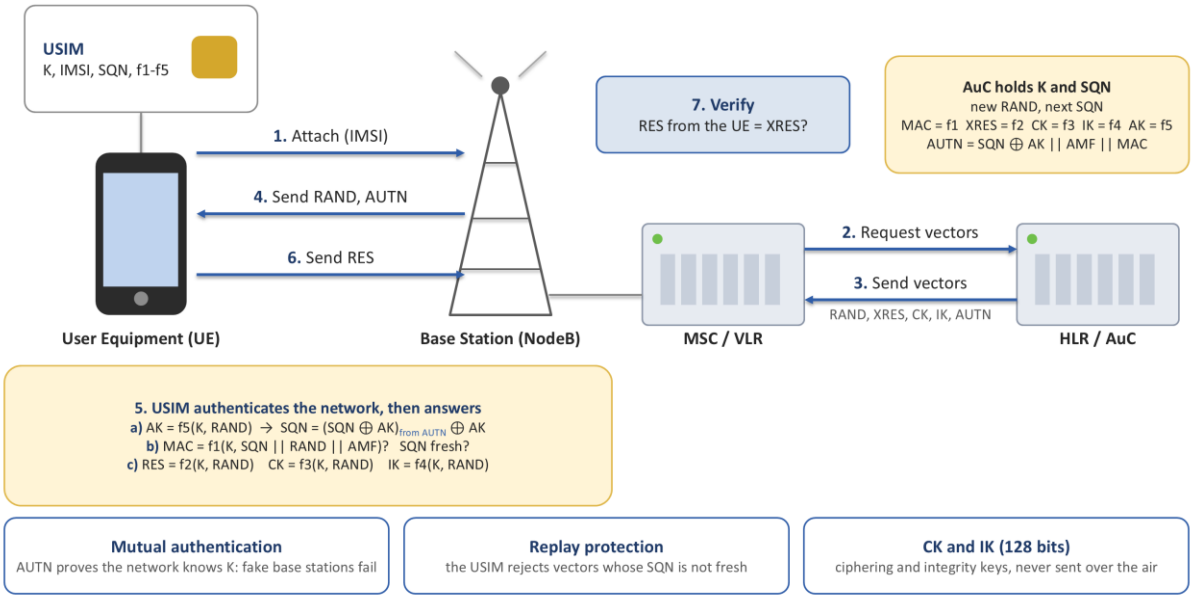
UMTS (3G) upgrades the same architecture with 128-bit keys throughout and, crucially, mutual authentication: the USIM card also verifies the network. The figure shows the full exchange: the MSC requests authentication vectors from the HLR/AuC given the IMSI; each vector is (RAND, XRES, CK, IK, AUTN), where AUTN = SQN XOR AK || AMF || MAC carries the masked sequence number, the authentication management field and the message authentication code. CK and IK stay in the network and the USIM and are never sent over the air.

The USIM checks the vector before answering - the SQN must be fresher than the card's own copy, and the MAC must verify - then produces RES and the working keys and stores the new SQN. Only then does the phone send RES, which the MSC matches against XRES. If the card rejects the vector, the session simply does not start.

The glossary on the slide is the reference: the f1-f5 functions derive everything from K and RAND; AK exists to hide the sequence number, which would otherwise reveal how long a subscriber has been active; and the 128-bit IK provides message integrity that 2G lacked. The design shows how a mature protocol layers freshness, integrity and confidentiality on top of a single shared key.

The mutual check deserves the emphasis: the phone verifying the network is what kills the fake-base-station attacks that 2G tolerated, and it works because the MAC inside AUTN could only be produced by someone holding the subscriber's key - the network proves knowledge of the key without revealing it. The resynchronization procedure (AUTS) is the kind of detail that decides whether a real deployment works: counters drift when phones roam or power-cycle, and without a fresh, authenticated resync message the user experiences "no service" instead of a security event. Note also the key hierarchy: one permanent key, and per-session cipher and integrity keys derived from it, so the long-term secret is used only inside the derivation - the same pattern the secure-communication protocols later in the course apply with master secrets and session keys.

UMTS Subscriber authentication (AKA)



This figure is the complete UMTS authentication and key agreement (AKA) exchange, worth reading in two halves. On the home network side, the authentication centre draws a fresh RAND and takes the subscriber's sequence number, and computes the vector: XRES (the expected response), CK and IK (the cipher and integrity keys), and AUTN, which packs the masked sequence number, the key-management field and the MAC computed with f1.

On the phone side, the USIM first verifies AUTN - recomputing the MAC and checking that the recovered sequence number is genuinely newer than its own - and only then computes RES and the working keys with the f-functions. The order matters: a USIM that answered before verifying would be authenticating a possible attacker, which is exactly the 2G weakness this design removes. The MSC's last step is the comparison the whole exchange exists for: RES against XRES.

The functions f1-f5 derive every value in the vector from the single long-term key and the fresh RAND, and the anonymity key exists only to mask the sequence number, which would otherwise broadcast how long the subscriber has been on the network. The design is a template worth remembering: freshness, mutual proof, key separation and privacy carried by one round of functions over one shared key. If you remember one thing from the figure, make it the order of operations: verify first, then answer - every message the card sends presumes the network has already proved itself. The same exchange also distributes the session keys, so authentication and key agreement happen in a single round, the AKA in the title.

Authentication of Systems

- By name (DNS) or MAC/IP address
  - Extremely weak, without cryptographic proof
  - Still... it is used by some services
  - e.g., NFS, TCP wrappers

- With cryptographic keys
  - Secret keys, shared between entities that communicate frequently
  - Asymmetric key pairs, one per host
  - Public keys pre-shared with entities that communicate frequently
  - Public keys certified by a third party (a CA)

Hosts and systems authenticate in a weaker range than people do. Name-based identification - "I am the host at this IP/MAC" via DNS or ARP - provides no cryptographic proof at all: anyone on the network can answer for any name. It persists because some legacy services (NFS, TCP wrappers) were built on it, and it is only acceptable on networks where address spoofing is itself hard. The name-based option is a legacy with a specific failure mode: DNS and ARP are lookup mechanisms, not authentication, so the proof is the absence of a contradicting lookup, and a single spoofed response on the segment is enough to impersonate any host.

The credible options are cryptographic. Shared secret keys work for small, stable groups of machines that talk to each other constantly; asymmetric key pairs, one per host, scale better; and the strongest arrangement is to have the host keys certified by a CA - exactly the PKI machinery from earlier in this course, applied to servers instead of people.

This is the substrate for what follows: TLS, SSH and RADIUS all need the servers (and often the clients) to be identifiable in a way that survives the open internet. A host that cannot prove who it is is a host that cannot be trusted with a session key.

The name-address bullet is a reminder that identification and authentication are different acts: DNS and ARP answer "where", never "who", and their answers are only as trustworthy as the network that carries them. The cryptographic ladder on the slide reads as a maturity path: shared keys are the cheapest and scale worst (a key for every pair of hosts), per-host key pairs remove the sharing but need distribution, and certification adds the third party that makes distribution tractable at scale. Most of what follows in this lecture - TLS server certificates, SSH host keys, AAA shared secrets - is a point on this ladder, and choosing the rung is a classic design decision: the cost of the certification machinery versus the cost of managing pre-shared keys.

Authentication of Services

- Authentication of the host
  - All services co-located in the same host are automatically and indirectly authenticated
- Credentials exclusive to each service
- Authentication:
  - Secret keys shared with clients
    - When they require authentication of the clients (e.g. MS-CHAPv2, RFC 2759)
  - Asymmetric key pairs by host/service
    - Certified by others or not

Services deserve their own authentication discussion because a single host typically runs many of them. Authenticating the host indirectly covers every service on it, but that is a coarse brush: a compromised web service should not be able to act as the mail service, and a stolen host key should not unlock every protocol on the box.

The stronger design gives each service its own credentials. Where the service must also authenticate its clients, a shared secret may be used (MS-CHAP v2 over PPP/RADIUS is the classic example); more commonly each host or service holds an asymmetric key pair, optionally certified, and presents the corresponding credential when a peer demands it.

The practical consequence is the certificate-per-service model that dominates today: the same machine holds distinct, independently revocable identities for each service it exposes, so compromise and revocation stay contained.

The indirect (host-level) option is common because it is cheap: one certificate, one key, one renewal date for everything on the machine. Its failure mode is concentration: the web server's credential speaks for the database, the mail service and the admin interface, so one compromised process or one leaked key file is a total compromise. Per-service credentials invert the trade: more provisioning work, but revocation, monitoring and least privilege become per-service decisions.

In practice the split follows the deployment: containers and virtual machines tend to inherit host credentials through a platform (a load balancer or service mesh terminating the connection), while installations that must survive partial compromise issue per-service certificates from their own internal CA. When you meet a system, the question "who am I actually talking to - the host or the service?" is answered by looking at which key authenticated the session.

TLS (Transport Layer Security, RFC 8446)

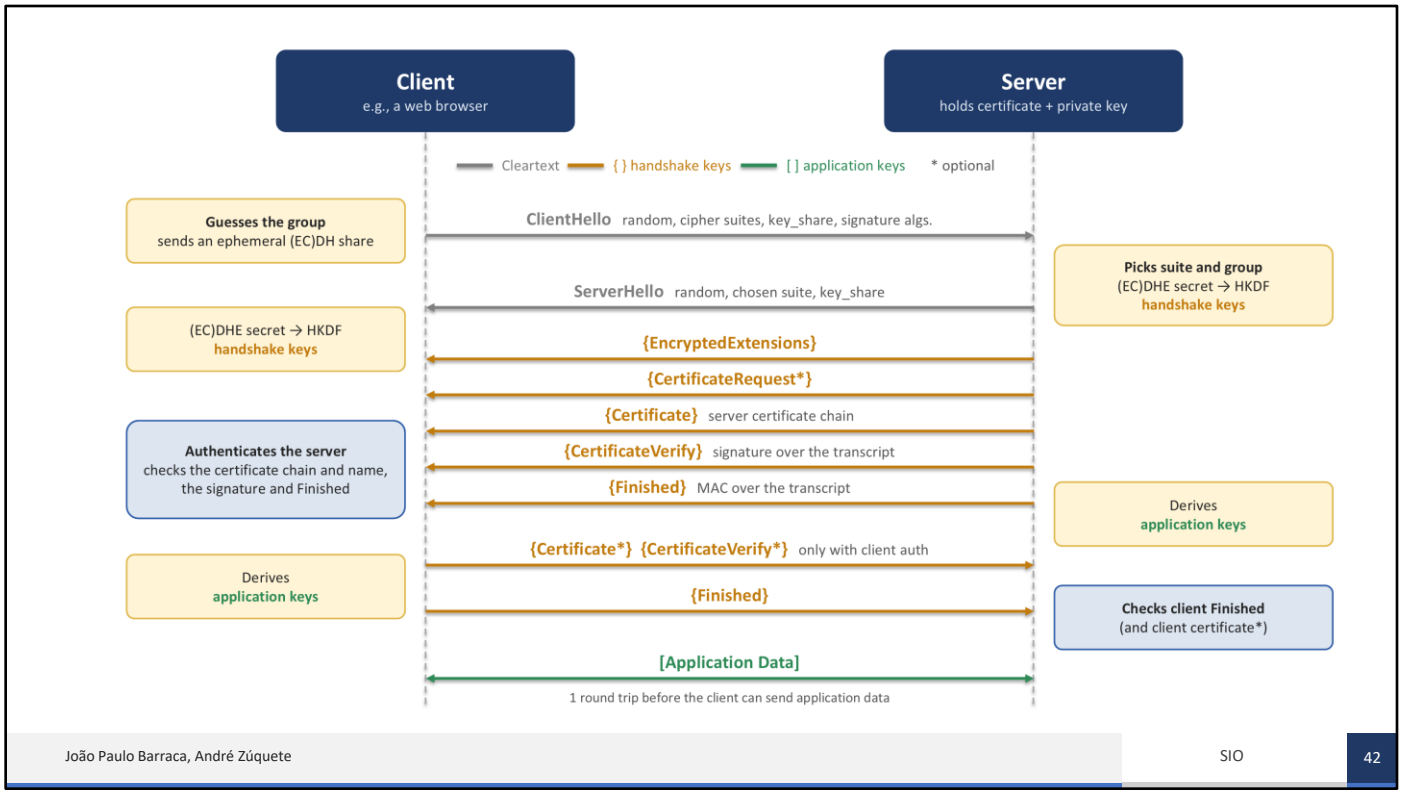
- Secure Communication Protocol over TCP/IP
  - Evolved from the SSL V3 (Secure Sockets Layer) standard
  - Manages secure sessions over TCP/IP, individual to each application
  - Initially designed for HTTP traffic
    - Currently used for many other types of traffic
- Security mechanisms
  - Confidentiality and integrity of the communication between entities
    - Key distribution, negotiation of ciphers, digests and other mechanisms
- Authentication of the participating entities
  - Servers, services, etc... (normal, but may be disabled)
  - Clients (not so common)
  - Both executed with asymmetric keys (not common) and X.509 certificates (common)

TLS (RFC 8446, version 1.3) is the standard for securing communication over TCP/IP, descended from SSL v3. It is protocol-agnostic - "initially designed for HTTP traffic" is now historical - and carries mail, database, VPN and game traffic in equal measure.

Its job has three parts: confidentiality and integrity of the session, with keys negotiated fresh for each connection; authentication of the participants; and the glue that makes the rest of the stack possible. Server authentication is the usual case, client authentication is less common but standard, and mutual authentication is built in - all of it rides on asymmetric cryptography, with X.509 certificates the common mechanism, tying TLS directly to the PKI lecture.

TLS 1.3 simplified the handshake to a single round trip, removed the legacy algorithms, and made forward secrecy the default - a good illustration of how a mature protocol prunes its attack surface over time instead of accumulating options.

The authentication bullets describe deployment reality rather than the protocol's capability: server authentication with X.509 certificates is universal, client certificates are rare outside closed ecosystems (banks, enterprise VPNs, machine-to-machine fleets), and both directions can be disabled - which is why "TLS" alone never implies "authenticated peer". The protocol's own evolution is a lesson in maintenance: SSL v3 and early TLS carried algorithms that later analysis demolished, and the current version succeeded by deleting them and encrypting more of the handshake, shrinking the surface instead of patching around it. The skeleton - authenticate, negotiate keys, then protect - is the one you will recognise in every secure session protocol that follows. Note also that TLS authenticates once per session, and the identity it proves is the one in the certificate: a hostname, not a person, which connects back to the system-authentication discussion.



This figure illustrates the TLS session: the encrypted channel that sits between the two endpoints once the handshake has completed. Everything that would be visible in the clear on the outside becomes ciphertext on the inside, and every record carries an integrity tag, so tampering is detectable as well as prevented.

The handshake that produced this state is the subject of the following slides: the exchange of supported mechanisms, the key negotiation, and the authentication of the server. The only part of TLS most users ever see is the padlock - the visual shorthand for "the server authenticated itself to me, and from here on the channel is private and bound".

The figure shows the session state after the handshake: two endpoints, a negotiated ciphersuite, and a bidirectional flow of protected records. Each record carries a sequence number and an authentication tag, so replay, reordering and tampering are detected, and the traffic keys are fresh per session and derived from the handshake's key exchange, which is what gives forward secrecy: recording today's traffic pays off only if the session keys themselves can later be broken, not the server's long-term key.

What the figure does not show is everything the handshake settled before the first data record: the version, the ciphersuite, the server's certificate chain and its verification, the ephemeral key exchange, and the derived keys. That invisible negotiation is where the security actually lives - the padlock in the browser is the user-facing summary of a chain of checks (name matches certificate, chain validates, suite is acceptable, keys are fresh), any of which can silently fail if skipped. The session is the reward; the handshake is where the work happens.

TLS Ciphersuites

- If a server supports a single algorithm, it cannot be expected for all clients to also support it
  - More powerful server/limited IoT, older/newer
- The ciphersuite concept allows the negotiation of mechanisms between client and server
  - The client sends its supported ciphersuites
  - The server chooses one they both share
- Example: ECDHE-RSA-AES128-GCM-SHA256
  - Key negotiation algorithm: ECDHE (Elliptic Curve Ephemeral Diffie-Hellman)
  - Authentication algorithm: RSA
  - Cipher algorithm and cipher mode: AES-128 Galois/Counter Mode
  - Hash for key derivation (PRF): SHA-256 (integrity comes from GCM)

No single algorithm combination serves every client: hardware capability, protocol version and policy all differ. The ciphersuite concept solves this by making the algorithm set a negotiable unit - the client offers the list it supports and the server picks one from it. The negotiation is part of the handshake, before any application data flows.

In practice the server makes the final choice (the slide's "the server chooses"), which concentrates the policy decision in one place: the server operator can retire weak suites without any client change. Clients influence the choice by what they offer; the server decides what is acceptable. The server-side choice is a policy decision with a cost: it concentrates the retirement of weak suites in one place, but it also means a misconfigured server can force a weak suite on every client that accepts the choice, which is why client-side minimums exist as a counterweight.

The example ECDHE-RSA-AES128-GCM-SHA256 unpacks the naming scheme: ECDHE is the key-exchange (ephemeral Elliptic-Curve Diffie-Hellman, giving forward secrecy), RSA is the server authentication mechanism, AES-128-GCM is the encryption (an authenticated cipher, so encryption and integrity are one mechanism), and SHA-256 is the hash used by the PRF for key derivation, not for integrity, which is provided by GCM itself as an AEAD mode. TLS 1.3 suites (for example TLS_AES_128_GCM_SHA256) name only the AEAD cipher and the hash; key exchange and signature algorithm are negotiated separately.

SSH (Secure SHell)

- Manages secure console sessions over TCP/IP
  - Initially designed to replace the Telnet application/protocol
  - Currently used in many other applications
    - Execution of remote commands in a secure manner (rsh / rexec)
    - Secure copy of contents from/to remote hosts (scp, replacing rcp)
    - Secure file transfer (SFTP, not FTP)
    - Secure (generic) communication tunnels (TCP port forwarding)

- Security Mechanisms
  - Confidentiality and integrity of the communications
    - Key distribution
  - Authentication of the participating entities
    - Server / Hosts
    - Client users
    - Both achieved through several, and differentiated mechanisms

SSH is the Telnet killer: a secure console session over TCP/IP that, over two decades, grew into a general-purpose secure channel. Remote command execution (replacing rsh/rexec), secure file transfer (SFTP, the SSH File Transfer Protocol, which is not FTP over SSH), and generic port-forwarding tunnels all ride on the same protocol and the same security model.

Its security model mirrors TLS's: key negotiation for confidentiality and integrity, plus authentication of the participants. Server authentication, client-user authentication, or both - with different mechanisms for each direction, as the slide says. The session is established before any user data crosses, and everything inside it is encrypted and integrity-protected.

The tunneling feature is what makes SSH a network tool rather than just a terminal: any TCP connection can be carried inside it (port forwarding carries TCP streams; IP packets only with tun-based tunnels), which is how administrators reach internal services that are not exposed to the open internet - a poor man's VPN with per-connection control.

The historical framing is more than trivia: SSH was built by a community that assumed the network was hostile from the first packet, which is why authentication of both ends is part of session setup rather than an add-on. Its authentication model is deliberately plural - host keys, user keys, passwords, PAM-bridged mechanisms, and certificates in modern implementations - because the protocol negotiates methods, letting an installation choose its own point on the convenience-versus-assurance curve. For the reader, SSH is the best example in this lecture of a protocol whose security depends on operational discipline (key management, known-hosts hygiene, algorithm choices) as much as on cryptography. Its ubiquity in administration also means its authentication failures are high-value events: a hijacked SSH session is a shell on the server, with everything that implies.

SSH: Authentication mechanisms

- Server: an asymmetric key pair
  - Public keys are distributed during the interaction
    - Usually not certified (OpenSSH certificates exist)
  - Clients store the public keys from previous interactions
    - Key should be stored in some trusted environment
    - If the key changes the client user is warned
      - e.g., server is reinstalled, key is regenerated, an attacker is hijacking the connection
      - Client can refuse to continue with the authentication process

- Clients: authentication is configurable
  - Default: username and password
  - Recommended: username + private key
    - The public key MUST be pre-installed in the server
  - Other: integration with PAM for alternative authentication mechanisms

The server side of SSH is deliberately simple and, as the slide warns, not certificate-based. The server presents a public key; the client compares it against the key from previous sessions and warns on any change. A changed key is genuinely ambiguous - a reinstalled server, a regenerated key, or an active attacker - and the client can refuse to proceed.

This trust-on-first-use model is the practical weakness: the very first connection to a host has no prior to compare against, and a MITM at that moment wins forever. Mitigations exist - known-hosts files managed from inventory, SSHFP DNS records (validated with DNSSEC), or routing the first contact through a channel you already trust - but the default experience is "trust me on first use", and most users never see the warning. OpenSSH also supports host and user certificates signed by an SSH CA, which removes trust-on-first-use because clients only need to trust the CA key.

The client side is configurable: the default is username and password (encrypted by the session, not exposed on the wire), the recommended mode is a private key whose public half is pre-installed on the server, and PAM integration allows other mechanisms. Key-based client authentication is the norm in any serious administration practice, and it is what makes SSH comfortable for automation.

The asymmetry between the two directions is the teaching point: the server proves itself with a key pair and a warning system, while the client proves itself with whatever the configuration allows - and in practice the client side is where accounts get taken over, through stolen private keys without passphrases or agents left unlocked. The operational minimum is worth memorising: host keys pinned and monitored, user keys with passphrases, agent forwarding off unless needed, and direct root login disabled - each item closing one of the paths that password-only SSH leaves open.

Centralized network authentication

- Used for restricting network access to known clients
  - In cabled networks
  - In wireless networks
  - In VPNs (Virtual Private Networks)

- Usually implemented by a central service
  - AAA server
    - Authentication, Authorization and Accounting
    - e.g. RADIUS and Diameter
  - This server defines which network services the user can make use of

freeRADIUS

Centralized network authentication gates access to a network - wired, wireless, or VPN - through an AAA server that handles Authentication, Authorization and Accounting. RADIUS (the older, UDP-based protocol) and Diameter (its successor, running over TCP or SCTP) are the two protocols that carry the AAA exchange between the network access device and the server. The AAA model is the network version of the CA and RA separation from the PKI lecture: the device enforces, the server decides, and the credential store is central, the same architecture applied to network access instead of certificates.

The key design point is separation: the access device (switch, access point, VPN concentrator) only enforces the verdict; the AAA server holds the policy and the credential store. "Which services can this user use?" is a question the server answers, not the device - the device is a thin gate with a decision to relay.

This is the architecture behind every enterprise Wi-Fi login and most VPN gateways: the device you connect to knows nothing about you, and the identity store is central. It scales, it centralises policy, and it concentrates the credential data in one protected place - a trade explored on the next topic.

The AAA split is worth memorising because it recurs everywhere: authentication (who are you), authorization (what may you do) and accounting (what did you do) are three questions with different answers, different logs and different failure modes, even when one server handles all three. The protocol split matters too: the older, UDP-based protocol pairs with older equipment, its successor runs over reliable transports, but both carry the same architecture - the access device is a policy-free gate, and the verdict always comes from the centre. The port-based access mechanism used in wired and wireless enterprise networks is the standard consumer of this design.

Centralized authentication

- Advantages:
  - Can reuse same credentials over multiple systems/services
  - Single secure repository for credentials
    - More difficult to steal credentials when used in many services
  - Can implement restrictions to services/systems

- Disadvantages:
  - Requires additional servers
  - Single point of failure: without authentication systems, no one will be authenticated
    - Important to also deploy local credentials for admins
  - Introduces delays in the authentication process
  - Privacy issues (tracking because it records every device/user session)

The advantages are operational: one credential set works across systems, the credential repository is a single hardened target rather than a forest of local password files, and service-level restrictions (this user may reach the guest VLAN but not the lab network) are expressible in one place and enforced everywhere.

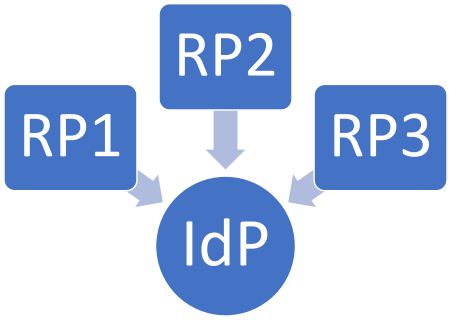
The disadvantages are equally operational. The central server is a single point of failure - which is why administrators keep local credentials for the few people who must work when the AAA server is down - it adds latency to every authentication, and it creates a tracking database: every device, every session, recorded in one place, which is a privacy exposure as well as an attack target.

The lesson is that centralization trades distributed risk for concentrated control; the design job is to make the concentrated risk survivable - redundancy for availability, local admin fallback for emergencies, and minimisation of what is logged, so the accounting database is as small as the audit needs.

The single-point-of-failure bullet is the one that bites in practice, and the mitigation is a decision made before deployment: redundant AAA servers, cached verdicts for already-connected devices, and local break-glass credentials whose use is itself logged and alerted. The privacy bullet is subtler: an AAA database is a behavioural record (who connected from where, when, for how long) that outlives the sessions it describes, so retention limits apply to it like any other personal-data store. A useful exercise when auditing a centralized design is to enumerate what stops working when the centre does - and to check that the answer is "administrators with local credentials", not "everyone".

Authentication by an IdP (Identity Provider)

- Unique, centralized authentication for a set of federated services
  - The identity of a client, upon authentication, is given to all federated services
  - The identity attributes given to each service may vary
  - The authenticator is called Identity Provider (IdP)
  - The federated service is called a Relying Party (RP)
  - In some cases, the provided identity attributes are shown to the client



- Examples
  - Authentication at UA
    - Performed by a central, institutional IdP (idp.ua.pt)
    - The identity attributes are securely conveyed to the service accessed by the user
  - Autenticação.gov (www.autenticacao.gov.pt)
    - Performed by a central, national IdP
    - The identity attributes are shown to the user
  - Other:
    - Services used worldwide: Google, Facebook, etc.

An Identity Provider is the central authenticator for a set of federated services. The user authenticates once to the IdP, and the IdP then asserts the user's identity - possibly with different attribute sets for different services - to each Relying Party that accepts it. The relying party never sees the password; it sees an assertion it can verify.

The two Portuguese examples on the slide are the clearest illustration. The university IdP (idp.ua.pt) authenticates students and staff once, and each campus service receives the identity securely in the background; the national authentication portal shows the user the attributes being shared, adding a visible consent step. Google or Facebook login on third-party sites is the same pattern at consumer scale.

The federation model is what makes "log in once" possible across organisations that would otherwise never share password stores: the IdP is the only party that sees the credential; the relying parties see an assertion. Trust is delegated to the IdP, which is exactly why the IdP's own security is the most important security question in the federation.

The varying-attributes bullet is where federation becomes a policy instrument: the IdP decides which service learns your name, your email, your department or only an opaque identifier, and the privacy of the whole system lives in that decision table. The two Portuguese examples sit at the ends of the transparency spectrum - institutional attributes flow silently in the background, while the national portal makes the release of attributes an explicit, user-visible step - and both patterns are legitimate, chosen according to how sensitive the attributes are. Keep the roles straight as you meet the protocols that carry these assertions: the IdP authenticates and asserts, the relying party consumes the assertion, and neither needs to share a password store with the other - that separation is the whole point.

Single Sign-On (SSO)

- A facility usually associated with IdP
  - Neither is mandatory nor always appropriate

- SSO exists for simplifying users' lives
  - They log in just once to access several federated services during a given period

Single Sign-On is the user-visible half of federation: after one login to the IdP, the user reaches many services during a session window without re-authenticating. The slide correctly notes that IdP and SSO are related but distinct concepts - a system can federate identities without offering transparent SSO, and vice versa.

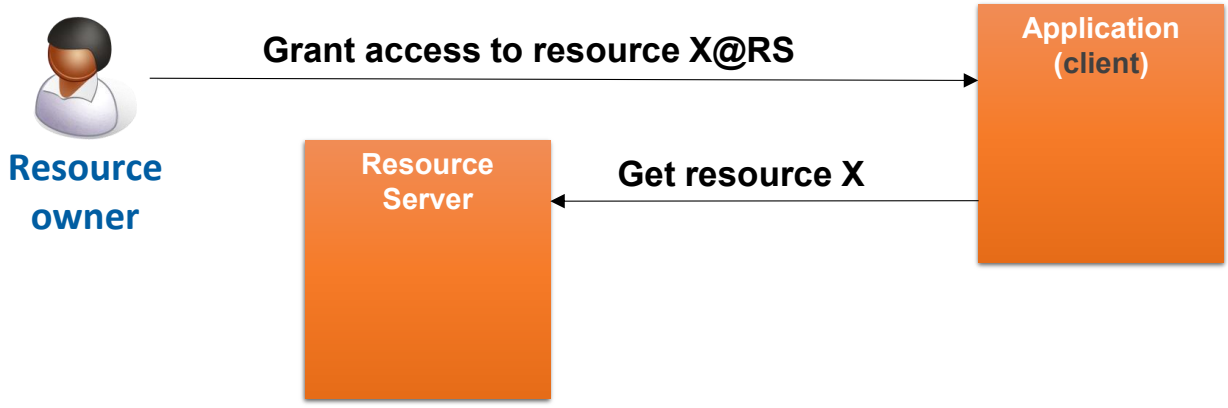
The security story of SSO is the session credential: the IdP issues a token (a SAML assertion, an OIDC token, a federated cookie) that the relying parties accept in place of a fresh login. The risk profile shifts from "many weak passwords" to "one session token of high value" - losing it is more expensive, so its protection matters more: short expiry, binding to the client, and revocation on logout.

For the user the benefit is real: one credential, one password to remember, one phishing surface to protect. For the operator the benefit is one place to enforce policy, and one place to kill a session when something goes wrong.

The caveat - neither mandatory nor always appropriate - is the part to internalise. SSO concentrates risk by design: one powerful session unlocks many doors, so the session's lifetime, its binding to the client device, and the ability to terminate it centrally become the controls that matter. A common refinement is partial SSO: silent sign-on for low-risk services, fresh authentication for sensitive ones, with the IdP remembering both decisions separately. For the user the model changes what "logging out" means: ending the session at one service does not end the session at the IdP, which is why federation portals provide an explicit "sign out everywhere" action - and why forgetting it is one of the most common privacy mistakes in federated systems.

OAuth 2.0: delegation (RFC 6749)

- A framework to allow users to **delegate access** to their resources on their behalf



OAuth 2.0 (RFC 6749) is a delegation framework: the resource owner (a user) authorizes a client (an application) to access a specific resource on the resource server, without handing the client the owner's own credentials. The client never sees the user's password - it receives a scoped, revocable access token instead.

The distinction that makes OAuth different from the authentication protocols earlier in this lecture: OAuth is about authorization, not authentication. It answers "may this client touch this resource?" - the user's identity is a precondition, handled by the authorization server, and the token carries the scope of what the client may do.

The figure on the slide is the whole model in miniature: the owner grants the client access to resource X on the resource server, and the client then presents its grant to retrieve the resource. The grant is the object that moves between the parties - and its lifetime and scope are the knobs that control the damage of a leak.

The delegation idea is easier to hold with a concrete case: a printing service that must read your photos from a storage service. Without delegation you would hand the printer your storage password - full account control, unrevokable, visible to a third party. With the framework, you authenticate to the authorization server, consent to a narrow grant (read-only, those folders, thirty days), and the printer receives a token that embodies exactly that consent - and nothing else. The framework's power and its danger live in the same place: a token is a bearer credential, so whoever holds it exercises the consent, which is why the scope, lifetime and audience of tokens are the three knobs every implementation must set carefully.

OAuth 2.0 players

- Resource owner
  - An entity capable of granting access to a **protected resource**
  - **End-user**: a resource owner that is a person
- Client
  - An **application** making requests for protected resources on behalf of the resource owner and with its authorization
- Resource Server
  - The server hosting protected resources
  - Responds to protected resource requests that have an **access token**
- Authorization Server
  - The server issuing access tokens to clients after successfully **authenticating resource owners** and obtaining their **authorization** for the clients to access one of their (users) resources

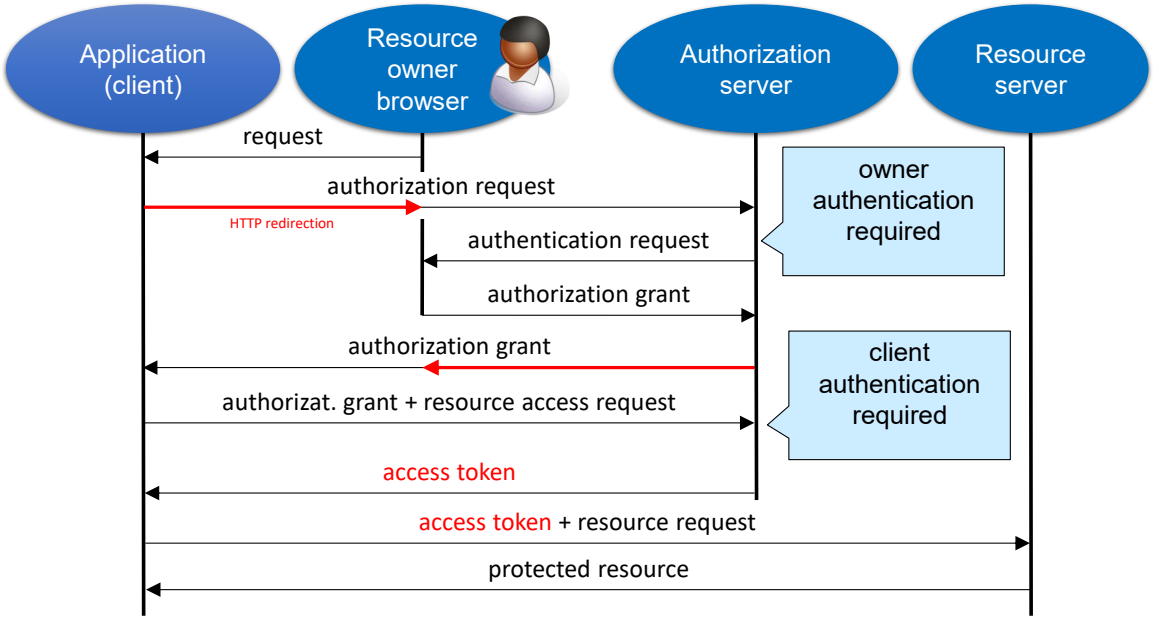
The four roles are the vocabulary of the RFC. The resource owner is the entity that can grant access to a resource - in practice, a person. The resource server hosts the protected resources and accepts access tokens. The client is the application acting on the owner's behalf, and the authorization server is the issuer: it authenticates the owner, obtains the owner's consent, and issues the tokens to the client. The end-user bullet is the practical note: in most deployments the resource owner is a person, and the person's authentication is the only place a real authentication protocol, password or passkey, appears in the whole OAuth flow.

The separation between resource server and authorization server is deliberate: large deployments run one authorization server for many resource servers, which is why the identity-provider pattern from earlier fits so naturally - in most stacks the IdP is the authorization server, and the resource servers are the relying parties.

The definitions matter because real-world stacks blur the roles: many products combine the authorization server and the resource server in one process, and some "clients" are themselves services (confidential clients in service-to-service flows). Knowing the roles lets you audit any concrete system against the model.

A quick way to check your grasp of the roles is to place a real system in the model: your photo-printing app is the client, the photo API is the resource server, the identity platform issuing tokens is the authorization server, and you are the resource owner. The model stays stable while products merge and split the roles: single-vendor stacks often collapse all four roles into two processes, and machine-to-machine deployments replace the human owner with the client itself (the client credentials grant). What never changes is which party holds which credential - and that mapping, not the product names, is what a security review must verify.

Protocol flow



The figure shows the full authorization-code flow, the standard interactive path. The user hits the application, which detects there is no token and redirects the browser to the authorization server; the owner authenticates and consents there; the authorization server redirects back with an authorization code; the client exchanges the code (out of band, server-to-server) for the access token; and only then does the client call the resource server with the token to fetch the protected resource. The flow has five messages and two authentication events, and the order is the security: the owner authenticates first, the consent is explicit, and the token is issued only after both, the sequence is what keeps the token away from anyone who has not been authorized.

The split between the in-band authorization code and the out-of-band token exchange is the security heart of the design: the code is short-lived and single-use, the token never passes through the browser, and the client can be authenticated during the exchange - which is why this flow is mandatory for public clients (mobile apps, single-page applications) in modern profiles.

Note the two distinct authentications in play, marked on the figure: the owner authenticating to the authorization server (a real authentication protocol - password, passkey, whatever the IdP uses) and the client authenticating to the authorization server during the token exchange (a machine authentication, classically a client secret, increasingly a signed request). Conflating the two is the most common OAuth design mistake.

OAuth 2.0 security best practices (RFC 9700)

- Use the authorization code flow with PKCE (RFC 7636)
  - For all clients, including confidential ones
- Do not use deprecated grants
  - Implicit and resource owner password credentials grants
- Limit the damage of leaked codes and tokens
  - Exact redirect URI matching; short-lived, narrowly scoped tokens
  - Sender-constrained tokens (DPoP, mTLS); refresh token rotation

RFC 9700 (2025) is the Best Current Practice for OAuth 2.0 security and the basis of the upcoming OAuth 2.1. Its central recommendation is to use the authorization code flow with PKCE for every client: the client sends a hash of a random secret with the authorization request and the secret itself when redeeming the code, so a stolen authorization code cannot be exchanged by anyone else.

Two grants from the original RFC 6749 are deprecated. The implicit grant returns the access token in the browser URL, where it leaks through history, logs and referrers. The resource owner password credentials grant gives the user's password to the client, which defeats the whole point of delegation.

The remaining measures limit the impact of leaks: redirect URIs must be matched exactly to stop open redirects that send codes to an attacker; access tokens should be short-lived and limited to the scopes needed; sender-constrained tokens (DPoP or mutual TLS) are bound to a key held by the client, so a stolen token cannot be used by another party; and refresh tokens should be rotated on each use.

The deprecated grants are worth knowing by name because they still appear in old integrations: implicit flows were built for pages that could not make server-side requests, and password grants for clients that predated tokens - both solve a short-term problem by creating a permanent one. PKCE, originally designed for mobile apps, turned out to protect confidential clients too, which is why the current guidance collapses every flow into one: authorization code plus PKCE, always. The sender-constrained tokens are the direction of travel: binding the token to a key the client holds turns "stolen token works anywhere" into "stolen token is useless without the key", the same upgrade that passkeys brought to user authentication.

OpenID Connect (OIDC)

- An identity (authentication) layer on top of OAuth 2.0
  - OAuth 2.0 only provides authorization;
  - OIDC adds an ID Token (signed JWT)

- The protected resources are identity attributes
  - Packed in **scopes**
  - The attributes are called (identity) **claims**

OpenID Connect is an identity layer on top of OAuth 2.0. OAuth gives you a token that says "you may access this resource"; OIDC adds a token that says "this user is who they claim to be". The protected resource in OIDC is the set of identity attributes about the user, and the issuer is an OpenID Provider speaking the same authorization-server machinery as before.

The vocabulary maps directly: the identity attributes are called claims, they are requested via scopes, and they arrive packed in a signed token (the ID token, a signed JWT) that the client can verify without calling back to the IdP. The "sign in with Google" button is OIDC: the redirect to the IdP is the OAuth flow, and the payload that comes back is identity, not just access.

The practical consequence is that a single protocol now carries both authentication and authorization to web and mobile applications, which is why OIDC has become the default for anything that needs "log in" in a federated world - and why the distinction between the two jobs, kept carefully separate all through this lecture, is finally collapsed into one standard.

The layering is the design lesson: the delegation framework defines how a client obtains tokens and what they authorize, and this protocol defines one specific protected resource - the user's identity - plus the token format (a signed ID token) that carries it. The division also creates the classic integration pitfall: receiving an access token proves nothing about who the user is, only that someone authorized the request, and libraries that treat any token as a login are the source of real vulnerabilities. The ID token, validated against the issuer's keys, its audience and its expiry, is the actual authentication result - and that distinction, authorization token versus identity assertion, is the last piece the lecture leaves you with: one framework, two tokens, two different questions answered.