Identification

Sicurezza (CT0539)   2025-26
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Introduction

Identification is the task of correctly identifying a user or entity

It is typically **required** for enforcing other security properties

Any time the **access to a resource** needs to be regulated, some form of identification is necessary

Examples:

- Users identify into a system when they **login**
- Users identify to mobile network providers through the **SIM card**
- Users identify to the SIM card through a **PIN**
- Users identify to **ATMs** with cards and PINs

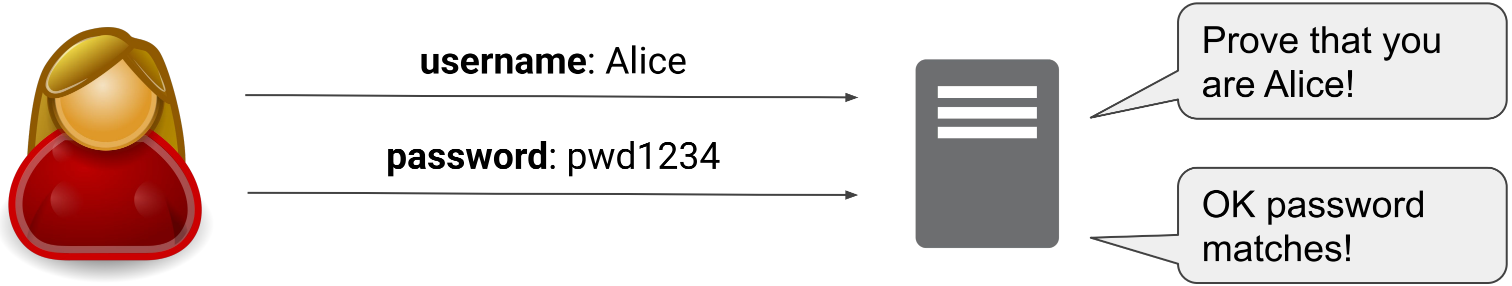
Entity authentication

Identification can be thought as **authenticating a user** or, more generally, an **entity**

- Allow a **verifier** to check **claimant's** identity

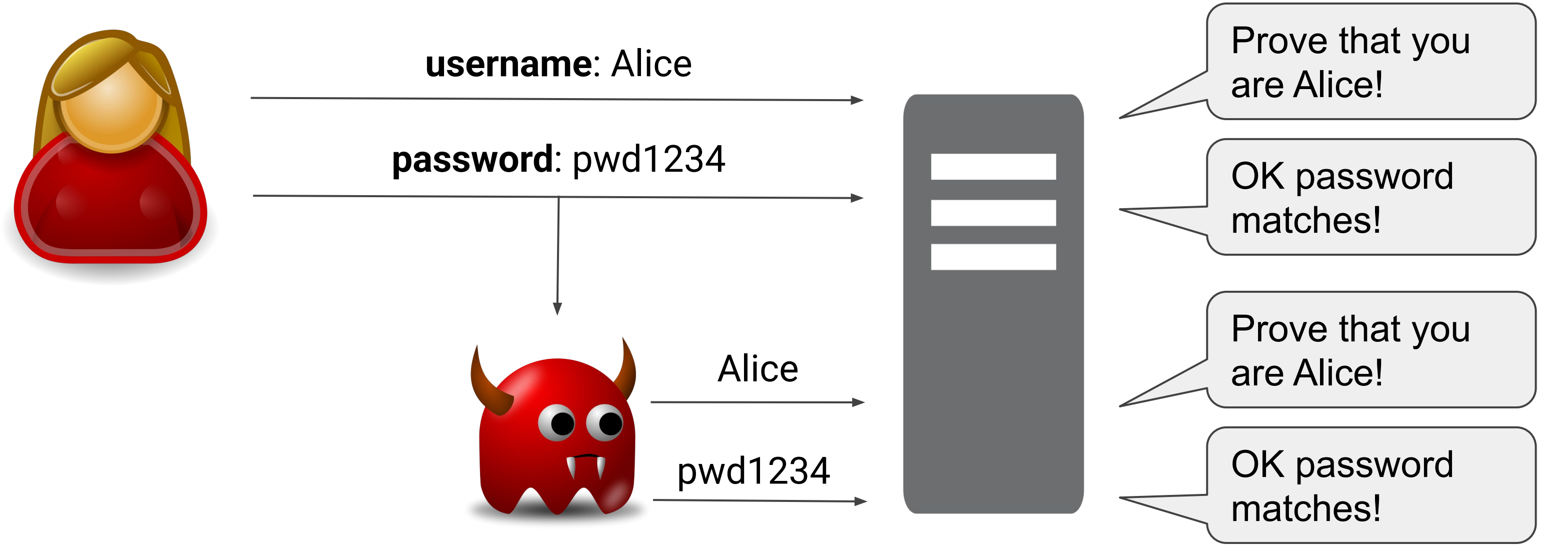
Example: login-password scheme

- The user **claims** her identity by inserting the **username**
- The system **verifies** the identity by asking for a **secret password**



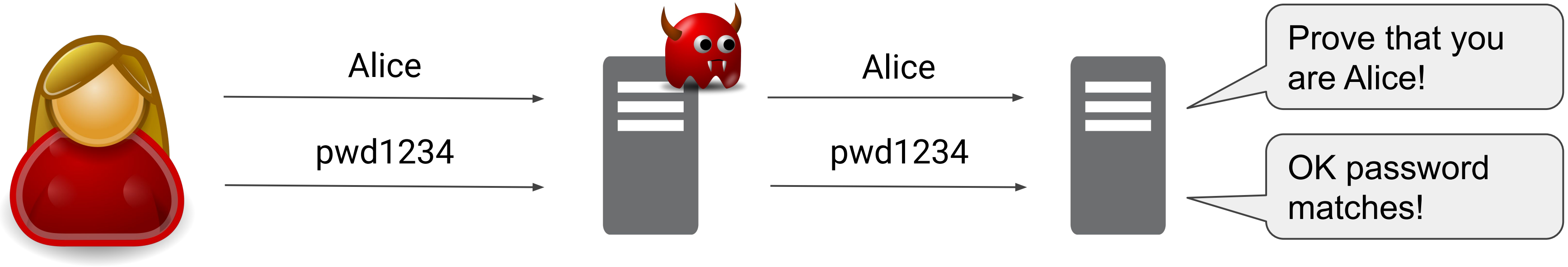
Impersonation

An identification scheme should prevent **impersonation**, even observing previous identifications



Transferability

The verifier should not **reuse** a previous identification to impersonate the claimant with a different verifier, unless **authorized**



NOTE: The verifier has more information available than an attacker, e.g., when the communication is **encrypted**

⇒ Passwords <u>shouldn't be reused</u>!

Classes of identification schemes

Something known. Check the **knowledge** of a secret

- passwords, passphrases, Personal Identification Numbers (PINs), cryptographic keys

Something possessed. Check the **possession** of a device

- ATM cards, credit cards, smartcards, One Time Password (OTP) generators, USB crypto-tokens, smartphones, …

Something inherent. Check **biometric** features of users

- Paper signatures, fingerprints, voice and face recognition, retinal patterns

Passwords

The identity claimed through the **login** information is checked by asking for a corresponding **secret password**

Problem 1: What if the password is *sniffed*?

⇒ stolen passwords allow for **impersonation** (*weak authentication*: secret is exhibited)

Problem 2: What if password is *guessed*?

⇒ guessed passwords allow for **impersonation**

Problem 3: How are password **stored** on the server?

⇒ an attacker getting into the server might steal all the passwords (might be reused for other servers)

Preventing leakage and guess

Problem 1: What if the password is *sniffed*?

Solution: only use password over encrypted channels

Example 1: passwords and card numbers sent over **`https`**

Example 2: `telnet` was an **insecure** remote terminal client sending passwords in the clear

Problem 2: What if password is *guessed*?

Solution 1: Disable the service after MAX attempts

Example: lock SIM after 3 attempts

Solution 2: Use strong passwords

⇒ useful in offline attacks when the service cannot be disabled

“Encrypted” passwords

Problem 3: How are password **stored** on the server?

IDEA: The server stores a *one-way hash* of passwords

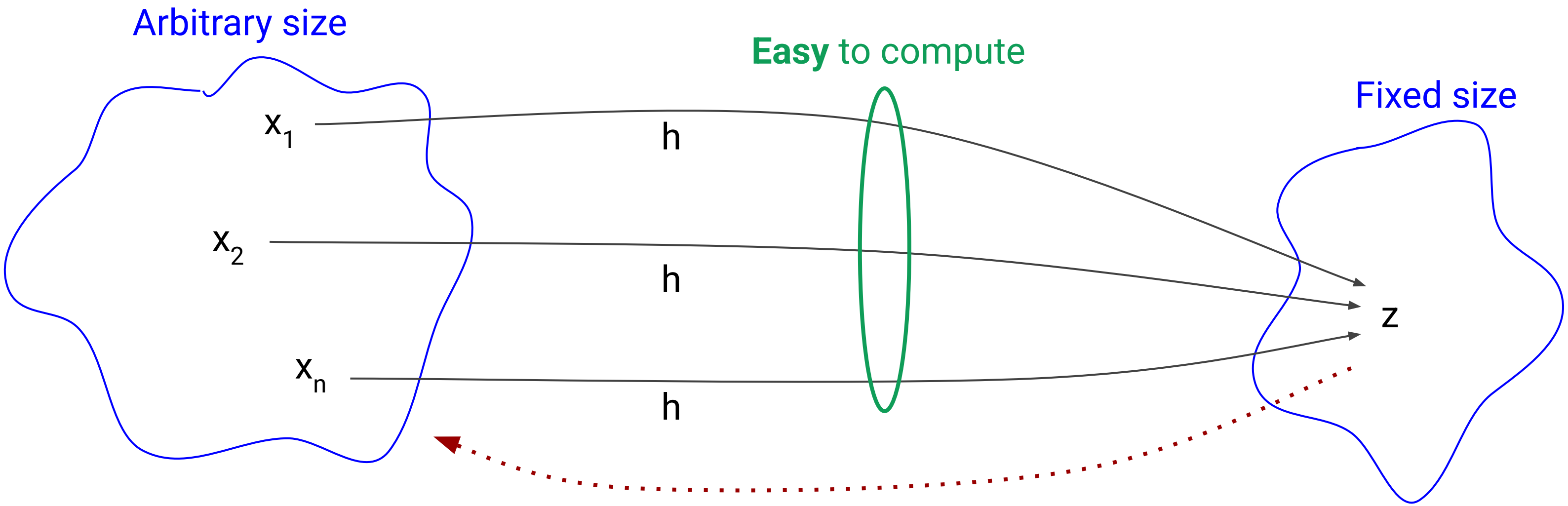
Definition (***hash function***). A hash function h computes efficiently a **fixed length** value h(x)=z called **digest**, from an x of **arbitrary size**.

NOTE: **Collisions** are possible: h(x1) = h(x2)

Definition (***one-way hash function***). A hash function h is **one-way** if given a digest z, it is ***infeasible* to compute a preimage** x’ such that h(x’)=z

⇒ **Finding** a pre-image is computationally infeasible

One-way hash function



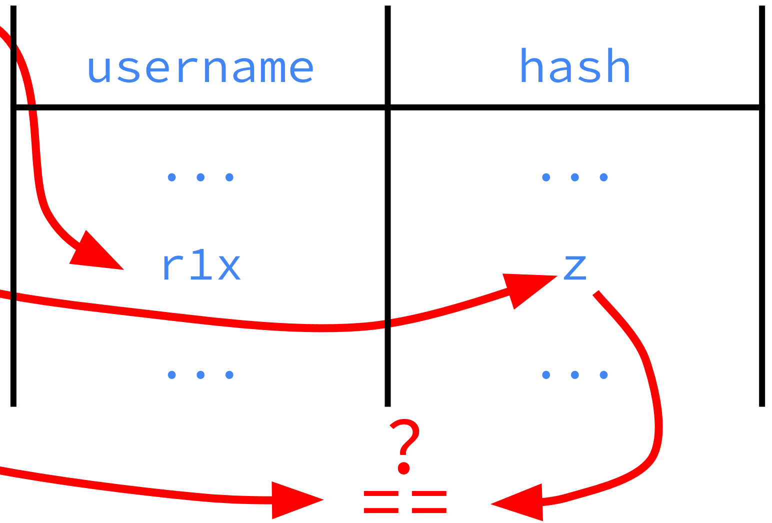
Hard to invert: computing any of
$x_1, x_2, \ldots, x_n$ is **infeasible**

Verification of hashed passwords

User is asked for **username**, **pwd**

The system retrieves the stored hash **z** of the password for that username

The system computes **h(pwd)** and checks it is the same as z



⇒ Since h is one-way, in principle, **no password can be recovered from its hash z**

Standard one-way hash functions

MD5 (Message-Digest algorithm 5) produces 128-bit (16-byte) hash

SHA-1 (Secure Hash Algorithm 1) produces a 160-bit (20-byte) hash

Collision attacks: it is possible to find collisions in MD5 and SHA-1: finding x1 and x2 such that h(x1) = h(x2)

⇒ No efficient attack to compute a valid preimage (still **one-way**!)

SHA-2 (Secure Hash Algorithm 2) produces 224, 256, 384 or 512 bits hashes (28, 32, 48, 64 bytes)

SHA-3 (Secure Hash Algorithm 3) is the result of a NIST competition to establish the new cryptographic hash function standard

SHA-2 is the **most used one**, no reason to switch to SHA-3 yet ...

Examples

Dash '-' stands for stdin (see next slide)

```
$ echo -n "mypassword" | md5sum
34819d7beeabb9260a5c854bc85b3e44  -

$ echo -n "mypassword" | sha1sum
91dfd9ddb4198affc5c194cd8ce6d338fde470e2  -

$ echo -n "mypassword" | sha224sum
9b1cdbab8c8410d63ca8700b12d03b9f0bf93d33b793653cc0983ef3  -

$ echo -n "mypassword" | sha256sum
89e01536ac207279409d4de1e5253e01f4a1769e696db0d6062ca9b8f56767c8  -

$ echo -n "mypassword" | sha384sum
95b2d3b2ad7c2759bf3daa53424e2a472bc932798dae30b982621833a449492883b7ae9d31d30d32372f98abdbb256ae  -

$ echo -n "mypassword" | sha512sum
a336f671080fbf4f2a230f313560ddf0d0c12dfcf1741e49e8722a234673037dc493caa8d291d8025f71089d63cea809cc8
ae53e5b17054806837dbe4099c4ca  -
```

File integrity (never use MD5 and SHA-1)

```
$ sha256sum Assembly/*
23b21ab11641c6bfc3ec3599bcc85a61414fa9b8316002112ff164231efc0fea  Assembly/checkPassword
6ad802b2b45b229abffdf1433df949b526db07b10543b0bd38c56deb65d34820  Assembly/count
034e1535a391e2a3cdf404fc144e124af457155a5a3d2782b122c5d1dae8be2a  Assembly/count.c
$ sha256sum Assembly/* > checksum
```

Digest are computed and stored in checksum

```
$ sha256sum -c checksum
Assembly/checkPassword: OK
Assembly/count: OK
Assembly/count.c: OK
```

Hashes are recomputed and compared with the ones in file checksum

```
$ nano Assembly/count.c

$ sha256sum -c checksum
Assembly/checkPassword: OK
Assembly/count: OK
Assembly/count.c: FAILED
sha256sum: WARNING: 1 computed checksum did NOT match
```

Any modification is detected! Note that for **MD5** and **SHA-1** it is possible to find collisions so **NEVER** use them for file integrity!

Offline attacks

Attacker model: we assume the attacker has access to the password file and tries to recover passwords from their hashes

⇒ **offline** attack!

One-way hashes protect passwords stored on the server, but ...

Problem 2: What if password is *guessed*?

Solution 1 was: disable the service after MAX attempts

With the password file, the attacker can try **unlimited** password hashes offline

⇒ <u>useless</u> for offline attacks!

Solution 2: use strong passwords

⇒ protects from offline attacks

Dictionary attacks

Brute force: even if one-way hashes cannot be inverted, an attacker can try to compute hashes of *easy passwords* and see if the hashes match

Note: It is possible to **precompute** the hashes of a dictionary and just search for z into it

Example:

```
$ echo -n "mypassword" | sha256sum
89e01536ac207279409d4de1e5253e01f4a
1769e696db0d6062ca9b8f56767c8  -
```

Password `"mypassword"` is clearly weak, we can search for the hash directly in search engines or using existing [online services]()

Salting passwords

Precomputation of password hashes is prevented by adding a ***random salt***, different for each user, which is stored together with the hashes

username	hash	**salt**
...	...	**...**
r1x	z	**s**
...	...	**...**

Verification of “salted” passwords

User is asked for **`username`**, **`pwd`**

The system retrieves the stored hash **`z`** of the password for that username

The system retrieves the stored salt **`s`**

The system computes **`h(pwd,s)`** and checks if it is the same as z

`username`	`hash`	`salt`
`...`	`...`	`...`
`rlx`	`z`	`s`
`...`	`...`	`...`

$\stackrel{?}{==}$

The salt s is **different for each user** and is **stored** in the password file

⇒ Precomputing hashes for each possible salt would require **too much space**

Example

```
$ echo -n "mypassword54otdf84" | sha256sum
3181527671d5dd6b3c1a990ed7b47f3afd69bdfa7794757451639f2b4aa7d65e
```

Password `"mypassword"` is clearly weak

We add “random” salt `"54otdf84"`

Searching for the hash directly in search engines or using existing online services will fail!

⇒ since salt is stored in the file, an attacker can still **bruteforce** easy passwords computing, on-the-fly, the hashes (slower but feasible!)

“Slow” hashes

Instead of using a single hash, hashes are usually iterated so to slow down brute-force

Example: Linux passwords

```
goofy:$6$Lc5mF7Mm$03IT.AXVhC3Vl4/rLAdomffgv5feOlKBzNGtpEei
2dBgK9z/4QBqM3ZMRK4qcbbYJhkAE.2KscEZx0Am/y50: .....
```

- `6`: SHA512-based hashing, iterated **5000** times, by default
- `Lc5mF7Mm`: salt
- `03IT.AXVhC3...Zx0Am/y50`: digest

Example ctd.

Linux passwords in python:

```
>>> import crypt
>>> crypt.crypt("donald","$6$Lc5mF7Mm$")
'$6$Lc5mF7Mm$03IT.AXVhC3Vl4/rLAdomffgv5feOlKBzNGtpEei2dBgK9z8B/4QB
qM3ZMRK4qcbbYJhkAE.2KscEZx0Am/y50'
```

Command line tool (provided by `whois` package in ubuntu):

```
$ mkpasswd donald -m sha-512 -S Lc5mF7Mm
$6$Lc5mF7Mm$03IT.AXVhC3Vl4/rLAdomffgv5feOlKBzNGtpEei2dBgK9z8B/4QBq
M3ZMRK4qcbbYJhkAE.2KscEZx0Am/y50
```

Increasing the iterations

```
$ time mkpasswd donald -m sha-512 -S Lc5mF7Mm
$6$Lc5mF7Mm$03IT.AXVhC3Vl4/rLAdomffgv5feOlKBzNGtpEei2dBgK9z8B/4QBqM3ZMRK4qc
bbYJhkAE.2KscEZx0Am/y50
real     0m0.005s
user     0m0.003s
sys 0m0.002s

$ time mkpasswd donald -m sha-512 -S Lc5mF7Mm -R 5000000
$6$rounds=5000000$Lc5mF7Mm$FWm/GeTLTryHa0Nt/WfrbLqjVOsipSBNP3IUgwbNP7H95eR8
lhKj.6Pc7YcznupXjHXA9QBirkmmaxh3oqt4v.

real     0m1.926s
user     0m1.925s
sys 0m0.001s
```

Default number of iterations is 5000

We raise the number of iterations to 5000000

Salt examples

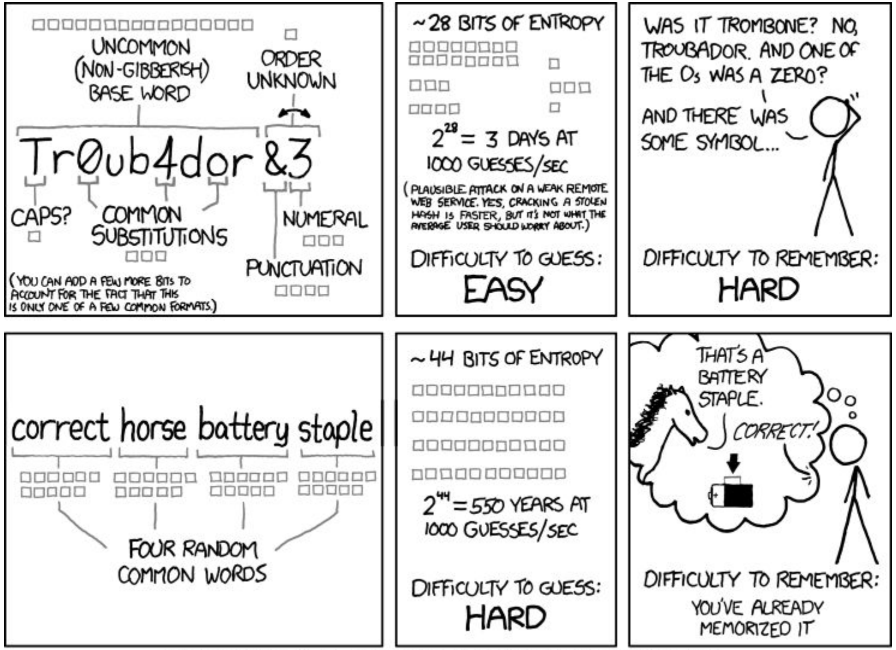
Up to 16 random chars from [a-zA-Z0-9./]

```
$ mkpasswd donald -m sha-512
$6$XGX3asxc$srRtplHNT0Itr44D/xyYbxBNQoPPsYYb6gVNxP372PL0hw9Toit9DQ
KVMtg9/I9DR9UGaZF1sCclcYRscJgDm1

$ mkpasswd donald -m sha-512
$6$zLm12FS6w/Dr$LBUDF9J.uneghlepBGi.OGrWJ9NCdzro5O.j8iq3gJQLt7A2mj
WavWYw7PkISKYHdy63pVI9zLDmkXU2L2Vex.

$ mkpasswd donald -m sha-512
$6$uTOR38Mo16$PLjldovzZAuu6eRVZtbL2HwUeB.VIQ.hQiwhmxmnggDy5EZZufKK
CjrMbXS3rM.2S6oKWK.aEoVFtAFsPJaPP0
```

Password policies

NIST SP 800-63-2 suggests the following alternative rules:

- Password must have at least sixteen characters (**basic16**)
- Password must have at least eight characters including an uppercase and lowercase letter, a symbol, and a digit. It may not contain a dictionary word (**comprehensive8**)



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

xkcd.com

Diceware

Passphrase of N words picked at random from a fixed list, by rolling 5 dice

- 5 dice gives $6^5 = 7776$ possible words
- Entropy for each word is $\log_2 7776 \sim 12.9$ bits

The **whole entropy** is thus 12.9 N

- for N=4 entropy is ~**52** bits
- for N=5 entropy is ~**64** bits
- for N=6 entropy is ~**77** bits

Word list: http://world.std.com/~reinhold/dicewarewordlist.pdf

Token-based authentication

Something possessed. Check the **possession** of a device

- ATM cards, credit cards, smartcards, One Time Password (OTP) generators, USB crypto-tokens

Memory cards

Passive card with a memory

Examples:

- Old ATM cards with magnetic stripe
- Hotel cards to open doors

When **paired with a PIN** the attacker needs to steal/duplicate both

Problems:

- Passive cards are usually simple to clone

Example:

- Old ATM cards were cloned by putting a fake reader and a camera (to also steal the PIN)

Smart cards

Smart token with embedded chip

Various devices:

- Standard smartcard
- USB token
- Small portable objects
- Bigger objects with display and/or keyboard

⇒ **One time passwords** (OTPs) and **Challenge-response**

Biometrics

Something inherent. Check **biometric** features of users

- Signatures, fingerprints, voice, face, hand geometry, retinal patterns, iris, ...

Biometrics

1. **Enrollment**: features are extracted and stored in database
2. **Verification**: features are extracted and compared with the stored ones

A delicate balance:

No impersonation (no false positives) but correct user should be identified most of the times (no false negatives)

Problems:

A breach in the biometric database has **high impact**:

- biometric data is unique, belongs to users
- differently from passwords it cannot be changed if leaked

New attacks: *adversarial machine learning*