# User Authentication 1

Security 1 (CM0475, CM0493) 2020-21
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

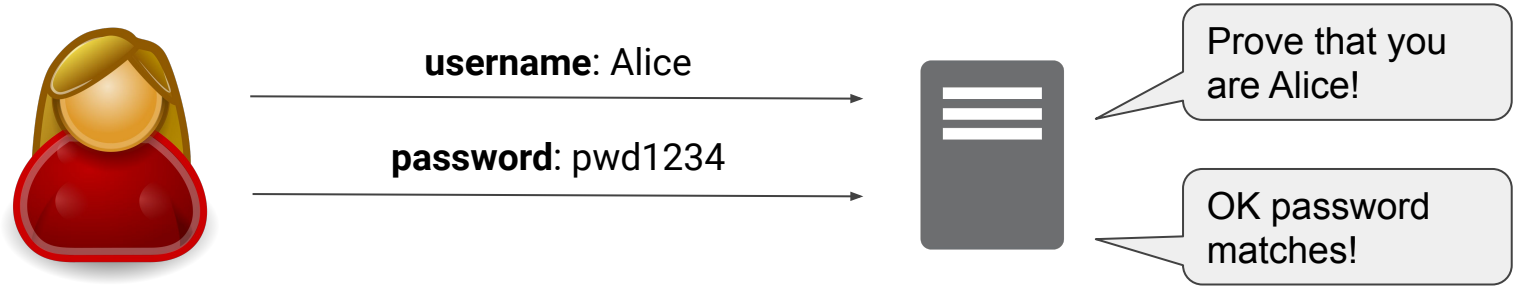
# Identification == entity authentication

Identification can be though 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**



# Properties

An identification scheme should always prevent:

**Impersonation**, even observing previous identifications

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

- The verifier has more information available than an attacker, e.g., when the communication is encrypted
- **Example**: same password for different web sites!

# 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

**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**.

**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

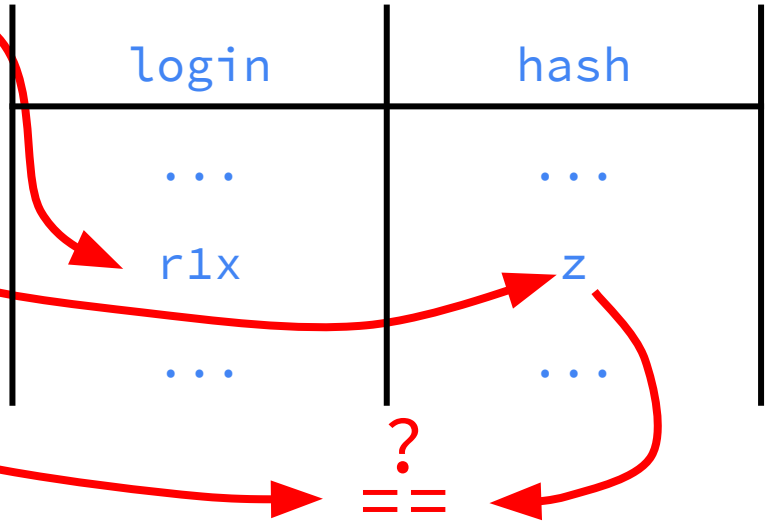
⇒ **Finding** a pre-image is computationally infeasible

# Verification of hashed passwords

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

The system retrieves the stored hash **`z`** of the password for the given login

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**

# 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*

| login | passwd | salt |
|---|---|---|
| ... | ... | ... |
| r1x | z | s |
| ... | ... | ... |

`h(pwd,s) ?== z`

# “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

# Rainbow tables

Suppose we want to precompute hashes for a huge set of passwords (not just words in a dictionary)

- Storage and searching becomes problematic

Rainbow tables are a technique that allows for a **time/space tradeoff**

- Chains from a password **p** to a final hash **z**
- **p** is hashed and then "reduced" to **p'**
- **$p \rightarrow h(p) \rightarrow p' \rightarrow h(p') \rightarrow \ldots p_f \rightarrow h(p_f) = z$**

Reduction is *any function* returning a candidate pwd

# A simple example

```
p = pwd
for (i in [0,C_len-1]):
    print(p)
    h = hash(p)
    print(h)
    p = red(h)
```

`hash` is sha256

`red` takes the first 8 bytes and makes them “printable”

# Simple example

**donald**
4138cfbc5d36f31e8ae09ef4044bb88c0c9c6f289a6a1c27b335a99d1d8dc86f
...
6bI!l%"d
c880c7f068e2b4fe6ec76fea6756d8b1ee92b0d96d0b867be3b952a3ac75cf96
k#j6h(WD
75532eec682a5c65f5a6f8717afc00f67f2518f8bd251865374447cb6bc50725
uS.2h*\e
9d384a0c159b257534258b255023062cbf560491de12ca79ddffca052a5b67b5
@8Jir>%u
6b16a5147f320f182d8d55ed5631203cede6fde5292ba3bd697cb430c2102d22
ksHq"2lu
**25f94e180a5abcf4c4c70ab68fc2c6365dee0778e86652fdef8ddeab60d939d2**

# Searching rainbow tables

Suppose we have **n chains** of **length `C_len`**

$(p_1, h_1)$ $(p_2, h_2)$ … $(p_n, h_n)$

and we want to invert h

We proceed as follows:

```
r=h, i=0
while (r not in {h1,h2, …, hn} and i < C_len):
    r = hash(red(r))
    i++
```

# If h is in the chain we find it!

**donald**
4138cfbc5d36f31e8ae09ef4044bb88c0c9c6f289a6a1c27b335a99d1d8dc86f
...
6bI!l%"d
**c880c7f068e2b4fe6ec76fea6756d8b1ee92b0d96d0b867be3b952a3ac75cf96**
k#j6h(WD

75532eec682a5c65f5a6f8717afc00f67f2518f8bd251865374447cb6bc50725
uS.2h*\e

9d384a0c159b257534258b255023062cbf560491de12ca79ddffca052a5b67b5
@8Jir>%u

6b16a5147f320f182d8d55ed5631203cede6fde5292ba3bd697cb430c2102d22
ksHq"2lu

**25f94e180a5abcf4c4c70ab68fc2c6365dee0778e86652fdef8ddeab60d939d2**

# Inverting the hash

If we find the hash after k steps we do

```
r = p  // the password of the matching chain
for C_len - k - 1 steps:
    r = red(hash(r))
return r
```

# Inverting the hash

```
adonald
4138cfbc5d36f31e8ae09ef4044bb88c0c9c6f289a6a1c27b335a99d1d8dc86f
A8r_]69{
b6993563cc9fb06b68bc8766b2b556a179557bfb306daade3f032dcf208e9865
Y<5coBSk
1af94c530693bd80abb1bd9eca143324eb3185fbf559634167ece0aa494fd2a1
w?LSc6`#
e5138aee690f1ec23e4fbee436138c51b955b3438a96be23188a7277f1554530
+p-4il{e
93bfa6db6c82dcc1bdf6c9de7682f236817f2e4b25907f7934b0d8d8c28b3107
6bI!l%"d
c880c7f068e2b4fe6ec76fea6756d8b1ee92b0d96d0b867be3b952a3ac75cf96
k#j6h(WD
75532eec682a5c65f5a6f8717afc00f67f2518f8bd251865374447cb6bc50725
uS.2h*\e
9d384a0c159b257534258b255023062cbf560491de12ca79ddffca052a5b67b5
@8Jir>%u
6b16a5147f320f182d8d55ed5631203cede6fde5292ba3bd697cb430c2102d22
ksHq"2lu
25f94e180a5abcf4c4c70ab68fc2c6365dee0778e86652fdef8ddeab60d939d2
```

C_len - 4 - 1 = 10 - 5 = 5

k = 4

# Merging chains and space/time tradeoff

Chains can merge, in this case we **lose coverage:** after two chains merge, next hashes will **overlap**

**IDEA**: Make **red$_i$** depend on **step i**

⇒ if two chains merge they will split, unless they merge at the very same step!

This is where the name "**Rainbow**" comes from!

P is the set of passwords that we want to cover (assume no collisions)

⇒ We need about |P| / C_len chains (**space decreases** if we increase the chain length)

⇒ Searching **time** is proportional to C_len$^2$ (notice that with red$_i$ we cannot reuse `red(hash(r))` from previous steps)