

User Authentication

System Security (CM0625, CM0631) 2026-27
Università Ca' Foscari Venezia

Riccardo Focardi

<https://www.unive.it/people/focardi>
seclab.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

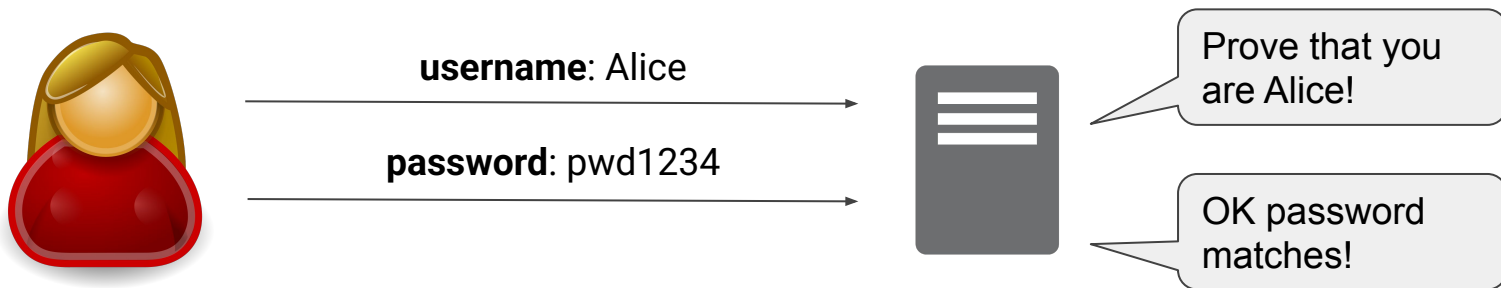
Identification == entity authentication

Identification can be 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**



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

Password-based authentication

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

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

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

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	hash	salt
...	...	...
r1x	z	s
...	...	...

$$h(\text{pwd}, s) \stackrel{?}{=} z$$

“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/rLAdomffgv5fe0lKBzNGtpEei**
2dBgK9z/4QBqM3ZMRK4qcbYJhkAE.2KscEZx0Am/y50 :

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

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 an **embedded chip**

Various devices:

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



Smart card interface and protocol

Interface:

- **Contact:** a conductive contact plate on the surface of the card (typically gold plated) for transmission of commands, data, and card status
- **Contactless:** Both the reader and the card have an antenna, and communicate using radio frequencies

Protocol:

1. **Static:** token provides a fixed secret (as for passive cards)
2. **One time password (OTP):** the token generates a fresh OTP that is used for authentication
3. **Challenge-response:** a challenge is processed by the token that produces a response (e.g. digitally signed)

One Time Passwords (OTP)

Once a secret is leaked it can be used to authenticate many times:

- sniffed password
- cracked password hash
- cloned passive token

One Time Passwords (OTPs) are never reused

They mitigate password leakage/crack by allowing for a single authentication (es. bank OTPs)

⇒ The token and the computer system must be kept **synchronized** so the computer knows the OTP that is current for this token.

OTP security

What is the **threat model**?

- An attacker might intercept some OTP values, which however cannot be reused
- This is why OTPs are useful ...
- **NOTE:** OTP values are **related**, as they come from the same generator

What **security property**?

- An attacker shouldn't be able to **forge** the current OTP value v_n ...
- ... even knowing **previous** OTP values $v_1, v_2, \dots, v_{n-1}$

Lamport's hash-based OTP

Given a secret s and a one-way hash function h we compute:

$$h^t(s) \quad \text{which is: } h(h(\dots h(s)\dots)) \quad t \text{ times}$$

We let the Claimant and the Verifier share this value

- The Claimant uses the list of passwords:
 $h^{t-1}(s), h^{t-2}(s), \dots, h(s), s$
- The Verifier computes $h(pwd)$ and checks if it is equal to the stored hash:
 $h(h^{t-1}(s)) == h^t(s)$
- If the check succeeds the Verifier stores $h^{t-1}(s)$

Lamport's hash-based OTP

passwords: $h^{t-1}(s)$ $h^{t-2}(s)$... $h(s)$ s

stored hashes: $h^t(s)$ $h^{t-1}(s)$... $h^2(s)$ $h(s)$

Limitation: Only t authentications are possible

Security: Computing next passwords from the current is equivalent to compute the preimage of h , which is **infeasible** (h is one-way)

⇒ More secure than storing a shared secret “seed” used to generate the OTP like in $h(\text{seed}, \text{counter})$, also known as HOTP (HMAC-Based OTP)

Case study 1: RSA seed breach

RSA SecurID Breach (March 2011)

- The values of secret “seeds” were stored insecurely and have been leaked through phishing
- ⇒ 40M of devices replaced, big companies attacked, huge image damage for RSA

