Information systems security

Lorenzo Di Maio

October 2024

Contents

Chapter 1

Authentication techiniques protocols, and architecures

Authentication refers to the process of verifying the identity of an entity (whether it's a human, software component, or hardware element) before granting access to resources in a system. Authentication can be applied to various type of "actors", such as:

- **Human being**
- **software component**
- **Hardware element**

Authentication vs Authorization

- **Authentication (authC/authN)**: established the indetity of an entity.
- **Authorization (authZ)**: determines where a authenticated entity has permission to access.

1.1 Authentication factors

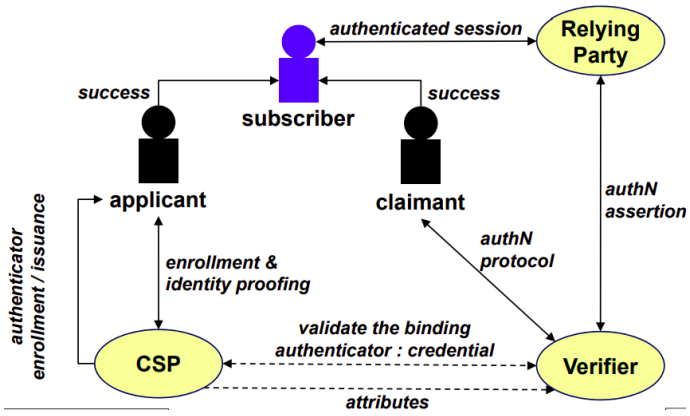
Authentication can be based on 3 primary factors:

- **Knowledge**: Information that only the user knows and can provides as proof of ther identity.
- **Ownership**: Physical object or device that only the user has access to.
- **Inherence**: This factor relies on onique biological traits of the user (e.g fingerprint).

N.B. Authentication can be applied not just to human user, but also to processes and devices.

1.1.1 Risks

- **Knowledge:**
  - Storage → if passwords are stored improperly, they are vulnerable to thieft.
  - Demonstration → user might inadvertently reveal their password through social engineering.
  - Transmission → if passwords are sent over ensecured channel, they can be intercepted by attackers.
- **Ownership:**
  - Authentication thieft
  - Cloning
  - Unathorized usage
- **Inherence:**
  - Counterfeiting → biometric data can be spoofed or replicated by attackers using sophisticated techiniques.
  - Privacy → the use of biometric data riases the risk of biometric information being exposed.
  - Irreversibility → biometric traits cannot be raplaced if compromised.



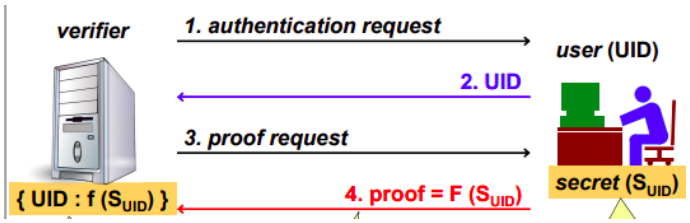
1.2 Digital Authentication model (NIST SP800.63B)

Entities:

- **Subscriber**: appliacant who has succesfully completed identity proofing.
- **Appliacant**: an indiviual applying to established a digital identity.
- **Claimant**: the user trying to prove their identity to access a system or service.
- **Relying Party**: will request/receive an authN assertion from the verifier to asses user identity (and attributes).
- **Verifier**: validates the user's credential during each authentication event.
- **CSP**:
  - Verifies the applicant's indetity during the initial enrollment process.
  - Issue a credential and bines it to an authenticator for the user.

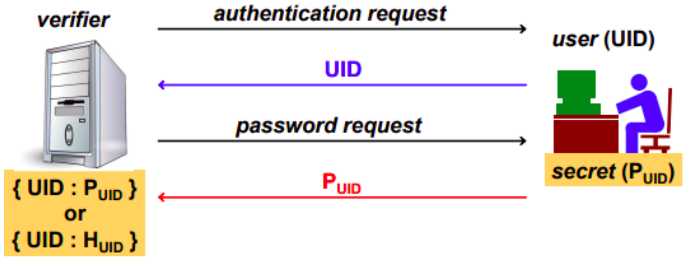
1.3 Generic authentication protocol

1. The user initiates an authentication request by sending their UID.
2. The user generates a proof based on their secret, useing a secure function $F(S_{UID})$, and send this proof to the verifier.
3. The verifier checks if the received proof matches the stored representation of the secret.
4. If it matches, the user is succesfully authenticated.



1.4 Password base authentication

1. The user sends their UID and P_{UID} (= Password) to the verifier.
2. The server verifies the proof:
   - If password are stored in cleartext, it directly compares the proof with the stored password.
   - If password are stored in hashes, it hasesh the proof and compares it to the store hash H_{UID}.



Problems of reusable Passowrds

- **PWD Sniffing** (attackers intercept password during transmission)
- **PWD Database attack** (if DB contains plaintext or obfuscated PWD)
- **PWD Guessing** (very dangerous if it can be done offline, e.g against a list of PWD hashes)
- **PWD Enumeration** (PWD brute force attack)
  - If PWD is limited in lenght and/or character type.
  - If authN protocol does not block repeated failures.
- **PWD Duplications** (using the same PWD for one service against another one, due to user PWD reuse)
- **Crypographic Aging** (as computing power grows, older crypographic methods become vulnerable to new attacks)
- **PWD capture via server spoofing and phishing** (attackers deceive user into givining away their PWD by pretending to be legitimate service)

Password best practies

Suggestion to reduce password risks:

- Use alphabetical characters (upper case + lower case), digits and special characters
- Make passwords long (at least 8 character)
- Never use dictionary words
- Change password regularly, but not too frequently
- Do not reuse passwords across different services

Password storage

- **Server Side:**
  - Passwords should never be stored in cleartext.
  - Encrypted passwords aren't ideal since the server would need to know the encryption key.
  - Better to store a password digest (hashed password), though vulnerable to dictionary attacks.
  - Rainbow tables can speed up these attacks, so it's important to add a "salt" (random variation) to each password.
- **Client-side:**
  - Ideally, passwords are memorized by the user, but having many passwords makes this difficult.
  - People may resort to writing them down or using simple passwords, which is risky.
  - Using a password manager or encrypted file is a safer alternative.

1.5 The "dictionary" attack

- **Hypothesis:** The attacker knows the hash algorithm and the hashed password values.
- **Pre-computation:** For each word in a dictionary, compute and store its hash $store(DB, Word, hash(World))$
- **Attack process:**
  - Let HP (=hash password) to be the hash of an unknown password.
  - Lookup HP in the precomputed dictionary (DB) to find a matching password.
  - If found, output the password; if not, indicate it's "not in dictionary".

1.6 Rainbow Table attack

Rainbow Table is a **space-time trade-off technique** that reduces storage needs for exhaustive hash tables, making certain brute-force attacks feasible within limited space. It uses a reduction function $r : h \to p$ (which is NOT h^{-1}) to generate chains of hashes.
Example:

- For a 12-digit password, an exhaustive hash table woudl require $10^{12} rows(P_i : HP_i)$
- rainbow = 10^9 rows, each representing 1000 possible passwords.

Attack

```
for (k=HP, n=0; n<1000; n++)
 ■ p = r(k)
 ■ if lookup( DB, x, p ) then exit ( "chain found, rooted at x" )
 ■ k = h(p)
exit ( "HP is not in any chain of mine" )
```

1.7 Salting Passwords: A Defense Against Dictionary and Rainbow Table Attacks

Salting passwords is a security technique used to protect stored passwords from dictionary attacks and rainbow table attacks. A salt is a unique, random string added to each password before hashing. This ensures that even if two users have the same password, their hashes will be different due to the unique salt.

Steps for each user (UID):

- Generate or ask for the user's password.
- Create a unique, random slat for each user.
- Compute the salted hash:$SHP = hash(password \mid\mid salt)$
- Store the triplet $\{UID,\ SHP,\ salt\}$

Password Verification with Salt

- **Claimant:** Provides their user ID (UID) and password (PWD).
- **verifier:**
  - Uses the UID to find the stored salted hash (SHP) and salt.
  - Computues $SHP' = hash(PWD \mid\mid salt)$.

The Linkedin attack

In 2012, LinkedIn was breached, exposing 6.5 million unsalted SHA-1 password hashes. The lack of salting allowed attackers to crack at least 236,578 passwords through crowdsourced efforts before restrictions halted the exposure.

1.8 Strong authentication definitions

The concept of strong authentication (authN) is crucial in ensuring secure identity verification, but it has never been formally defined with a universal definition. Various definitions exist depending on the context, such as the European Central Bank (ECB) and PCI-DSS.

ECB definition

The ECB defines strong authentication as a process that involves at least two independent elements from **knowledge** (e.g. password), **ownership** (e.g. smartcard), and **inherence** (e.g. biometrics). The key requirement is that these elements must be mutually independent, so compromising one should not affect the others. Furthermore, at least one element should be **non-reusable** or **non-replicable** (except for inherence), with the entire process safeguarding the confidentiality of the authentication data.

PCI-DSS Definition

PCI-DSS mandates **multi-factor authentication (MFA)** for access to cardholder data, particularly for administrators and remote access from untrusted networks. Since version 3.2, MFA has become compulsory for remote access, and the use of the same factor twice (e.g., two passwords) does not qualify as MFA.

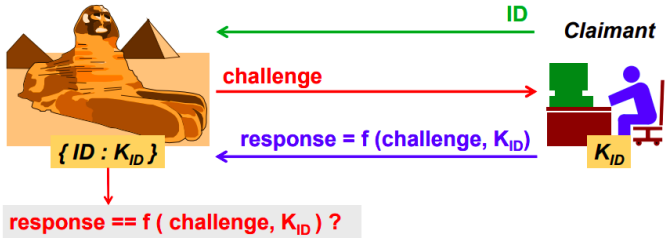
1.9 Challenge-Response Authentication (CRA)

Challenge-response authentication (CRA) is a widely used technique where a challenge is issued, and the claimant responds by solving it with a secret (shared or private). The challenge must be **non-repeatable** (usually a random nonce) to avoid replay attacks. The function used to compute the response must be **non-invertible**, otherwise, a listener can record the traffic and easily find the shared secret:

$$if(\exists f^{-1})\ then\ K_c\ =\ f^{-1}(response, challenge)$$

1.9.1 Symmetric CRA

Symmetric CRA involves a shared secret (like a password or key) between the claimant and verifier. This method is fast, often utilizing hash functions (e.g., SHA1, SHA2, SHA3).



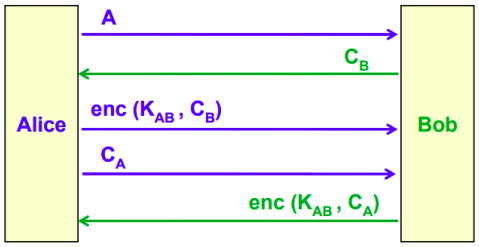
1.9.2 Mutual symmetric CRA

Mutual symmetric CRA requires both parties to authenticate each other. However, it's a old protocols so it has many vulnerabilities.

Version 1: Basic Exchange

In this case, the initiator explicitly provides its claimed identity (This version is considered outdated and insecure).
Process:

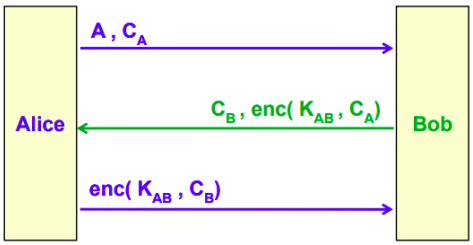
- Alice sends an encrypted challenge (C_B) to Bob using the shared key K_{AB}.
- Bob responds with an encrypted challenge (C_A) for Alice, also using K_{AB}.



Version 2: Improved Performance

Optimized by reducing the number of messages, which improves performance without compromising security.
Process:

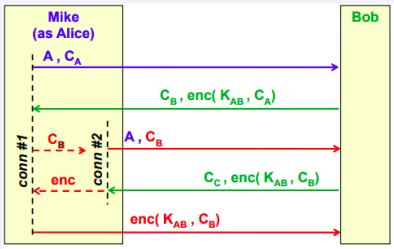
- Alice includes her identity (C_A) and sends an encrypted challenge (C_B) in the same message.
- Bob responds with his encrypted challenge $C_A$to complete the exchange.



Attack on Mutual Symmetric CRA

A potential attacker, "Mike" (posing as Alice), exploits the protocol by mimicking responses:

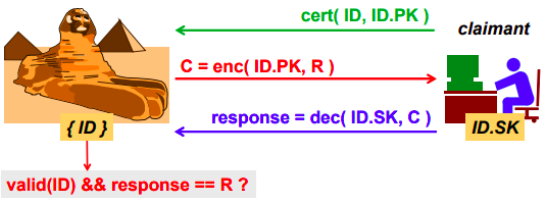
- The attacker intercepts Alice's identity (C_A) and Bob's challenge (C_B).
- The attacker uses the shared key K_{AB} to manipulate responses and mimic both parties.



1.9.3 Asymmetric CRA

Process:

- A **random nonce (R)** is generated by the Verifier.
- The verifier encrypts R using the user's public key ($ID.PK$) and sends it to the Claimant: $C = enc(ID.PK, R)$
- The Claimant decrypts C using their private key ($ID.SK$) and sends R back in cleartext: $response = dec(ID.SK, C)$
- The Verifier validates: $valid(ID)$ && $(response == R)$.



Applications

- Widely implemented in secure communication protocols like IPsec, SSH, and TLS.
- Fundamental in modern authentication frameworks such as FIDO.

Asymmetric CRA analysis

Security:

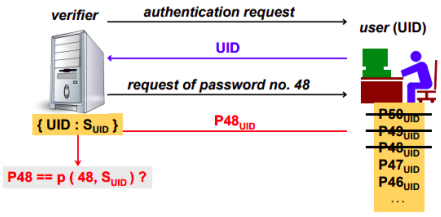
- It's the strongest mechanism.
- Does not require the Verifier to store any shared secret, reducing potential attack vectors.

Problems:

- It **slower** compared to symmetric methods.
- If designed inaccurately may lead to an involuntary signature by the Claimant.
- Trust issues managing root certificates, name constraint, and certificate revocation.

1.10 One-Time Password(OTP)

One-Time Passwords are temporary and valid for a single use in an authentication session. They mitigate risks like password reuse and passive sniffing but can still be vulnerable to man-in-the-middle (MITM) attacks. These passwords are often designed with random characters to prevent guessing, but this can make password instertion difficult for users. input.



1.10.1 S/KEY System

S/KEY System was the first OTP implementation by Bell Labs (1981). It pre-computes a sequence of passwords derived from a user's secret. Each password is validated and replaced with its predecessor, ensuring security without storing the secret:

$$Secret = S_{ID}$$

$$P_1 = h(S_{ID}), P_2 = h(P_1), ..., P_N = h(P_{N-1})$$

This approach minimizes verifier storage needs and offers robust protection, with users solely responsible for password retention.

One-time generation with S/KEY

In the S/KEY system, the user creates a secret passphrase (PP), which is combined with a server-provided seed to generate a 64-bit password. The passphrase is concatenated with the seed, and an MD4 hash is used to produce the password. The result is presented as six short words from a shared dictionary, making it easy to remember. This method allows secure password generation while using the same passphrase across multiple servers with different seeds. If the passphrase is compromised, security is at risk.