Security of Networks, Services, and Systems

Authentication Protocols

Ricardo Morla

FEUP – SSR/M.EEC, SR/M.EIC

Goals of authentication

- Goal: **authentication** of the identification provided by the user
- Depending on the identity, **authorization** to access a service will be given or not

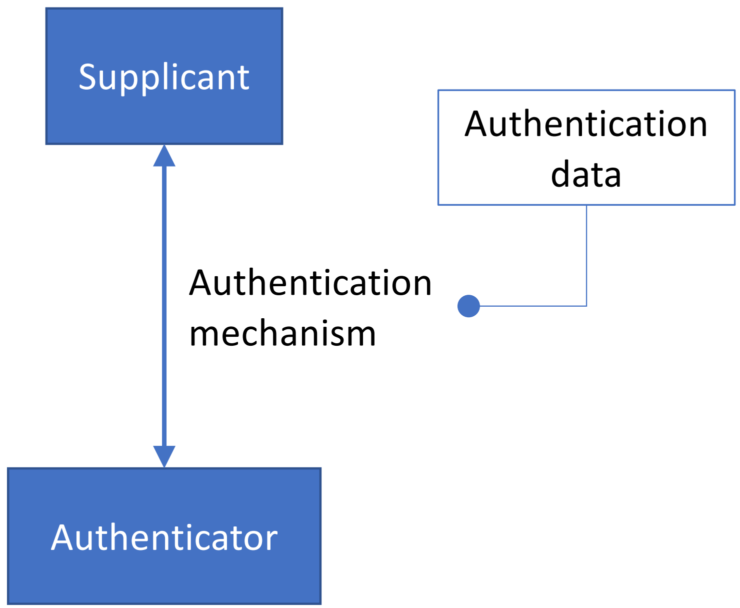
- Different from goal of authentication in **cryptographic data integrity**
  - We can claim a message is authentic by e.g. checking its MAC against the key and the message
  - If the MAC is valid, this shows that
    1. No one changed the message (data integrity)
    2. Only someone with the key was able to write the message (sender authentication)
  - Authenticates data, not user that wants access to a service

Means of authentication

- Secrets:
  - information the user has memorized – e.g. password
  - information the user can provide – e.g. crypto keys
  - physical mechanism that is hard to replicate – e.g. actual physical keys
- Biometrics:
  - static – e.g. fingerprints
  - dynamic – e.g. voice

Roles in the authentication process

- Supplicant
  - Software, hardware, or person that wants to be authenticated to access a service
- Authenticator
  - Software, hardware that checks the validity of the supplicant’s request
- Authentication data
  - Information derived from the means of authentication that the supplicant sends to the authenticator
- Authentication mechanism
  - The way authentication data is sent by the supplicant to be checked by the authenticator
- Related
  - Authorization, access control mechanism

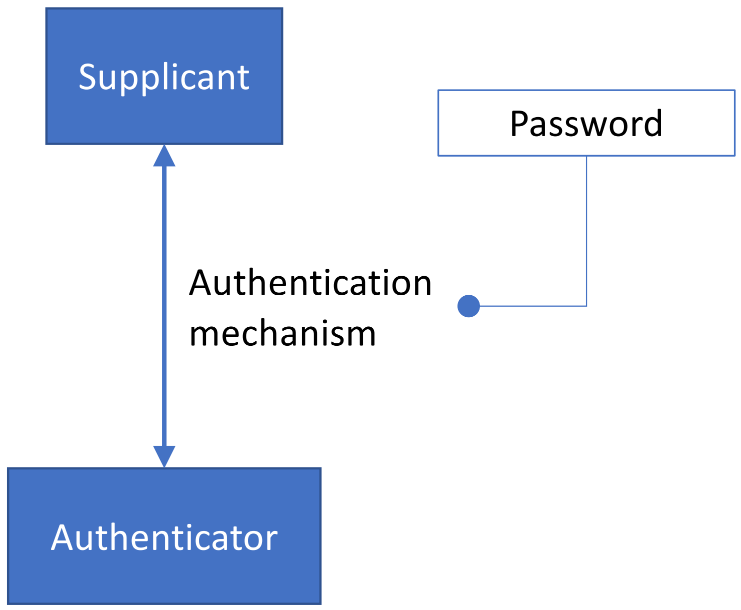


Threats for authentication

- Weak means of authentication, easy to guess
- Compromising the security of the authentication data
- Eavesdropping authentication data or replaying the authentication mechanism
- Malicious authenticator

Memorized secrets – aka passwords

- Involves the user typing password
  - e.g. `$ sudo su`
- Authenticator checks if the password is valid
- Attackers can guess password
  - Try to authenticate
  - On average they will have to fail a lot
- How hard is it to guess a password?



Threat 1 – brute forcing passwords

- Strong vs. weak passwords
- Entropy
  - Number of bits that takes to brute force the password
  - Minimum number of characters
  - Mixture of letters, numbers, upper and lower case, special characters
- Don’t use well known values
  - your birthday, etc
  - words you can find in a dictionary
  - reduces search space

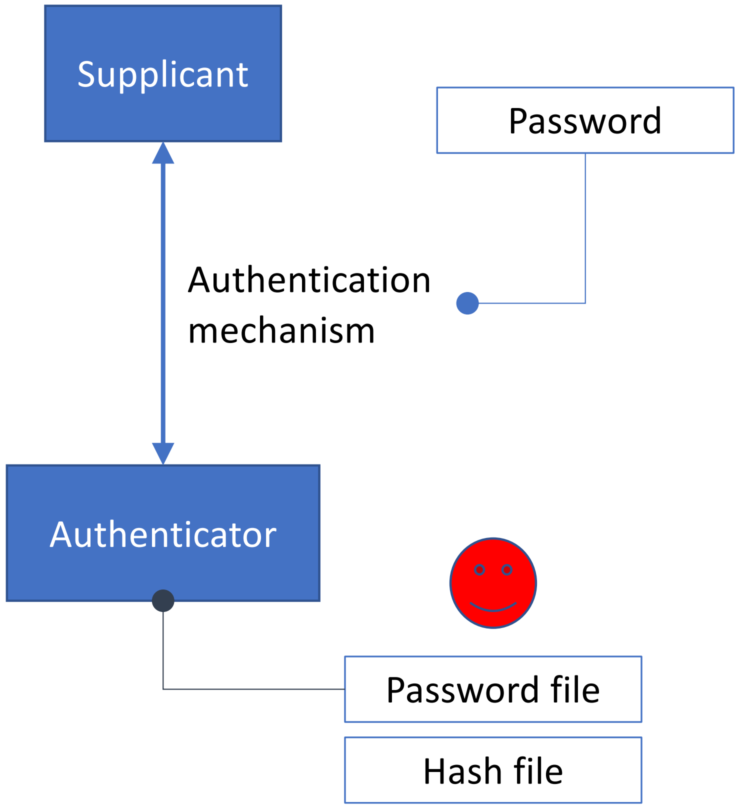
Symbol set	Symbol count *N*	Entropy per symbol *H*
Arabic numerals (0–9) (e.g. PIN)	10	3.322 bits
Hexadecimal numerals (0–9, A–F) (e.g. WEP keys)	16	4.000 bits
Case insensitive Latin alphabet (a–z or A–Z)	26	4.700 bits
Case insensitive alphanumeric (a–z or A–Z, 0–9)	36	5.170 bits
Case sensitive Latin alphabet (a–z, A–Z)	52	5.700 bits
Case sensitive alphanumeric (a–z, A–Z, 0–9)	62	5.954 bits
All ASCII printable characters except space	94	6.555 bits
Binary (0–255 or 8 bits or 1 byte)	256	8.000 bits

https://en.wikipedia.org/wiki/Password_strength

Threat 2 – data in the authenticator

- How does the authenticator checks if the password is valid?
- Store passwords in file, compare with password provided by user
  - Authentication compromised if the attacker has access to the file
- Store password in file, encrypt file
  - Authentication compromised if the attacker has access to the file and key
- Store hashes of password in file
  - Authentication compromised if attacker can brute force hashes (pre-image attack)
  - Authentication compromised if authenticator accepts hashes instead of passwords

    https://en.wikipedia.org/wiki/Pass_the_hash



Rainbow table attack

- Pre-image attacks are computationally expensive
- Rainbow table is a tradeoff between computation, storage, and hash coverage
- Create chain of hashes with hash and 'Reduce' functions, store only first and last items in chain

$$\underset{\texttt{pwd}}{\texttt{aaaaaa}} \xrightarrow[H]{} \underset{\texttt{hash}}{\texttt{281DAF40}} \xrightarrow[R]{} \underset{\texttt{pwd}}{\texttt{sgfnyd}} \xrightarrow[H]{} \underset{\texttt{hash}}{\texttt{920ECF10}} \xrightarrow[R]{} \underset{\texttt{pwd}}{\texttt{kiebgt}}$$

- Find reduced hash in table, applie H/R to first item

$$\texttt{920ECF10} \xrightarrow[R]{} \texttt{kiebgt} \qquad \underset{\texttt{pwd}}{\texttt{aaaaaa}} \xrightarrow[H]{} \underset{\texttt{hash}}{\texttt{281DAF40}} \xrightarrow[R]{} \underset{\texttt{pwd}}{\texttt{sgfnyd}} \xrightarrow[H]{} \underset{\texttt{hash}}{\texttt{920ECF10}}$$

`sgfnyd` is the password for hash `920ECF10`

Rainbow table attack example

Hash for which we want the password:
`re3xes`

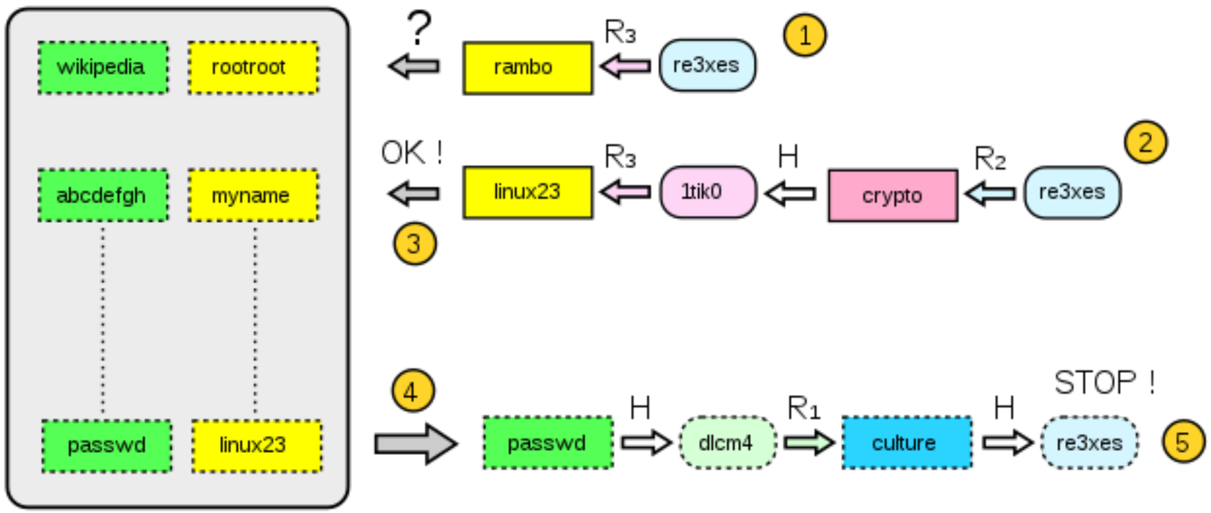
1. Apply Reduce to hash, find in table (`rambo` not found)

2. Apply Reduce/Hash/Reduce to hash, find in table (`linux23` found)

3. First item in chain for `linux23` `passwd`

4. Start with `passwd`, apply H/R/H until `re3xes` found

5. Password for `re3xes` is previous password in chain



If can't find values in table after multiple H/R/H, attack fails

https://pentestmonkey.net/cheat-sheet/john-the-ripper-hash-formats
https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/
https://en.wikipedia.org/wiki/Rainbow_table

Salts to prevent rainbow table attack

- Concatenate a ’salt’ value to the password before hashing

```
saltedhash(password) = hash(password + salt)
```

- Salt value not secret, can be random, can be stored with the hash
- Cost for attacker: one rainbow table for each salt value
  - 12 bits => 4096 tables
- Linux /etc/shadow

```
openssl passwd -1 -salt xyz mypass

user1:$1$xyz$Hroq70ktxuFpz2u8V9Mdb0:13064:0:9999:7:::
```

1 Login

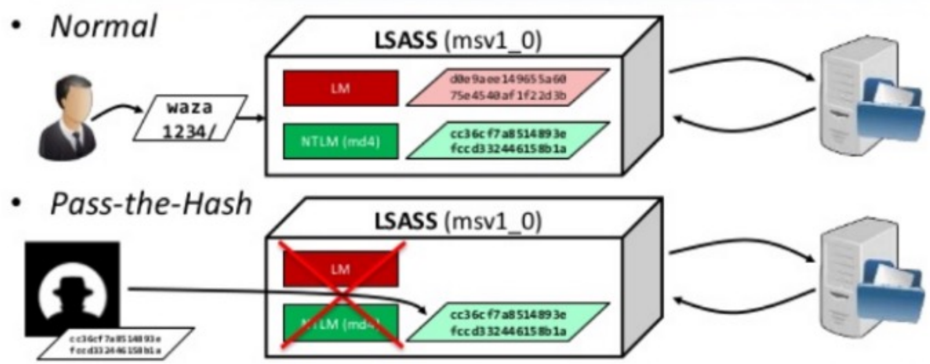
2 Password: idsalt$hash
id: (1,MD5), (5,SHA-256),(6,SHA512

Threat 3 – eavesdropping

- Supplicant needs to send authentication data over the wire
- If password sent in plain text
  - Authentication compromised if attacker eavesdrops channel
- Options
  - Send hash instead of password (don’t, please)
  - Use secure communications channel to send passwords or hashes

Pass the hash

- To avoid sending plaintext passwords, send hash instead of password
  - This prevents eavesdropper from knowing password
- Opens door to brute-forcing the hash sent in plaintext
- Worse: now the attacker only needs to get hold of the hash, not the password, to accesss the service



https://en.wikipedia.org/wiki/Pass_the_hash

SSH authentication with password
RFC 4252

- Setup user and password on server
- Supplicant and authenticator negotiate encrypted channel
  - /.ssh/known_hosts
  - DH key exchange, encrypt data with block ciphers and MAC
- Supplicant sends plaintext password over encrypted channel
- Authenticator checks password, allows access or not

Threat 4 – malicious authenticator

- At some point the authenticator will have to get access to the password or the password hash to validate it
- If authenticator is compromised it receives password or hash from supplication and can reuse it in other services
  - Think ssh server key in /etc/known_hosts compromised
- Use implicit authentication to not send password or hash:
  - Challenge/response including password or hash of password
  - Asymmetric keys

Implicit Authentication – challenge/response
RFC 2759

- MSCHAPv2 Challenge-Handshake Authentication Protocol

- Authenticator sends session id and authenticator challenge string
- Supplicant sends username, supplicant challenge string, and hash of: (both challenge strings, session id, and hash of user password)
  - Note that supplicant does not send hash of password
- Authenticator validates with stored hash of user password

Implicit Authentication – SSH, public key
RFC 4252

- Possession of private key serves as authentication instead of password or hash
  - Server has public key of user to validate
- Method
  - Hash over session id, user name, and other shared data
  - Client encrypts hash with private key, sends to server
  - Server decrypts, validate if decrypted hash similar to its own hash
    - Only works if client has the private key associated with the user's public key that the server has

Asymmetric keys - SSH authentication RFC4252

- $ ssh-keygen
- $ ssh-copy-id -i ~/.ssh/id_rsa.pub remote-host % provide password
- $ ssh remote-host
  % no need to provide password
- $ ssh remote-host -v
  % will verbose authentication steps

Extensible Authentication Protocol
RFC 3748

- Authentication framework with different authentication methods
- Method examples
  - EAP-TLS: authenticate user by certificate
  - EAP-TTLS: establish tunnel, authenticate user by which ever method - could be clear text password, hash, etc
  - PEAP/MS-CHAPv2 : establish tunnel, use MS-CHAPv2 protocol for authentication with user password

https://en.wikipedia.org/wiki/Extensible_Authentication_Protocol

Securing your keys with Smart Cards

Personal Identification Verification

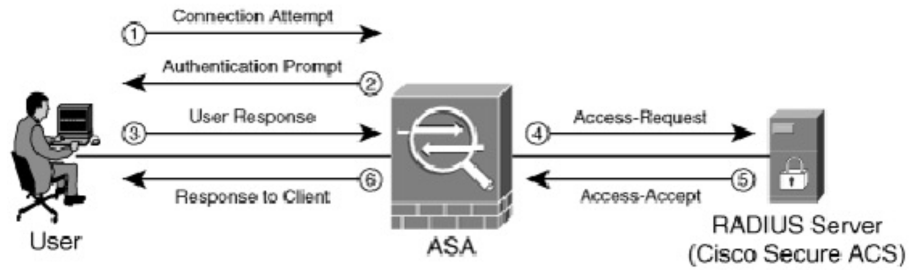
- Keys are stored on the card
  - useful when you have many long keys
- Keys do not leave card
  - useful when you don't trust the computer you're accessing from
- How can you use the keys if they don't leave the card?
  - Encryption/signature processing are done by the card's microprocessor
  - Asymmetric keys and key certificate for encrypting and signing PIN number secures access to card's private key

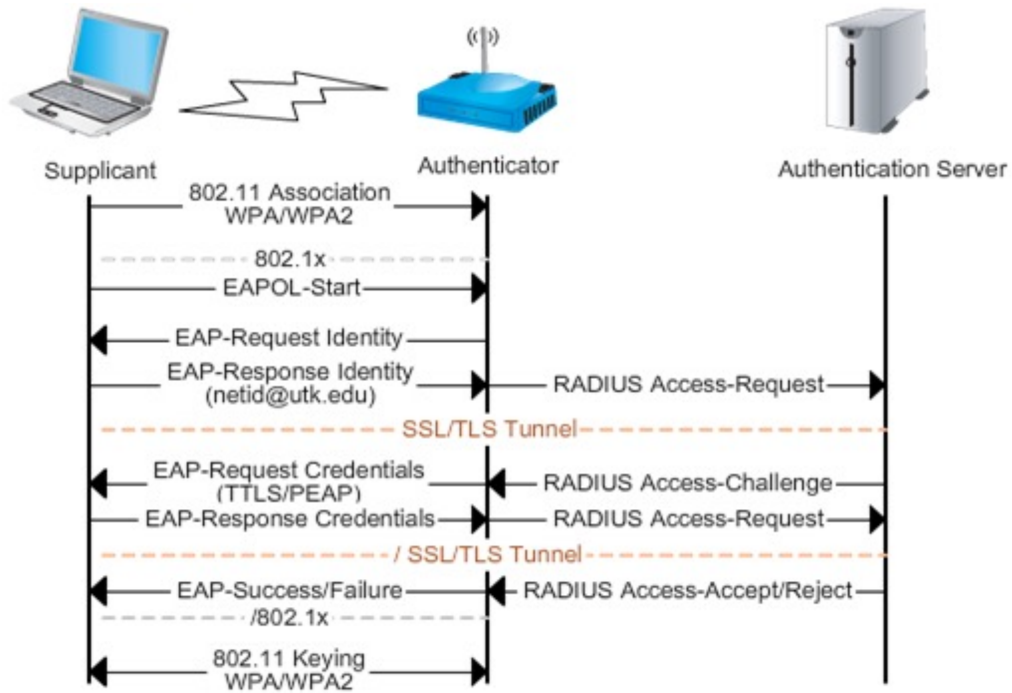
Smart cards, SSH, others

- $ opensc-tool - - list-readers
- $ pkcs15-init -S .ssh/id_rsa - - auth-id 01 - -label "My Private SSH Key" - - public-key-label "My Public SSH Key"
  % store existing key in card
- $ pkcs15-tool - - list-public-keys
- $ pkcs15-tool - - read-ssh-key <keyid>
  % get public key, copy to .ssh/authorized_keys on the server
- $ ssh -I /usr/lib/x86_64-linux-gnu/opensc-pkcs11.so 10.0.0.1 % access a secure shell on the server
- Encryption and signing operations not related to SSH: $ pkcs15-crypt [options]

RADIUS

- AAA: Accounting + Entangled Authentication and Authorization
- Network Service Providers authenticate clients on RADIUS on the client's behalf
  - Or allow traffic only to authentication server (TLS tunnel)
- Explicit authentication or challenge-response





Kerberos

Symmetric Key Authentication Infrastructure

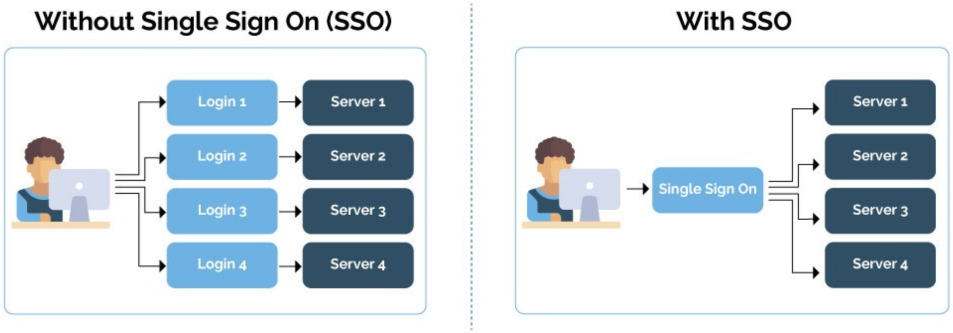
- Centralized authentication and authorization
  - Per-service access control
- Client sends client and service id together with password to server
  - Server authenticates password and user id, authorizes user for service
- Server generates access ticket
  - encrypts client and service id with service's key
- Client accesses service by providing ticket and client id
- Service decrypts ticket and checks if client and service id match
  - possession of access ticket serves as authentication

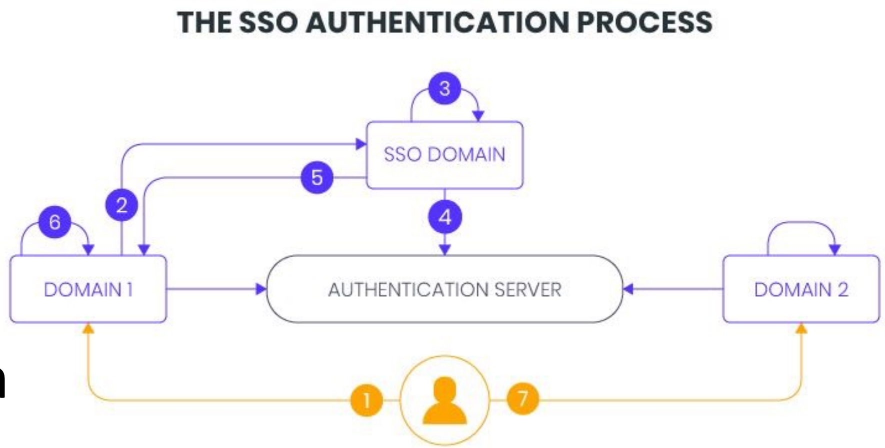
Multi-factor authentication

- If more than one independent authentication methods are used, the harder it is to break the authentication
- Example: mobile phone two-factor authentication
  - Factor 1: Password provided at web site
  - Factor 2: One-time code sent through another channel (e.g. SMS)

Single sign-on

- User wants to access service
- Service provider redirects user to authentication service
- Authentication service returns access token and information to service provider
- Service provider allows access without requiring to directly register the user on the website or checking passwords





https://spin.atomicobject.com/2016/05/30/openid-oauth-saml/

SSO alternatives

- OAuth2
  - https://oauth.net/2/
  - Authorization
  - JSON and REST, HTTP
  - Used for API authorization, e.g. Google API
- OpenID Connect
  - Built over OAuth2
  - Authentication and basic user profile
  - Used across the web industry
- Shibboleth and SAML
  - XML-based
  - Authentication and authorization
  - Used mostly in academics
  - Federation of identity providers
  - Where are you from WAYF service

Security of Networks, Services, and Systems

Authentication Protocols

Ricardo Morla

FEUP – SSR/M.EEC, SR/M.EIC