TLS 1.3 and authenticated key-exchange protocols on the Internet typically provide one-sided authentication (i.e., client learns id of the server, but not vice versa).

Question: how does the client authenticate to the server (without providing a certificate)?

\[ \text{e.g., how does client login to a web service?} \]

Typical setting:

\[ \text{client and server assumed to have some shared state} \]

\[ (sk) \quad \text{client} \quad (vk) \quad \text{server} \]

\[ \text{client learns server's identity} \quad \text{cannot replace this with anonymous key exchange} \]

\[ \text{identification protocol} \quad \text{becomes vulnerable to a man-in-the-middle attack} \]

\[ \text{AKE protocol} \]

Threat models: Adversary's goal is to authenticate to server

- Direct attack: adversary only sees \( vk \), and needs to authenticate
  \[ \text{(e.g., physical analogy: door lock) adversary can observe the lock, does not see the key} \ sk \]

- Eavesdropping attack: adversary gets to observe multiple interactions between honest client and the server
  \[ \text{(e.g., physical analogy: wireless car key) adversary observes communication between car key and car} \]

- Active attack: adversary can impersonate the server and interact with the honest client
  \[ \text{(e.g., physical analogy: fake ATM in the mall) honest clients interact directly with the adversary} \]

Simple (insecure) password-based protocol:

\[ \text{client} [sk : pwd] \quad \text{server} [vk : pwd] \]

\[ \text{pwd} \quad \text{accept if} \ vk = \text{pwd} \]

Not secure even against direct attacks! Adversary who learns \( vk \) can authenticate as the client \[ \text{adversary who breaks into server} \]

\[ \text{learns user's password!} \]

**NEVER STORE PASSWORDS IN THE CLEAR!**

Slightly better solution: hash the passwords before storing

\[ \text{server maintains mappings } \text{Alice} \rightarrow H(\text{pwd}_\text{Alice}) \quad \text{Bob} \rightarrow H(\text{pwd}_\text{Bob}) \]

\[ \text{where } H \text{ is a collision-resistant hash function} \]

\[ \text{client} [sk : \text{pwd}] \quad \text{server} [vk : H(\text{pwd})] \]

\[ \text{pwd} \quad \text{accept if} \]

\[ vk = H(\text{pwd}) \]
If passwords have high entropy, then hard to recover password via \( H(pw) \) [by one-wayness of \( H \)]

\[ \rightarrow \text{But not true in practice...} \]

Users often choose weak passwords (e.g., 123456, password, 123456789, ...)

\[ \rightarrow \text{With a dictionary of 360 million entries, can cover about 25% of user passwords} \]

\[ (5\% \text{ choose 123456}) \]

\[ (10\% \text{ choose among top 25 common passwords}) \]

Based on password hashes that have been leaked from compromised databases

Simple hashing vulnerable to "offline dictionary attack":

Adversary computes table \( (p\text{aid}, H(p\text{aid})) \) for common passwords — completely offline.

Given \( H(p\text{aid}) \), can now invert with a single lookup if \( pw\text{aid} \) is contained in the database.

For LinkedIn breach in 2012, attacker stole password file with \( \approx6\text{ million passwords} \)

\( \text{all passwords hashed using single iteration of unmasked SHA-2} \rightarrow 90\% \text{ of passwords recovered in } \approx6 \text{ days!} \)

Problem: One-time precomputation (computing the lookup table) can be reused to compromise many passwords.

Overall cost of attack: \( O(m+n) \) where \( m \) is the dictionary size and \( n \) is the number of passwords to attack.

Define #1: Salt passwords before hashing: namely when storing password \( pw\text{aid} \), sample \( \text{Salt} \approx \{0,1\}^e \) and store \( \text{Salt, } H(\text{Salt}\|\text{pw}) \) on the server.

Note: Salt is a public value (needed for verification)

Typically, \( e \approx 64 \)

Offline dictionary attack no longer effective since every salt value induces different set of hash values.

Overall cost of dictionary attack: \( O(mn) \) — need to re-hash dictionary for every salt.

Define #2: Use a slow hash function [SHA-1 is very fast — enables fast brute-force search]

- PBKDF2 (password-based key-derivation function): iterate a cryptographic hash function many times:

\[ \text{PBKDF2}(\text{pad}, \text{salt}) = H(H(\cdots H(\text{Salt}\|\text{pw})\cdots)) \]

\[ \text{Can use 100,000 or} \quad (100,000,000) \text{ iterations of SHA-256} \]

Drawback: custom hardware can evaluate SHA-256 very fast

- Scrypt: more recent: Argon2i): slow hash function that needs lots of memory (space) to evaluate

\[ \text{Custom hardware do not provide substantial savings (limiting factor is space, not compute)} \]

Can also use a layered hash function (e.g., HMAC with key stored in HSM)

\[ \text{Ensures adversary who does not know key cannot brute force at all!} \]

Best practice: Always salt passwords

Always use a slow hash function (e.g., PBKDF2, scrypt) or layered hash function or both!

\[
\text{
$\text{cur} = \text{'password'}$
$\text{cur} = \text{md5($\text{cur}$)} \text{ raw MD5 hash - not secure!}$
$\text{salt} = \text{randbytes(20)}$
$\text{cur} = \text{hmac_sha1($\text{cur}, \text{salt}$)}$
$\text{cur} = \text{remote_hmac_sha256($\text{cur}, \text{secret}$)}$
$\text{cur} = \text{scrypt($\text{cur}, \text{salt}$)}$ \text{ slow hash function}$
$\text{cur} = \text{hmac_sha256($\text{cur}, \text{salt}$)}$

Facebook password onion
\]

\[ \text{(circa 2011)} \]

\[ \text{Layers gradually added over time to achieve better security}$
\]

\[ \text{(and probably to avoid password reusing} \]

\[ \text{Partially)} \]
Password-based protocols are secure against eavesdropping adversaries (adversary sees k and transcript of multiple interactions between honest server and honest verifier).

One-time passwords (OTP) (SecurID tokens, Google authenticator, Duo)

**Construction 1:** Consider setting where verification key vk is secret (e.g., server has a secret)
- Client and server have a shared PRF key k and a counter initialized to 0.
- Client (k,c)
- Server (k,c)

\[
\begin{align*}
 C' \leftarrow F(k,c) \\
 y \leftarrow F(k,c) \\
 C \leftarrow C + 1
\end{align*}
\]

Concretely: can interpret output as 6-digit number
- check that \( y = F(k,c) \) and \( c' > c \) (no replaying) \{ car key authentication

- RSA SecurID: stateful token (counter incremented by pressing button on token)
  \( \Rightarrow \) State is cumbersome - need to maintain consistency between client/server
- Google Authenticator: time-based OTP: counter replaced by current time window (e.g., 80-second window)

If PRF is secure \( \Rightarrow \) Above protocol secure against eavesdroppers (but requires server secrets)

**Construction 2:** No server-side secrets (3/keys) \( \sim \) "under composition"
- Relies on a hash function (should be one-way)
- Secret key is random input x and counter n
- Verification key is \( H^{(n)}(x) = H(H(H(\ldots H(x) \ldots))) \)

\[
\begin{align*}
 x & \rightarrow H(x) & H^{(2)}(x) & \ldots & H^{(n)}(x)
\end{align*}
\]

\( \Rightarrow \) to verify \( y \): check \( H(y) = \text{vk} \) \{ attacker has to invert \( H \) in order to authenticate

- Verification key can be public (credential is preimage of \( \text{vk} \))
- Can support bounded number of authentications (at most \( n \)) - need to update key after \( n \) logins
- Output needs to be large (> 80 bits or 128 bits) since password is the input/output to the hash function
- Naively, client has to evaluate \( H \) many times per authentication (< \( O(n) \) times)
- Can reduce to \( O(\log n) \) hash evaluations in an amortized sense by storing \( O(\log n) \) entries along the hash chain

Thus far, only considered passive adversaries, but in reality, adversaries can be malicious \( \Rightarrow \) no man-in-the-middle protection
- Adversary can impersonate server (e.g., phishing) and then try to authenticate as client (but cannot interact with client during such)
- All protocols thus far are vulnerable \( \{ \) all consist of client sending token that server checks, which can be extracted by active adversary
- For active security, we use challenge-response
Signature-based challenge–response
- Server stores a verification key \( vk \) for digital signature scheme
- Client holds signing key \( sk \)

\[
\begin{array}{c}
\text{client (sk)} \\
\xleftarrow{\text{random message}} \\
\text{server (vk)} \\
\end{array}
\]

\[
\text{m} \xleftarrow{\text{M}} \quad \sigma \leftarrow \text{Sign} (sk, m) \\
\]

check that \( \text{Verify} (vk, m, \sigma) = 1 \)

Server asks client to sign a random message

- Client’s signature indicates proof of possession of \( sk \) associated with \( vk \)
- Active adversary that interacts with the client before interacting with the prover cannot forge signatures

Provides active security but signatures are long (~384 bits)