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)?

- e.g., how does client login to a web service?

Typical setting:

\[ (sk) \] client → \[ \text{some shared state} \] \[ (vk) \] server

- client learns server’s identity
- cannot replace this with anonymous key exchange
- 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
  (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
  (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
  (e.g., physical analogy: fake ATM in the mall — honest clients interact directly with the adversary)

Simple (insecure) password-based protocol:

\[
\begin{align*}
\text{client} & \quad [sk: \text{pwd}] & \quad \text{server} & \quad [vk: \text{pwd}] \\
\text{pwd} & \quad \downarrow & \quad \text{accept if } vk = \text{pwd}
\end{align*}
\]

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

**NEVER STORE PASSWORDS IN THE CLEAR!**

Slightly better solution: hash the passwords before storing

\[
\begin{align*}
\text{server} & \quad \text{maintains mappings} & \quad \text{Alice} & \mapsto H(\text{pwd}_{\text{Alice}}) \\
& & \quad \text{Bob} & \mapsto H(\text{pwd}_{\text{Bob}})
\end{align*}
\]

where \( H \) is a collision-resistant hash function

\[
\begin{align*}
\text{client} & \quad [sk: \text{pwd}] & \quad \text{server} & \quad [vk: H(\text{pwd})] \\
\text{pwd} & \quad \downarrow & \quad \text{accept if } \quad & \quad \text{vk} = H(\text{pwd})
\end{align*}
\]

\[ \text{where } H \text{ is a collision-resistant hash function} \]
If passwords have high entropy, then hard to recover from $H(\text{pwd})$ \cite{ow} by one-wayness of $H$.

$\Rightarrow$ But not true in practice...

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

$\Rightarrow$ With a dictionary of 360 million entries, can cover about 25% of user passwords.

(3% choose 123456).

(10% 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 $(\text{pwd}, H(\text{pwd}))$ for common passwords — completely offline.

given $H(\text{pwd})$ can now invert with a single lookup if pwd is contained in the database for "Hashed In Brunch" in 2012, attackers stole password file with $\sim 6$ million passwords.

(all passwords hashed using single iteration of uncorrected SHA-2) $\Rightarrow$ 90% of passwords recovered in $\sim 6$ 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 store stringing password, salt, $H(\text{salt}||\text{pwd})$ on the server typically, $n \geq 64$.

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

Offline dictionary attack no longer effective once 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-2 is very fast — enables fast brute-force search].

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

\[
\text{PBKDF2} (\text{pwd}, \text{salt}) = H(H(\cdots H(\text{salt}||\text{pwd})\cdots))
\]

Can use 100,000 or 100,000,000 iterations of SHA-256.

Drawback: custom hardware can evaluate SHA-256 very fast.

- Scrypt (more recent: Argon2): slow hash function that needs lots of memory (space) to evaluate.

- Custom hardware do not provide substantial savings (limiting factor is space, not compute).

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

- 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 keyed hash function or both!

$\text{cur} = \text{‘password’}$

$\text{cur} = \text{md5(\text{cur})}$ raw MD5 hash — not secure.

$\text{salt} = \text{randbytes(20)}$

$\text{cur} = \text{hmac\_sha1(\text{cur}, \text{salt})}$ salted, keyed hash function (key on remote service)

$\text{cur} = \text{remote\_hmac\_sha256(\text{cur}, \text{secret})}$ layers gradually added over time to achieve better security (and probably to avoid password reusing.

$\text{cur} = \text{scrypt(\text{cur}, \text{salt})}$ slow hash function

$\text{cur} = \text{hmac\_sha256(\text{cur}, \text{salt})}$ Facebook password onion (circa 2014)
Password-based protocols not secure against eavesdropping adversary

(adversary sees v_k and transcript of multiple interactions between honest peer + honest verifier)

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

Construction 1: Consider setting where verification key v_k is secret (e.g., server has a secret)
- Client and server have a shared PRF key k_0 and a counter (initialized to 0):

\[ C', y' \leftarrow F(k, c) \]
\[ c \leftarrow c + 1 \]

\[ \text{output as 6-digit number} \]

- RSA SecurID: stateful token (counter incremented by pressing button on token)
  - State is cumbersome - need to maintain consistency between client/server
- Google Authenticator: time-based OTP: counter replaced by current time window (e.g., 30-second window)

If PRF is secure \[ \Rightarrow \] above protocol secure against eavesdroppers (but requires server secrets)

Construction 2: No server-side secrets (3-key) "under computation"
- Relies on a hash function (should be one-way)
- Secret key is random input x and counter n
- Verification key is \[ H^{(n)}(x) = H(H(...H(x)...)) \]

\[ \text{for n evaluations of H} \]

\[ x \rightarrow H(x) \rightarrow H^{(2)}(x) \rightarrow ... \rightarrow H^{(n)}(x) = v_k \]

- Verification key can be public (credential is preimage of v_k)
  - 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
  - Necessarily, 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

\[ \Rightarrow \text{no man-in-the-middle protection} \]

- 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 \[ \{ \text{all consist of client sending token that server checks, which can be extracted by active adversary} \}
- For active security, use challenge-response