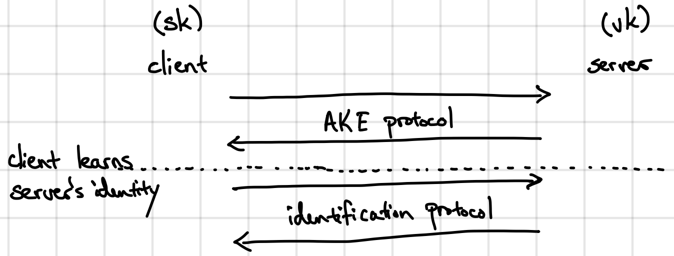
TLS 1.3 and authenticated key-exchange protocols on the Internet typically provide one-sided authentication (i.e., client learns id of the server, but not vice versa)

Question: how does the client authenticate to the server (without providing a certificate)
↳ e.g., how does client login to a web service?

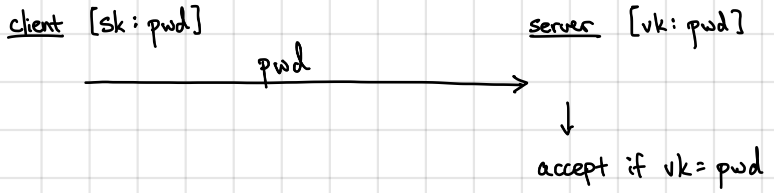
Typical setting: client and server assumed to have some shared state [e.g., client has a password and server has an HMAC of the password]



} cannot replace this with anonymous key exchange
↳ becomes vulnerable to a man-in-the-middle attack

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:



Not secure even against direct attacks! Adversary who learns vk can authenticate as the client [adversary who breaks into server learns user's password!]

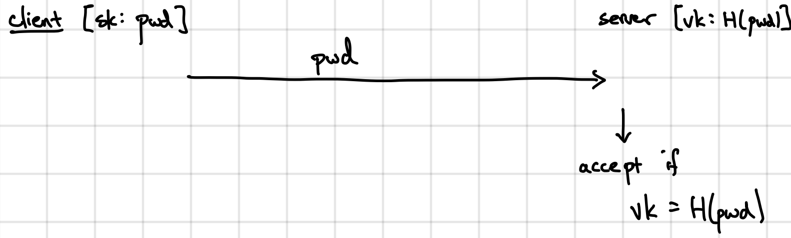
NEVER STORE PASSWORDS IN THE CLEAR!

Slightly better solution: hash the passwords before storing server maintains mappings

$\text{Alice} \mapsto H(\text{pwd}_{\text{Alice}})$

$\text{Bob} \mapsto H(\text{pwd}_{\text{Bob}})$

where H is a collision-resistant hash function



If passwords have high entropy, then hard to recover pwd from $H(pwd)$ [by one-wayness of H]

↳ But not true in practice...

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

↳ 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 $(pwd, H(pwd))$ for common passwords — completely offline

given $H(pwd)$, can now invert with a <u>single</u> lookup if pwd is contained in the database

for LinkedIn breach in 2012, attacker stole password file with ~6 million passwords

(all passwords hashed using single iteration of unsalted SHA-1) ⟶ 90% of passwords recovered in ~6 days!

<u>Problem</u>: One-time precomputation (computing the lookup table) can be <u>reused</u> to compromise <u>many</u> passwords

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

<u>Defense #1</u>: <u>Salt</u> passwords before hashing: namely when storing password pwd, sample $salt \xleftarrow{R} \{0,1\}^n$ and store $(salt, H(salt \| pwd))$ on the server

↑ typically, $n \geq 64$

<u>Note</u>: Salt is a <u>public</u> value (needed for verification)

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

<u>Defense #2</u>: Use a <u>slow</u> 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:
(or bcrypt)

$$PBKDF2(pwd, salt): \underbrace{H(H(\cdots H(salt \| pwd)\cdots))}_{\text{can use 100,000 or 1,000,000 iterations of SHA-256}}$$

honest user only needs to evaluate hash function once per authentication; adversary evaluates <u>many</u> times

<u>Drawback</u>: custom hardware can evaluate SHA-256 <u>very</u> fast

- scrypt (more recent: Argon2i): slow hash function that needs lots of <u>memory (space)</u> to evaluate

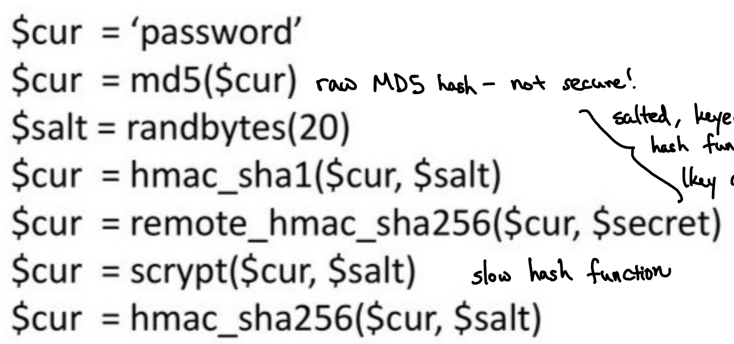
↳ custom hardware do not provide substantial savings (limiting factor is <u>space</u>, not compute)

Can also use a <u>keyed</u> hash function (e.g., HMAC with key stored in HSM)

↳ ensures adversary who does not know key cannot brute force <u>at all</u>!

<u>Best practice</u>: Always salt passwords

Always use a slow hash function (e.g., PBKDF2, scrypt) or keyed hash function or <u>both</u>!



```
$cur  = 'password'
$cur  = md5($cur)
$salt = randbytes(20)
$cur  = hmac_sha1($cur, $salt)
$cur  = remote_hmac_sha256($cur, $secret)
$cur  = scrypt($cur, $salt)
$cur  = hmac_sha256($cur, $salt)
```

raw MD5 hash - not secure!

salted, keyed hash function (key on <u>remote</u> service)

slow hash function

Facebook password onion (circa 2014)

↓

layers gradually added over time to achieve better security

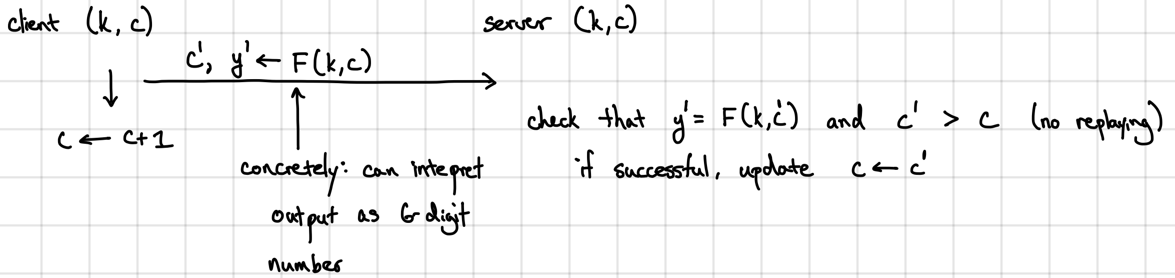
(and probably to <u>avoid</u> password rehashing)

Password-based protocol <u>not</u> secure against eavesdropping adversary
(adversary sees vk and transcript of multiple interactions between honest prover + honest verifier)

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

<u>Construction 1</u>: Consider setting where verification key vk is <u>secret</u> (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)
$c', y' \leftarrow F(k, c)$
$c \leftarrow c+1$
concretely: can intepret output as 6-digit number

check that $y' = F(k, c')$ and $c' > c$ (no replaying)
if successful, update $c \leftarrow c'$

} car key authentication

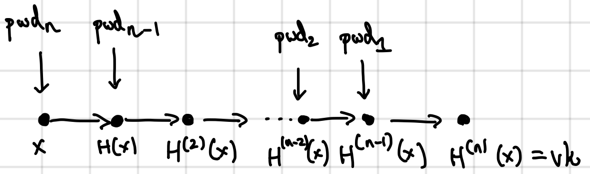
- <u>RSA SecurID</u>: stateful token (counter incremented by pressing button on token)
  - ↳ State is cumbersome — need to maintain consistency between client/server
- <u>Google Authenticator</u>: time-based OTP: counter replaced by current time window (e.g., 30-second window)

If PRF is secure ⟹ above protocol secure against eavesdroppers (but requires <u>server</u> secrets)

↳ can be problematic: RSA breached in 2011 and SecurID tokens compromised and used to compromise defense contractor Lockheed Martin

<u>Construction 2</u>: No server-side secrets (S/Key)

- Relies on a hash function (should be one-way) "under composition"
- Secret key is random input x and counter n;
  Verification key is $H^{(n)}(x) = \underbrace{H(H(\cdots H(x)\cdots))}_{n \text{ evaluations of } H}$



to verify y: check $H(y) \overset{?}{=} vk$
if successful, update $vk \leftarrow y$

} attacker has to invert H in order to authenticate

- Verification key can be public (credential is preimage of vk)
  - ↳ Can support <u>bounded</u> 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 <u>input</u>/<u>output</u> 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 <u>passive</u> adversaries, but in reality, adversaries can be <u>malicious</u>

- Adversary can impersonate server (e.g., phishing) and then try to authenticate as client (but cannot interact with client during auth.) no man-in-the-middle protection
- 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 <u>challenge-response</u>