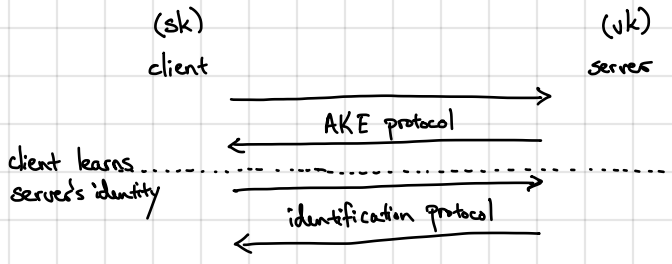
TLS 1.3 and authenticated key-exchange protocols on the Internet typically provide <u>one-sided</u> authentication (i.e., client learns id of the server, but not vice versa)

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

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



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

<u>Threat models</u>: Adversary's goal is to authenticate to server

- <u>Direct attack</u>: 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)
- <u>Eavesdropping attack</u>: adversary gets to <u>observe</u> multiple interactions between honest client and the server
  (e.g., physical analogy: wireless car key — adversary observes communication between car key and car)
- <u>Active attack</u>: 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:

<u>client</u> [sk: pwd] — pwd → <u>server</u> [vk: pwd]

↓

accept if vk = pwd

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!

<u>Slightly better solution</u>: hash the passwords before storing

server maintains mappings Alice ↦ $H(pwd_{Alice})$
Bob ↦ $H(pwd_{Bob})$

where H is a collision-resistant hash function

<u>client</u> [sk: pwd] — pwd → <u>server</u> [vk: H(pwd)]

↓

accept if vk = H(pwd)

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 single 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!

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

Defense #1: Salt 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, $n \geq 64$

Note: Salt is a public 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

Defense #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:
(or bcrypt) PBKDF2(pwd, salt): $\underbrace{H(H(\cdots H(\text{salt} \| \text{pwd}) \cdots))}$
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 many times

Drawback: custom hardware can evaluate SHA-256 very fast
- scrypt (more recent: Argon2i): 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!

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

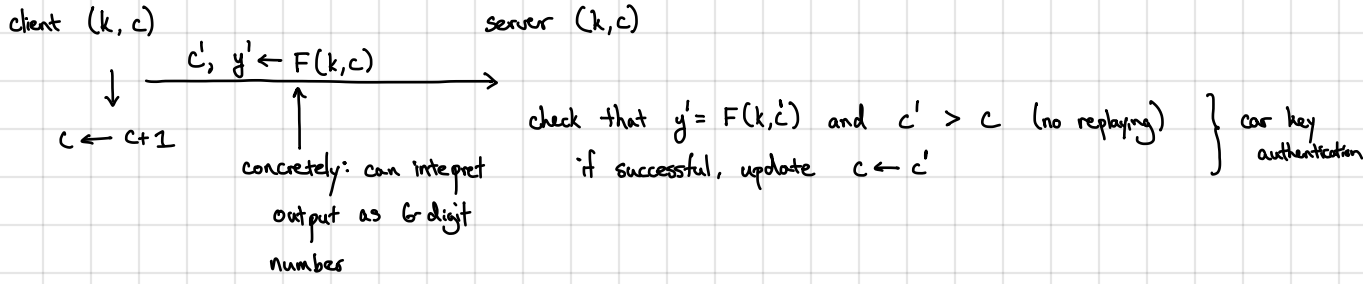
salted, keyed hash function (key on remote service)

Facebook password onion (circa 2014)
↓
layers gradually added over time to achieve better security
(and probably to avoid password rehashing)

Password-based protocol not 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)

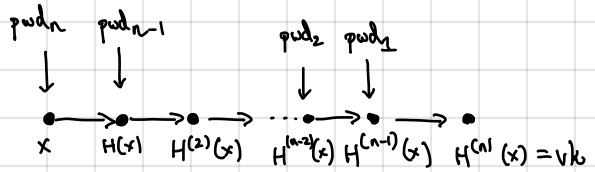
Construction 1: Consider setting where verification key vk is secret (e.g., server has a secret)
- Client and server have a shared PRF key k and a counter (initialized to 0):



- 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 windows)

If PRF is secure ⇒ above protocol secure against eavesdroppers (but requires server secrets)
↳ can be problematic: RSA breached in 2011 and SecurID tokens compromised and used to compromise defense contractor Lockheed Martin

Construction 2: No server-side secrets (S/Key) ← "under composition"
- Relies on a hash function (should be one-way)
- Secret key is random input x and counter n;
  Verification key is $H^{(n)}(x) = \underbrace{H(H(\cdots H(x)\cdots))}_{n \text{ evaluations of } H}$



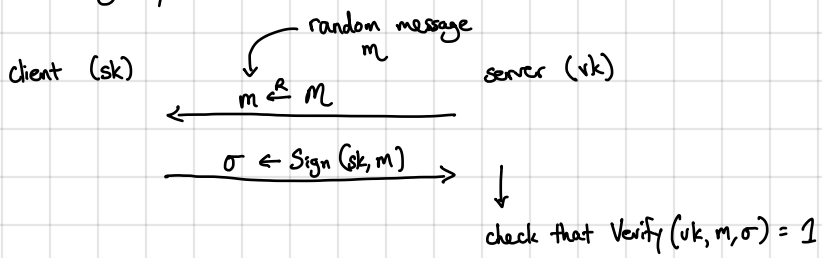
to verify y: check $H(y) \stackrel{?}{=} vk$
if successful, update $vk \leftarrow y$
} attacker has to invert H in order to authenticate

- Verification key can be public (credential is preimage of vk)
  ↳ Can support 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
- Naïvely, 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
- 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 challenge-response

Signature-based challenge-response

- Server stores a verification key vk for digital signature scheme
- Client holds signing key sk



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)

Signature-based challenge response: client "demonstrates knowledge" of signing key

↳ we will generalize this to "proving" arbitrary statements