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): $H(H(\cdots H(\text{salt} \| \text{pwd}) \cdots))$
  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)             salted, keyed hash function (key on remote service)
$cur = remote_hmac_sha256($cur, $secret)
$cur = scrypt($cur, $salt)                slow hash function
$cur = hmac_sha256($cur, $salt)
```

Facebook password onion (circa 2014)

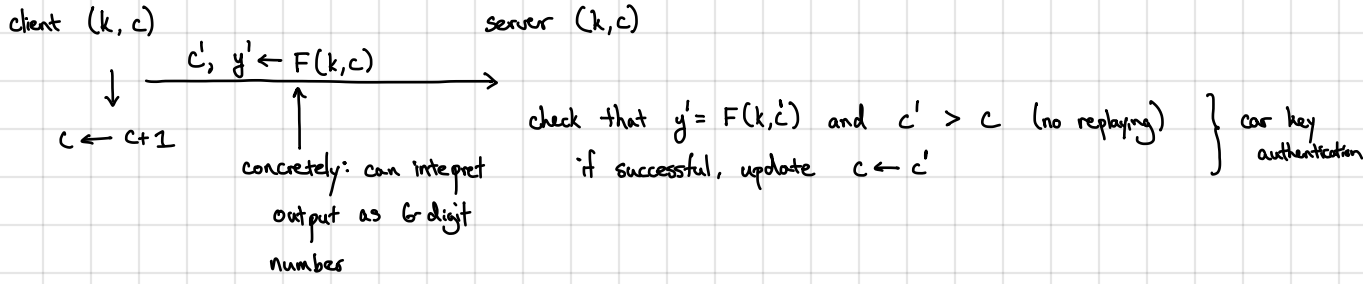
↓

layers gradually added over time to achieve better security (and probably to avoid 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 (SecurID tokens, Google authenticator, Duo)
(OTP)

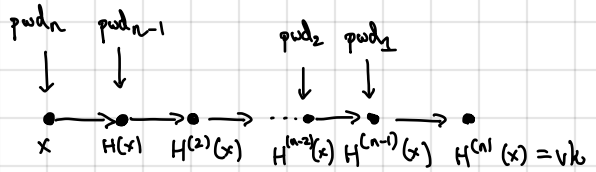
<u>Construction 1</u>: Consider setting where verification key vk is <u>secret</u> (e.g., server has a secret)
- Client and server have a shared PRF key k and a counter (initialized to 0):



- <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 windows)

If PRF is secure $\Rightarrow$ 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) ← "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 <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
- Naïvely, client has to evaluate H many times per authentication ($\approx O(n)$ times)
  ↳ Can reduce to $O(\log n)$ hash evaluations in an amortized sense by storing $O(\log n)$ entries along the hash chain