Alice -> give me your PK -> Bob

M

\text{Enc}(PK_B, m) -> m

\text{PK}_{\text{Adv}} \leftarrow PK_B

\text{Enc}(PK_{\text{Adv}}, m) \leftarrow \text{Enc}(PK_B, M)

\text{attacker MiTM}

\text{Trusted directory}

random
choosen
cy by Alice

Alice

PKTD
	hardcoded

in her
device

nonce me PK Bob ID

PK_B, sign(K_{MITM}, PK_B) (Bob's nonce)

MITM

\text{name} PK

\begin{tabular}{c|c}
Bob & 0x52 ... 0x2e \\
MITM & 0xae ...
\end{tabular}

\text{sign(PKTD)} is MITM

\text{Assume update happens securely}

\text{Updating a Key}

\text{Replay attack:}

Attacker replays old information (old sig with old PK)
Alice embeds nonce in her request checks sig from TD to contain nonce & to verify with PK_TD & contains Bob’s name => knows PK of Bob is latest

**Drawbacks of TD**

- Scalability (store & serve all PKs)
- TD is a central point of attack/trust
- Difficult to recover from TD compromise
- Updating key requires trust
- TD has to be always available
  - Central point of failure
Approach 2: Digital Certificates

$\rightarrow$ association between name & PK by a CA (certificate authority)
eg. Vensign

certificate: $\text{sign} (SK_{CA}, \text{Bob's PK is } \text{Ox51...}) = \text{cert}_{Bob}$

assume browsers have PK_{CA} hardcoded

[\text{+ anyone can serve PK}_{Bob}, \text{cert}_{Bob}]

Alice checks:
- cert_{Bob} verifies with PK_{CA}, is not expired, is for Bob

Alice no longer contacts TD to fetch PK_{Bob}, but can contact local server, e.g. Bob's Server
Alice

- give me your PK

PKB, cert

PKB

→

Bank.com

SKB, PKB

→

CA

wet = Sign(SKCA, Bank has PKB, expiry)

Adv

Better than TD:
+ can contact bank (or anyone) to obtain PK

Certificate hierarchies & chains

Versign CA

only certifies level underneath, presidents

UC President

David

Stanford president

Raluca

\[ C_1 = \text{sign}(SKCA, "UC Pres. has PKU", expiry) \]

Sign(SKU, "David has PKD", expiry)

Certificate chain

When I ask for David's PK, I will receive PKD, C1, C2,
- check PKU using C1 using PKCA
- check PKD using C2 and knowledge of PKU

David@berkeley.edu

Path of authority
Root servers serve cert. edu

Revocation

How can we revoke a certificate that has not yet expired?
- Wait till expiry, make expiry shorter
- Revocation lists: CA could push revocation
  - Sign (SKCA, "Revoked cert") into browsers; not ideal solution because browsers might not be downloading lists

Long-term problem with CAs:
- CAs can be compromised or could be deceived to sign incorrect certificates
- Transparency logs promise to address this problem
Passwords

Tension between usability and security

- choose memorable passwords
- choose random and long passwords (hard to guess)
Attack mechanisms

• Online guessing attacks
  – Attacker tries to login by trying different user passwords in the live system

• Social engineering and phishing
  – Attacker fools user into revealing password

• Eavesdropping
  – Network attacker intercepts plaintext password on the connection

• Client-side malware
  – Key-logger/malware captures password when inserted and sends to attacker

• Server compromise
  – Attacker compromises server, reads storage and learns passwords
Defences/mitigations

Network eavesdropper:
• Encrypt traffic using TLS (will discuss later)

Client-side malware: hard to defend
• Intrusion detection mechanisms – detect malware when it is being inserted into the network
• Various security software (e.g., anti-virus)
• Use two-factor authentication
Mitigations for online-guessing attacks

• Rate-limiting
  – Impose limit on number of passwords attempts

• CAPTCHAs: to prevent automated password guessing

• Password requirements: length, capital letters, characters, etc.
Mitigations for server compromise

• Suppose attacker steals the database at the server including all password information
• Storing passwords in plaintext makes them easy to steal
• Further problem: users reuse passwords at different sites!

Don’t store passwords in plaintext at server!
Hashing passwords

- Server stores hash(password) for each user using a cryptographic hash function
  - hash is a one-way function

<table>
<thead>
<tr>
<th>username</th>
<th>hash of password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>hash(Alice’s password)</td>
</tr>
<tr>
<td>Bob</td>
<td>hash(Bob’s password)</td>
</tr>
</tbody>
</table>

- When Alice logs in with password w (and provides w to server), server computes hash(w) and compares to Alice’s record
Password hashing: problems

• Offline password guessing
  – **Dictionary attack:** attacker tries all passwords against each hash(w)
  – If D is dictionary size, n number of hashes passwords, attack takes Dn
  – Study shows that a dictionary of \(2^{20}\) passwords can guess 50% of passwords

• Amortized password hashing
  – Idea: **One** brute force scan for all/many hashes (D+n time)
  – Build table \((H(password), password)\) for all \(2^{20}\) passwords
  – Crack 50% of passwords in this **one pass**
LinkedIn was storing $h(password)$

"Link" was the number one hacked password, according to Rapid7. But many other LinkedIn users also picked passwords — "work" and "job" for example — that were associated with the career site's content.

Religion was also a popular password topic — "god," "angel" and "jesus" also made the top 15. Number sequences such as "1234" and "12345" also made the list.
Prevent amortized guessing attack

• Randomize hashes with salt
• Server stores \((\text{salt, hash(password, salt)})\), salt is random
• Two equal passwords have different hashes now
• Dictionary attack still possible, BUT need to do one brute force attack per hash now, not one brute force attack for many hashes at once
• Attacks takes \(Dn\) time instead of \(D+n\) time
Salted hash example

<table>
<thead>
<tr>
<th>username</th>
<th>salt</th>
<th>hash of password</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>235545235</td>
<td>hash(Alice’s password, 235545235)</td>
</tr>
<tr>
<td>Bob</td>
<td>678632523</td>
<td>hash(Bob’s password, 678632523)</td>
</tr>
</tbody>
</table>

Attacker tries to guess Alice’s password:
Computes table
  ‘aaaaaa’   | hash(‘aaaaaa’, 235545235),
  ‘aaaaab’   | hash(‘aaaaab’, 235545235),
  …          |                                       
  ‘zzzzzzzz’ | hash(‘zzzzzzzz’, 235545235)            |

This table is useless for Bob’s password because of different salt
Increase security further

- Would like to slow down attacker in doing a dictionary attack
- Use **slow hashes** = takes a while to compute the hash
- Define
  \[ H(x) = \text{hash(hash(hash(...hash(x))))} \]
  use with \( x = \text{password || salt} \)

- Tension: time for user to authenticate & login vs attacker time
- If \( H \) is 1000 times slower and attack takes a day with \( H \), attack now takes 3 years with \( F \)
Conclusions

- Do not store passwords in cleartext
- Store them hashed with salts, slower hash functions better