Announcement

Project 2 to be released Thursday
Midterm grades announced at end of week
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 SSL (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($w$)
  – 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
  – Build table $(H(\text{password}), \text{password})$ 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
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>‘aaaaaa’</td>
<td>hash(‘aaaaaa’, 235545235), hash(‘aaaaaab’, 235545235),</td>
</tr>
<tr>
<td>‘aaaaab’</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
</tr>
<tr>
<td>‘zzzzzzz’</td>
<td>hash(‘zzzzzz’, 235545235)</td>
</tr>
</tbody>
</table>

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
Alice

How does Alice know Bob's PK?

Bob (Bank of America)

give me your PK

Enc[PK_b, K]

Shared secret

Alice

Bob

Plaintext

PK_A

Attacker

Attacker

PK_b

obtain Sk_b

Key management

10/51
How can Alice obtain the PK of Bob securely?

Public-key Infrastructure (PKI) = infrastructure (roles, policies, protocols) for managing PK and certificates

1) Trusted Directory Service

- Everyone knows PK (Alice has PK hardcoded in her browser)

Sol1: MITM attacker can return a Sybil from TD for someone
Problem: If Alice doesn’t know if PK is Bob’s

Sol2: TD answers with sign(PK₂). Doesn’t know if PK is Bob’s
2) PKI Approach 2: Digital Certificates.

association between a username and their public as certified by some authority (e.g. Verisign)

CA = certificate authority (e.g. Verisign)

certificate = sign $sk_{CA}$ (username, $PK$)

Users verify certificates using $PK_{CA}$ hardcoded in browsers

Ex: sign $sk_{Verisign}$ (Google ID: $PK_{Google}$)
Suppose Bob's SKB was compromised → Attacker has SKB. Bob generates new (PKB', SKB') and updates TD so it contains [Bob, BKB'] in public.

Problem with Sol 3: Attacker becomes MITM and replaces TD's response containing PKB' with old response with PKB. (Replaying old information)
Alice → Hey, Bob, give me PK → Bob

PK_B, sign_C A_Bob, PK_B

can obtain certificate from anyone
+ CA does not have to be online.

What if Bob changes PK? - add expiry to certificate

certificate: sign_{SK_{CA}} (Bob, PK_B, expiry)

April, 2018

When expires, Bob needs to obtain new certificate from CA with new expiry date

Prevents replay only across expiry periods
If Bob's SK is compromised before expiry, Attacker can impersonate Bob till then
Sel 4:

Alice
- chooses nonce randomly

verify:
1) signature verifies using PKT
2) signature has Bob’s username
3) contains nonce Alice sent

Drawbacks: central point of attack and failure
TD has to be online always
Not scalable: has to know everyone’s PK