Life of a Password

Arvind Mani
Data & Infrastructure Security
LinkedIn
Secure user account.
Agenda

We cover the following topics:

- Hashing
- Transport
- Storage

Not covered:

monitoring, host and network security, access control, other account protection mechanisms.
Unsalted Password Hash

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>F(&quot;monkey&quot;)</td>
</tr>
<tr>
<td>m2</td>
<td>F(&quot;123456&quot;)</td>
</tr>
</tbody>
</table>

- Brute force short passwords
- Dictionary attack
- Rainbow Table
## Salted Password Hashes

### Cheap Salt (performance; what’s wrong?)

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>F(“monkey”, m1)</td>
</tr>
<tr>
<td>m2</td>
<td>F(“123456”, m2)</td>
</tr>
</tbody>
</table>

### Random Salt

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
<th>salt (64/96 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>F(“Jl6aerwhm”, s1)</td>
<td>s1</td>
</tr>
<tr>
<td>m2</td>
<td>F(“$^%YRTYFYU”, s2)</td>
<td>s2</td>
</tr>
</tbody>
</table>

**Susceptible to Targeted attack**
Keyed Crypto Hash (MAC)

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>F(“JI6aerwhm”, “secret”)</td>
</tr>
<tr>
<td>m2</td>
<td>F(“$^%YRTYFYU”, “secret”)</td>
</tr>
</tbody>
</table>

- Prevents dictionary attack
- Common passwords are revealed
- Prevents targeted attack

Next: Online attacks
Overwrite Attack

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
<th>salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>$F(\text{“ashjrgqw3nk”}, s1)$</td>
<td>s2</td>
</tr>
<tr>
<td>m2</td>
<td>$F(\text{“%RYThj#WY”}, s2)$</td>
<td>s2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
<th>salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>$F(\text{“password”}, s)$</td>
<td>s</td>
</tr>
<tr>
<td>m2</td>
<td>$F(\text{“password”}, s)$</td>
<td>s</td>
</tr>
</tbody>
</table>

Attacker overwrites m1 and m2’s real passwords
Swap Attack

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
<th>salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>F(&quot;password&quot;, s1, &quot;secret&quot;)</td>
<td>s1</td>
</tr>
<tr>
<td>m2</td>
<td>F(&quot;$%^YRTYFYU&quot;, s2, &quot;secret&quot;)</td>
<td>s2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
<th>salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1 (attacker)</td>
<td>F(&quot;password&quot;, s1, &quot;secret&quot;)</td>
<td>s1</td>
</tr>
<tr>
<td>m2 (victim)</td>
<td>F(&quot;password&quot;, s1, &quot;secret&quot;)</td>
<td>s1</td>
</tr>
</tbody>
</table>
Keyed Hash

Pros
• One-way
• Correlated input secure

Cons
• Hash computation is fast
• Fixed input, fixed output
Password Recipe

- Key Derivation Function (KDF) instead of crypto hash
- Random salt
- User or member id
- Work factor (active accounts)
- Application secret
- Encrypted Hashes vs MAC
Ongoing Key Rotation

• Increase likelihood that not all stored credentials can be cracked.
• You have fingerprinted your database – stolen hashes can pinpoint “when”
### Password History Table

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
<th>salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>F(&quot;password&quot;, s1, m1, &quot;secret&quot;)</td>
<td>s1</td>
</tr>
</tbody>
</table>

### Password Table

<table>
<thead>
<tr>
<th>User id</th>
<th>hash</th>
<th>salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>F(&quot;^%$TRsfwe&quot;, s2, m1, &quot;secret&quot;)</td>
<td>s2</td>
</tr>
</tbody>
</table>
Accidental Logging

proxy Tier(s) → login/registration frontend → Login backend systems

 Logs

2014/02/25 18:38:55.751 [(prod-host1,login-app,/login,2014/02/25 18:38:55.572) verifyPassword(email="foobart@yahoo.com", password=monkey, ip_address="1.1.1.1"), PASS, 11ms
• TLS throughout, so on network password is always encrypted
• Each hop sees password in clear – potential for improper handling
Fix at User Agent – Attempt 1

Send Hash(password)

- Equation in Step 3 holds if the sha1 is done consistently during registration, password reset, etc
- **Problem** – hashed password log is as bad as logging cleartext password!
Send Hash(password, salt)

1. Fix at User Agent – Attempt 2

2. read entry for user1, say Y

3. Is Y == AES(PBKDF2(H'))?

- Salt used in computing Y using PBKDF2 can’t be same as rand1
- **Problem** - Equation in Step 3 can’t hold for any verification, scheme not feasible
Fix At User Agent – Attempt 3

Send `PublicKeyEncryption(password)`

**Problem** – Can replay and use encrypted password instead of real password to login as user
Fix At User Agent - Summary

Send PublicKeyEncryption(password) + nonce

• Good news – this finally works!
• Bad news – must support all user agents including native mobile, some clients can’t be upgraded
Fix at Ingress – Attempt 1

Instead of sending password, over TLS send either:

- `Hash(password, salt)`
- Password token not derived from password
Fix at Ingress – Attempt 2

PublicKeyEncryption(password)

No replay from outside, can replay from inside network
Cloaked Password

- Password encrypted PublicKey_{loginserver}
- Ciphertext is randomized
- Replay protection via short expiry or nonce infrastructure
- Can be decrypted only by verification end point
Storage

- SQL injection

- Attacker has username/password of database
- Attacker has access to filesystem
Dump credentials

• SQL injection (nosql stores are not by default safe)

password='foo' or 1=1 --

• Attacker with DB credential

Attacker on production host

select * from credential_table;
Centralizing Storage

• Many types of credentials – isolate application credentials

• Single point of attack

Diagram:
- Login
- API (pwned)
- Credential DB
  - Passwords
  - OAuth tokens
  - All credentials
Credential Store

- Access via Stored Procedure
- Isolate client data via dual encryption
- Access Control
- Auditing
- Monitoring
- Periodic key rotation
Credential Store

- All communication over TLS
- ACLs on operations
- Client encryption

id1, \( Y = E_{\text{login}}(\text{password}) \)

id1, \( Y = E_{\text{api}}(\text{accessToken}) \)

id1, \( Z = E_{\text{cred}}(Y) \)

- listener access IP restricted to cred service
- Access via Stored procs
Summary

• Made some progress securing passwords
• Re-usable infrastructure – apply to credit cards, OAuth tokens, etc
• Future Work – Key Management, SRP?, mitigate risk of compromise of critical applications
Acknowledgement

We want to thank Professor Dan Boneh, Applied Crypto Group, Stanford University for his help with password hashing scheme.
Questions?

amani@linkedin.com
https://www.linkedin.com/in/arvindmani