6.033 Spring 2016
Lecture #21

- Principal Authentication via Passwords
**complete mediation:** every request for resource goes through the guard

![Diagram of request flow from principal to resource through guard]

**guard typically provides:**

- **authentication:** is the principal who they claim to be?
- **authorization:** does principal have access to perform request on resource?
username | password
arya    | valarMorghul1s
jon     | w1nterIsC0ming
sansa   | LemonCakesForever
hodor   | hodor

check_password(username, inputted_password):
    stored_password = accounts_table[username]
    return stored_password == inputted_password

**Problem:** adversary with access to server can get passwords
check_password(username, inputted_password):
    stored_hash = accounts_table[username]
    inputted_hash = hash(inputted_password)
    return stored_hash == inputted_hash

<table>
<thead>
<tr>
<th>username</th>
<th>hash(password)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arya</td>
<td>de5aba604c340e1965bb27d7a4c4ba03f4798ac7</td>
</tr>
<tr>
<td>jon</td>
<td>321196d4a6ff137202191489895e58c29475ccab</td>
</tr>
<tr>
<td>sansa</td>
<td>6ea7c2b3e08a3d19fee5766cf9fc51680b267e9f</td>
</tr>
<tr>
<td>hodor</td>
<td>c6447b82fbb4b8e7dbcf2d28a4d7372f5dc32687</td>
</tr>
</tbody>
</table>
top 10 passwords from a leak of 32 million passwords in 2009


<table>
<thead>
<tr>
<th>password</th>
<th>number of users</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456</td>
<td>290,731</td>
</tr>
<tr>
<td>12345</td>
<td>79,078</td>
</tr>
<tr>
<td>123456789</td>
<td>76,790</td>
</tr>
<tr>
<td>Password</td>
<td>61,958</td>
</tr>
<tr>
<td>iloveyou</td>
<td>51,622</td>
</tr>
<tr>
<td>princess</td>
<td>35,231</td>
</tr>
<tr>
<td>rockyou</td>
<td>22,588</td>
</tr>
<tr>
<td>1234567</td>
<td>21,726</td>
</tr>
<tr>
<td>12345678</td>
<td>20,553</td>
</tr>
<tr>
<td>abc123</td>
<td>17,542</td>
</tr>
</tbody>
</table>

password usage has not improved in recent years. see, e.g.,
http://adamcaudill.com/2012/07/12/yahoos-associated-content-hacked/
http://www.huffingtonpost.com/2012/06/08/linkedin-password-leak-infographic_n_1581620.html
check_password(username, inputted_password):
    stored_hash = accounts_table[username]
    inputted_hash = hash(inputted_password)
    return stored_hash == inputted_hash

**problem:** adversary can create rainbow table
### check_password

```python
check_password(username, inputted_password)

    stored_hash = accounts_table[username]
    inputted_hash = hash(inputted_password | salt)
    return stored_hash == inputted_hash
```

<table>
<thead>
<tr>
<th>username</th>
<th>salt</th>
<th>hash(password</th>
<th>salt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arya</td>
<td>5334900209</td>
<td>c5d2a9fff6052a27e6183d60321c44c58c3c26cc</td>
<td></td>
</tr>
<tr>
<td>jon</td>
<td>1128628774</td>
<td>624f0ffa577011e5704bdf0760435c6ca69336db</td>
<td></td>
</tr>
<tr>
<td>sansa</td>
<td>8188708254</td>
<td>5ee2b8effce270183ef0f4c7d458b1ed95c0ccee5</td>
<td></td>
</tr>
<tr>
<td>hodor</td>
<td>6209415273</td>
<td>f7e17e61376f16ca23560915b578d923d86e0319</td>
<td></td>
</tr>
</tbody>
</table>
how can we avoid transmitting the password over and over?

session cookies

\{\text{username}, \text{expiration}, \text{H}(\text{server\_key} \mid \text{username} \mid \text{expiration})\}
how can we protect against phishing attacks, where an adversary tricks a user into revealing their password?

must avoid sending the password to the server entirely, but still allow valid servers to authenticate users.
challenge-response protocol

(username | password)
arya | valarMorghulis
jon | w1nterIsC0ming
sansa | LemonCakesForever
hodor | hodor

server computes
H(valarMorghulis | 458643)
and checks

(random number)
458653

= H(valarMorghulis | 458643)

password is never sent directly
challenge-response protocol

(username | password)

adversary-owned server

password is never sent directly

random number

H(valarMorghulis | 458643)

= afe7b4bb57ab6e6226b5205f51b95fc94fef9f45

adversary only learns
H(valarMorghulis | 458643); can’t recover the password from that
challenge-response protocol

valid server

<table>
<thead>
<tr>
<th>username</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>arya</td>
<td>valarMorghul1s</td>
</tr>
<tr>
<td>jon</td>
<td>w1nterIsC0ming</td>
</tr>
<tr>
<td>sansa</td>
<td>LemonCakesForever</td>
</tr>
<tr>
<td>hodor</td>
<td>hodor</td>
</tr>
</tbody>
</table>

server computes
H(valarMorghul1s | 458643)
and checks

(random number)
458653

afe7b4bb57ab6e6226b5205f51b95fc94fef9f45
= H(valarMorghul1s | 458643)

password is never sent directly

adversary-owned servers (that don’t know passwords) won’t learn the password; client never sends password directly

problems arise when the server stores (salted) hashes — as it should be doing — but there are challenge-response protocols that handle that case
how can we protect against phishing attacks, where an adversary tricks a user into revealing their password?

must avoid sending the password to the server entirely, but still allow valid servers to authenticate users.
how do we initially set (bootstrap) or reset a password?
• Using passwords securely takes some effort. Storing **salted hashes**, incorporating **session cookies**, dealing with **phishing**, and **bootstrapping** are all concerns.

• Thinking about how to use passwords provides more **general lessons**: consider human factors when designing secure systems, be explicit, small improvements are worthwhile, etc.

• There are always **trade-offs**. For instance, all of the methods discussed today add security, but also complexity.