Why Care about Security?

Security
- Confidentiality – keeping secrets
  - Protect user’s data
- Integrity – making data reliable
  - Preventing tampering
  - Only authorized people can insert/modify data
- Availability
  - Provide service (even when attacked)
  - Can’t do much about this without resources

How do you authenticate?
- Something you know
  - Password
- Something you have
  - Physical key (email account?, transparency?)
- Something you are
  - Biometrics (voiceprint, fingerprint, etc.)

Serious authentication requires at least 2 kinds

Early Password Schemes

<table>
<thead>
<tr>
<th>UserID</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alyssa</td>
<td>fido</td>
</tr>
<tr>
<td>ben</td>
<td>schemer</td>
</tr>
<tr>
<td>dave</td>
<td>Lx.Ly.x</td>
</tr>
</tbody>
</table>

Login does direct password lookup and comparison.

Login: alyssa
Password: spot
Failed login. Guess again.

Login Process

Terminal
Login: alyssa
Password: fido
login sends
"alyssa", "fido"

Trusted Subsystem

Eve
Password Problems

• Need to store the passwords
  – Dangerous to rely on database being secure  [Solve this today]

• Need to transmit password from user to host
  – Dangerous to rely on Internet being confidential  [Solve this Wednesday]

First Try: Encrypt Passwords

• Instead of storing password, store password encrypted with secret $K$.
• When user logs in, encrypt entered password and compare to stored encrypted password.

<table>
<thead>
<tr>
<th>UserID</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alyssa</td>
<td>encrypt_K(“fido”)</td>
</tr>
<tr>
<td>ben</td>
<td>encrypt_K(“schemer”)</td>
</tr>
<tr>
<td>dave</td>
<td>encrypt_K(“Lx.Ly.x”)</td>
</tr>
</tbody>
</table>

Problem if $K$ isn’t so secret: $\text{decrypt}_K(\text{encrypt}_K(P)) = P$

Hashing

- Many-to-one: maps a large number of values to a small number of hash values
- Even distribution: for typical data sets, probability of $(H(x) = n) = 1/N$ where $N$ is the number of hash values and $n = 0...N-1$.
- Efficient: $H(x)$ is easy to compute.

Example One-Way Function

Input: two 100 digit numbers, $x$ and $y$
Output: the middle 100 digits of $x \times y$

Given $x$ and $y$, it is easy to calculate

$f(x, y) = \text{select middle 100 digits } (x \times y)$

Given $f(x, y)$ hard to find $x$ and $y$.

Cryptographic Hash Functions

One-way

Given $h$, it is hard to find $x$ such that $H(x) = h$.

Collision resistance

Given $x$, it is hard to find $y \neq x$ such that $H(y) = H(x)$.

A Better Hash Function?

• $H(x) = \text{encrypt}_K(0)$
• Weak collision resistance?
  – Given $x$, it should be hard to find $y \neq x$ such that $H(y) = H(x)$.
  – Yes – encryption is one-to-one. (There is no such $y$.)
• A good hash function?
  – No, its output is as big as the message!
Actual Hashing Algorithms

- Based on cipher block chaining
  - Start by encrypting 0 with the first block
  - Use the next block to encrypt the previous block
- SHA [NIST95] – 512 bit blocks, 160-bit hash
- MD5 [Rivest92] – 512 bit blocks, produces 128-bit hash
  - This is what we will use: built in to PHP

Hashed Passwords

<table>
<thead>
<tr>
<th>UserID</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alyssa</td>
<td>md5(“fido”)</td>
</tr>
<tr>
<td>ben</td>
<td>md5(“schemer”)</td>
</tr>
<tr>
<td>dave</td>
<td>md5(“Lx.Ly.x”)</td>
</tr>
</tbody>
</table>

Dictionary Attacks

- Try a list of common passwords
  - All 1-4 letter words
  - List of common (dog) names
  - Words from dictionary
  - Phone numbers, license plates
  - All of the above in reverse
- Simple dictionary attacks retrieve most user-selected passwords
- Precompute $H(x)$ for all dictionary entries

(at least) 86% of users are dumb and dumber

<table>
<thead>
<tr>
<th>Type of Password</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single ASCII character</td>
<td>0.5%</td>
</tr>
<tr>
<td>Two characters</td>
<td>2%</td>
</tr>
<tr>
<td>Three characters</td>
<td>14%</td>
</tr>
<tr>
<td>Four alphabetic letters</td>
<td>14%</td>
</tr>
<tr>
<td>Five same-case letters</td>
<td>21%</td>
</tr>
<tr>
<td>Six lowercase letters</td>
<td>18%</td>
</tr>
<tr>
<td>Words in dictionaries or names</td>
<td>15%</td>
</tr>
<tr>
<td>Other (possibly good passwords)</td>
<td>14%</td>
</tr>
</tbody>
</table>

(Sources: Morris/Thompson 79)

Salt of the Earth

(This is the standard UNIX password scheme.)

<table>
<thead>
<tr>
<th>User ID</th>
<th>Salt</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alyssa</td>
<td>1125</td>
<td>DES+ (0, “Lx.Ly.x”, 1125)</td>
</tr>
<tr>
<td>ben</td>
<td>2437</td>
<td>DES+ (0, “schemer”, 2437)</td>
</tr>
<tr>
<td>dave</td>
<td>932</td>
<td>DES+ (0, “Lx.Ly.x”, 932)</td>
</tr>
</tbody>
</table>

DES+ (m, key, salt) is an encryption algorithm that encrypts in a way that depends on the salt.

How much harder is the off-line dictionary attack?

Python Code

```python
// We use the username as a "salt" (since they must be unique)
encryptedpass = crypt.crypt (password, user)
```

<table>
<thead>
<tr>
<th>user</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alyssa</td>
<td>9928ef0d7a0e4759ffe7ad8b8bc84228</td>
</tr>
<tr>
<td>evans</td>
<td>baf72c60f450ed665a6eadc92b3647f</td>
</tr>
</tbody>
</table>
Authenticating Users

- User proves they are a worthwhile person by having a legitimate email address
  - Not everyone who has an email address is worthwhile
  - It's not too hard to snoop (or intercept) someone's email
- But, provides much better authenticating than just the honor system

Registering for Account

- User enters email address
- Sent an email with a temporary password

```python
rnd = str(random.randint(0, 9999999))
encrnd = crypt.crypt(rnd, str(random.randint(0, 99999)))
users.userTable.createUser(user, email, firstnames, lastname, encrnd)
```

Users and Passwords

```python
def createUser(self, user, email, firstnames, lastname, password):
c = self.db.cursor()
encpwd = crypt.crypt(password, user)
query = "INSERT INTO users (user, email, firstnames, lastname, password) " + "VALUES ('" + user + "', " + email + ", " + firstnames + ", " + lastname + ", " + encpwd")"
c.execute(query)
self.db.commit()
```

Cookies

- HTTP is stateless: every request is independent
- Don’t want user to keep having to enter password every time
- A cookie is data that is stored on the browser’s machine, and sent to the web server when a matching page is visited

Problems Left

- The database password is visible in plaintext in the Python code
  - No way around this (with UVa mysql server)
  - Anyone who can read UVa filesystem can access your database
- The password is transmitted unencrypted over the Internet (next class)
- Proving you can read an email account is not good enough to authenticate for important applications
Charge

- Authentication
  - At best, all this does is check someone can read mail sent to a .virginia.edu email address
- Find PS8 partners now!