Engineering Robust Server Software

Security

Significant portions based on slides from
Micah Sherr @ Georgetown
Security

• First major topic: security
  • What do we mean by security?
Security

• First major topic: security
  • What do we mean by security?

• **Confidentiality**: things are kept secret
• **Integrity**: things cannot be tampered with
• **Availability**: things are useable (for users who are supposed to be able to)

• A key topic that helps all of the above:
  • **Authentication**: things can figure out that you are who you claim to be
Confidentiality: Example 1

Leftover Food in HH 218

Alice → Eve → Bob
Confidentiality: Example 1

\[ f(\text{Leftover Food in HH 218}) = \text{Al481manj417a@#1naL} \]

\[ f^{-1}(\text{Al481manj417a@#1naL}) = \text{Leftover Food in HH 218} \]
Confidentiality: Example 2

[eve@linux] $ cat ~alice/secret.txt
ls: /home/alice/secret.txt : Permission denied
[eve@linux] $
Integrity: Example 1

Alice

Please send $1000 to account 123456
Thanks, Alice

Eve

Please send $1000 to account 467129
Thanks, Alice

Bob
Integrity: Example 2

Eve's awesome software services! 20% off. This week only!
Security Difficulty: Hard To Detect Compromises

Is my browser hacked??

Alice
Confidentiality + Integrity

Physical attacks:
- Hardware support for defense: e.g., SGX
Availability: Example 1

- **Distributed Denial of Service (DDoS)**
- Attacker gets a bunch of compromised machines
- Tells all of them to visit victim site
- Victim has more traffic than it can handle
- Legitimate users can’t access the site – too slow/crashed
Availability: Example 2

- Attacker finds flaw in your website – now they can run any database commands they want (SQL injection attack)
- One cheap and destructive thing they can do: 
  ```sql
  DROP DATABASE site_database;
  ```
- Site loses all database data (unless it was backed up – covered later)
Hi I'm Alice

Please buy 1000 shares of foobar corp with the money in my account

Wait... Are you really Alice?
...Prove it!
Authentication

• Kinds of authentication:
  • Something you **know** (e.g., password)
  • Something you **have** (e.g., your phone)
  • Something you **are** (biometrics)

• Multi-factor:
  • When you use two or more of the above categories
Authentication

- Two common and relevant modes of authentication for servers:
  - **Password** (still most common, sadly)
    - How good are your passwords?
  - **Cryptographic keys**
    - ssh keys
Authentication scenario: sent plaintext password

username: alice
password: m$1iKa^PQ1t#aRn7

• What would happen if Alice sent her password cleartext?
Authentication scenario: sent plaintext password

username: alice
password: m$1iKa^PQ1t#aRn7

Alice

Now I know Alice's password!

Eve
Authentication scenario: stored plaintext password

(encrypted username/password)

- Ok, so Alice encrypts username/password...
Authentication scenario: stored plaintext password

(encrypted username/password)

<table>
<thead>
<tr>
<th>Username</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>m$1iKa^PQ1t#aRn7</td>
</tr>
<tr>
<td>bob</td>
<td>lWi8@P(~qtY3oR</td>
</tr>
<tr>
<td>eve</td>
<td>,$ql)Pq&gt;4iQldV7</td>
</tr>
</tbody>
</table>

- VERY BAD: DO NOT DO
  - Server stores password in plaintext
Authentication scenario: stored plaintext password

(encrypted username/password)

- VERY BAD: DO NOT DO
  - Server stores password in plaintext

<table>
<thead>
<tr>
<th>Username</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>m$1iKa^PQ1t#aRn7</td>
</tr>
<tr>
<td>bob</td>
<td>lWi8@P(~qtY3oR</td>
</tr>
<tr>
<td>eve</td>
<td>,$ql)Pq&gt;4iQIdV7</td>
</tr>
</tbody>
</table>

Sweet! Now I have everyone's password... Maybe they use them on other sites too!....
Authentication scenario: stored hashed password

(encrypted username/password)

- Better, but still wrong
  - Server stores hashes: computes $\text{hash(password)}$ + compares to table

<table>
<thead>
<tr>
<th>Username</th>
<th>PasswdHash</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>A1C399F501AB5432</td>
</tr>
<tr>
<td>bob</td>
<td>F09485ACB154A</td>
</tr>
<tr>
<td>eve</td>
<td>154FABC9F523C0</td>
</tr>
</tbody>
</table>
Authentication scenario: stored hashed password

(encrypted username/password)

Now I have hashes… What can I do with them?

- Better, but still wrong
  - Server stores hashes: computes hash(password) + compares to table

<table>
<thead>
<tr>
<th>Username</th>
<th>PasswdHash</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>A1C399F501AB5432</td>
</tr>
<tr>
<td>bob</td>
<td>F09485ACB154A</td>
</tr>
<tr>
<td>eve</td>
<td>154FABC9F523C0</td>
</tr>
</tbody>
</table>
Len = 1
while (1) {
    for each string s of length Len{
        int h = hash(s)
        users = mapOfStolenData.lookup(h);
        if (users is not empty) {
            print users + "password =" + s
        }
    }
    Len ++;
}
Difficulty to Crack?

- Generally 95 possible characters
- Number of possible strings for a given length:

<table>
<thead>
<tr>
<th>Length</th>
<th>Num Strings $(95^L)$</th>
<th>Time if 500K hash/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>81 M</td>
<td>2 minutes</td>
</tr>
<tr>
<td>6</td>
<td>735 B</td>
<td>17 days</td>
</tr>
<tr>
<td>8</td>
<td>6E+15</td>
<td>400 years</td>
</tr>
<tr>
<td>10</td>
<td>6E+19</td>
<td>3M years</td>
</tr>
</tbody>
</table>

Is 8 characters safe?
Len = 1
while (1) {
    for each string s of length Len{
        int h = hash(s)
        users = mapOfStolenData.lookup(h);
        if (users is not empty) {
            print users + "password =" + s
        }
    }
    Len ++;
}

Eve can speed up her attack by exploiting the fact that...

...this code can easily be parallelized
Difficulty to Crack?

- Generally 95 possible characters
- Number of possible strings for a given length:

<table>
<thead>
<tr>
<th>Length</th>
<th>Num Strings (95^L)</th>
<th>Time if 500K hash/sec</th>
<th>Time if 1.5T hash/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>81 M</td>
<td>2 minutes</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>6</td>
<td>735 B</td>
<td>17 days</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>8</td>
<td>6E+15</td>
<td>400 years</td>
<td>~1 hour</td>
</tr>
<tr>
<td>10</td>
<td>6E+19</td>
<td>3M years</td>
<td>~1 year</td>
</tr>
</tbody>
</table>

I just bought 100 GPUs..
Password Cracking

• That analysis is for
  • Every possible password (every combination)
  • Gets ALL stolen hashes at once
    • 10,000 users
    • 1 hour: every password of length 8 (thousands of passwords)

• Can probably speed up by using common passwords
  • password
  • 1234
  • …
Pre-Computation

I'm going to try to hack your server soon. But once you discover it, you might warn your users, and they might change their passwords... How can I prepare?
Password Cracking (cont'd)

- Can also trade time for space
  - Execution time/memory (or disk) requirement tradeoff
- Option 1 build map hash -> password (e.g., before stealing hashes)
  - \(95^6 \times 16 \text{ bytes/entry} \approx 10 \text{ TB}\) [HDD: costs about $300—$400]
  - \(95^8 \times 16 \text{ bytes/entry} \approx 96,540 \text{ TB}\) [seems expensive]
- Option in the middle?
  - Pre-compute some things
  - Make attack faster
  - Do not require so much storage?
Rainbow Tables

The **reduce**: Convert a hash to a possible password.

Note: no chance that the password hashes to the given value! This isn’t a “reverse the hash” function, since that’s our overall goal!

Just an arbitrary mapping so that the set of all strings in all the chains (the first one plus all the reduces, for all the chains) represent most/all of the password space.
Rainbow Tables

Hash Hash Hash Hash

1BFC190C2 fgr31 FC45019AB

Reduce Reduce Reduce .......

Hash Hash Hash Hash

92A3051B3 lo984 001324AC3

Reduce Reduce Reduce .......

Hash Hash Hash Hash

09B8501C2 pq!<4 985AB3021

Reduce Reduce Reduce .......

Hash Hash Hash Hash

fog42 fo984 fo0124AC3

Reduce Reduce Reduce .......

Hash Hash Hash Hash

xyzzy pq!<4 pq0124AC3

Reduce Reduce Reduce .......

Hash Hash Hash Hash

Rainbow Tables
Rainbow Tables

cat42  FC45019AB
frog1  001324AC3
xyzzy  985AB3021
Rainbow Tables

hashed password: 9A07135CB

157645A39

hashed password: 9A07135CB

in table?

No: keep going

Rainbow Table

cat42 FC45019AB
frog1 001324AC3
xyzzy 985AB3021

FC45019AB
Rainbow Tables

Hashed Password:

9A07135CB

Rainbow Table

<table>
<thead>
<tr>
<th>Password</th>
<th>Hashed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat42</td>
<td>FC45019AB</td>
</tr>
<tr>
<td>frog1</td>
<td>001324AC3</td>
</tr>
<tr>
<td>xyzzy</td>
<td>985AB3021</td>
</tr>
</tbody>
</table>
Rainbow Tables

This is the password

9A07135CB = hashed password
Rainbow Tables

- If we have C chains of length L, and the password is in one of our chains, then a rainbow table lets us break the password in
  - A: \( O(\lg(C) \times L^2) \) time
  - B: \( O(C^2 \times \lg(L)) \) time
  - C: \( O(L) \) time
  - D: \( O(C \times L) \) time

(assume linear search for the rainbow table)
Space vs Time

• Full map
  • $95^6 \times 16$ bytes/entry $\approx 10$ TB [HDD: costs about $500$]
  • $95^8 \times 16$ bytes/entry $\approx 96,540$ TB [seems expensive]

• Rainbow table (w/ 1B hashes/chain):
  • $95^6 \times 16$ bytes/entry $\approx 10$ KB [fits in L1 cache]
  • $95^8 \times 16$ bytes/entry $\approx 96$MB [fits in RAM]
  • $95^{10} \times 16$ bytes/entry $\approx 830$ GB [cheap hard disk]

Important Lesson: HASHING IS NOT ENOUGH
Authentication scenario: storing **salted** hashed passwords

**Salt**: A bit of per-user random extra data we include in the hash

(encrypted username/password)

<table>
<thead>
<tr>
<th>Username</th>
<th>PasswdHash</th>
<th>Salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>A1C399F501AB</td>
<td>1A45FB9C072B</td>
</tr>
<tr>
<td>bob</td>
<td>F09485ACB154</td>
<td>9841ABCD4167</td>
</tr>
<tr>
<td>eve</td>
<td>154FABC9F523</td>
<td>FAB981230CDB</td>
</tr>
</tbody>
</table>

- This is the correct approach (assuming we get everything else right)
  - Server stores hashes + salt: computes hash(password, salt)
Speaking of Authentication...

<table>
<thead>
<tr>
<th>Username</th>
<th>PasswdHash</th>
<th>Salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>A1C399F501AB</td>
<td>1A45FB9C072B</td>
</tr>
<tr>
<td>bob</td>
<td>F09485ACB154</td>
<td>9841ABCD4167</td>
</tr>
<tr>
<td>eve</td>
<td>154FABC9F523</td>
<td>FAB981230CDB</td>
</tr>
</tbody>
</table>

To crack Alice's password: try various combinations of strings (s) hash (s, 0x1A45FB9C072B)

To crack Bob's password: try various combinations of strings (s) hash (s, 0x9841ABCD4167)
What Does Salt Get Us?

- Pre-computation is ineffective
  - Build a map for each possible salt?
    - 64 bit salt -> will take forever + be huge
  - Rainbow tables?
    - Still need rainbow table for each salt
    - Expensive to build/store
- Crack each user's password *separately*
  - Rather than in parallel
  - Slowdown factor of number of users
What Does Salt Get Us?

Now this analysis is for **ONE** user's password (not all at once)
Multiply by number of user's to do them all…

<table>
<thead>
<tr>
<th>Length</th>
<th>Num Strings ((95^L))</th>
<th>Time if 500K hash/sec</th>
<th>Time if 1.5T hash/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>81 M</td>
<td>2 minutes</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>6</td>
<td>735 B</td>
<td>17 days</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>8</td>
<td>6E+15</td>
<td>400 years</td>
<td>~ 1 hour</td>
</tr>
<tr>
<td>10</td>
<td>6E+19</td>
<td>3M years</td>
<td>~ 1 year</td>
</tr>
</tbody>
</table>
Authentication scenario: storing salted hashed passwords

**Salt:** A bit of per-user random extra data we include in the hash

(encrypted username/password)

<table>
<thead>
<tr>
<th>Username</th>
<th>PasswordHash</th>
<th>Salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>A1C399F501AB</td>
<td>1A45FB9C072B</td>
</tr>
<tr>
<td>bob</td>
<td>F09485ACB154</td>
<td>9841ABCD4167</td>
</tr>
<tr>
<td>eve</td>
<td>154FABC9F523</td>
<td>FAB981230CDB</td>
</tr>
</tbody>
</table>

- This is the correct approach (assuming we get everything else right)
- Server stores hashes + salt: computes hash(password, salt)
Speaking of Authentication...

- What else do we need to get right?
  - Sufficiently long (>=64 bits), random salt
  - Secure hash: SHA-2 or SHA-3
    - *Not* MD5, SHA-0, or SHA-1
  - Use key stretching algorithm
    - E.g., PBKDF2 (also popular: bcrypt)
Key Stretching

- Do we want hashing algorithm to be slow or fast?
  - A: slow
  - B: fast
Key Stretching

- Do we want hashing algorithm to be **slow** or **fast**?

<table>
<thead>
<tr>
<th>Length</th>
<th>Num Strings (95^L)</th>
<th>Time if 500K hash/sec</th>
<th>Time if 1.5T hash/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>81 M</td>
<td>2 minutes</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>6</td>
<td>735 B</td>
<td>17 days</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>8</td>
<td>6E+15</td>
<td>400 years</td>
<td>~1 hour</td>
</tr>
<tr>
<td>10</td>
<td>6E+19</td>
<td>3M years</td>
<td>~ 1 year</td>
</tr>
</tbody>
</table>

**Slow :)**  **Fast :(**
Key Stretching

• Do we want hashing algorithm to be **slow** or **fast**?
  • Want attackers to have to spend more work to break hashes
• If we just hash...

password = m$1iKa^PQ1t#aRn7

```
salt = 1A45FB9C072BC90A
```

```
hash = ...
```

Total computation
Key Stretching

- PBKDF2

password = m$1iKa^PQ1t#aRn7

salt = 1A45FB9C072BC90A50C30000

Total computation
Speaking of Authentication...

• What else do we need to get right?
  • Sufficiently long (>=64 bits), random salt
  • Secure hash: SHA-2 or SHA-3
    • Not MD5, SHA-0, or SHA-1
  • Use key stretching algorithm
    • E.g., PBKDF2 (also popular: bcrypt)
  • Do NOT screw up and weaken things:
  • Do NOT try to invent things yourself!
Use Libraries That Do It Right

• Hashing in C/C++?
  • Use libssl

• Hashing in python?
  • Use `hashlib.pbkdf2_hmac('sha512', pwd, salt, itrs)`

• Authentication in Django?
  • [https://docs.djangoproject.com/en/2.0/topics/auth/passwords/](https://docs.djangoproject.com/en/2.0/topics/auth/passwords/)
Wrap Up

• Intro to security:
  • Confidentiality
  • Integrity
  • Availability
  • Authentication
    • Much discussion of password safety

• Next time:
  • Cryptography