Lecture 35: Cookie Monsters and Semi-Secure Websites

Secure Programming

“Honor System” Programming
All your users are nice and honest
Nothing terribly bad happens if your program misbehaves
Enough to (hopefully) make you dangerous!

“Real World” Programming
Some users are mean and dishonest
Bad things happen if your program misbehaves

Buffer Overflows

int main (void) {
    int x = 9;
    char s[4];
    gets(s);
    printf("s is: %s\n", s);
    printf("x is: %d\n", x);
}

Stack

C Program

Note: your results may vary (depending on machine, compiler, what else is running, time of day, etc.). This is what makes C fun!

Code Red

Can you have a Buffer Overflow vulnerability in Scheme, Charme, LazyCharme, StaticCharme, or Python?

No (unless there is a bug in the underlying implementation)! Memory is managed by the interpreter, so you don’t have to allocate it, or worry about how much space you have.
Web Application Security

- Malicious users can send bad input to your application
- Authentication: most interesting applications need user logins

Cross-Site Scripting

Python Code: Evaluate using Python interpreter, send output

#!/uva/bin/python
...

Output pages contain information provided by other users!

SQL Command

Values

Database

Cross-Site Scripting Demo

user: evans
password: $1$79756$Fq4bh/ajnBmzIX.12GPnL0

Enter Review:

<script language="javascript">
function button()
{
    while (1) alert("I Own you!");
}
</script>

<BODY onLoad="button()">

Preventing Cross-Site Scripting

- Never never never ever trust users!
- Everything you generate from user input needs to be checked and sanitized (remove the tags)

For your ps9 websites, you may assume all users are bound by the UVa Honor Code and won’t do anything evil. But, don’t forget how irresponsible it is to put something like this on the web!

Authentication

- 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

Login does direct password lookup and comparison.

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

Login Process

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

Eve

Trusted Subsystem

Password Problems

• Need to store the passwords
  – Dangerous to rely on database being secure

• Need to transmit password from user to host
  – Dangerous to rely on Internet being confidential

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>$\text{encrypt}_x (&quot;fido&quot;)$</td>
</tr>
<tr>
<td>ben</td>
<td>$\text{encrypt}_x (&quot;schemer&quot;)$</td>
</tr>
<tr>
<td>dave</td>
<td>$\text{encrypt}_x (&quot;Lx.Ly.x&quot;)$</td>
</tr>
</tbody>
</table>

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

Hashing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;dog&quot;</td>
<td>&quot;neanderthal&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;horse&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$H(\text{char } s[i]) = (s[i] - \text{‘a’}) \mod 10$

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 $H(y) = H(x)$.
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 \( y \).

A Better Hash Function?

- \( H(x) = \text{encrypt}_j(0) \)
- 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 use in HoosHungry
  - It has been broken!

Hashed Passwords

<table>
<thead>
<tr>
<th>UserID</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alyssa</td>
<td>md5(&quot;fido&quot;)</td>
</tr>
<tr>
<td>ben</td>
<td>md5(&quot;schemer&quot;)</td>
</tr>
<tr>
<td>dave</td>
<td>md5(&quot;Lx.Ly.x&quot;)</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>Password Type</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>

(Morris/Thompson 79)
### Salt of the Earth
(This is the standard UNIX password scheme.)

Salt: 12 random bits

<table>
<thead>
<tr>
<th>UserID</th>
<th>Salt</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>alyssa</td>
<td>1125 DES+25 (0, &quot;Lx.Ly.x&quot;, 1125)</td>
<td></td>
</tr>
<tr>
<td>ben</td>
<td>2437 DES+25 (0, &quot;schemer&quot;, 2437)</td>
<td></td>
</tr>
<tr>
<td>dave</td>
<td>932 DES+25 (0, &quot;Lx.Ly.x&quot;, 932)</td>
<td></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?

---

### Authenticating Users
- User proves they are a worthwhile person by having a legitimate email address
  - Not everyone who has an email address is worthwhile
  - Its 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
  
  \[ md = \text{str} \left( \text{random.randint} (0, 9999999) \right) + \text{str} \left( \text{random.randint} (0, 9999999) \right) \]
  
  \[ encmd = \text{md5crypt.encrypt} \left( \text{md}, \text{str} \left( \text{random.randint} (0, 999999) \right) \right) \]

  users.userTable.createUser (user, email, firstnames, lastname, encmd)

---

### Users and Passwords
```python
def createUser(self, user, email, firstnames, lastname, password):
    c = self.db.cursor()
    encpwd = md5crypt.encrypt (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
Using Cookies

- Cookie must be sent before any HTML is sent (util.printHeader does this)
- Be careful how you use cookies – anyone can generate any data they want in a cookie
  - Make sure they can’t be tampered with: use md5 hash with secret to authenticate
  - Don’t reuse cookies - easy to intercept them (or steal them from disks): use a counter than changes every time a cookie is used

Hungry vs. Cookies

```python
def checkCookie():
    try:
        if 'HTTP_COOKIE' in os.environ:
            cookies = os.environ['HTTP_COOKIE']
            c = Cookie.SimpleCookie(cookies)
            user = c['user'].value
            auth = c['authenticator'].value
            count = users.userTable.getCookieCount(user)
            ctester = md5crypt.encrypt(constants.ServerSecret + str(count) + user, str(count))
            if ctester == auth:
                users.userTable.setCurrentUser(user)
                return True
            else:
                users.userTable.setCurrentUser(False)
                return False
        else:
            return False
    except:
        return False
```

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 (later)
- Proving you can read an email account is not good enough to authenticate for important applications

Charge

- Feel free to use the ps8 users/cookies code for your ps9 site unchanged
- But, don’t put anything really valuable on your websites without paying more attention to security!