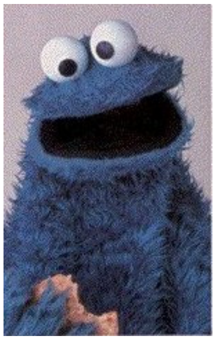


# Lecture 35: Cookie Monsters and Semi-Secure Websites

CS150: Computer Science
University of Virginia
Computer Science

David Evans
http://www.cs.virginia.edu/evans

## Secure Programming

**cs150**

**"Honor System" Programming**

All your users are nice and honest

Nothing terribly bad happens if your program misbehaves

Enough to (hopefully) make you dangerous!

**cs205**

**"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);
}
```

C Program

| | |
|---|---|
| | . . . |
| | h |
| return address | g |
| | f |
| x | e |
| s[3] | d |
| s[2] | c |
| s[1] | b |
| s[0] | a |
| | |

Stack



## 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);
}
```

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

```
> gcc -o bounds bounds.c
> bounds
abcdefghijkl
s is: abcdefghijkl
x is: 9
> bounds
abcdefghijklm
s is: abcdefghijklmn
x is: 1828716553
> bounds
abcdefghijkln
s is: abcdefghijkln
x is: 1845493769
> bounds
aaa... [a few thousand characters]
crashes shell
```

(User input)

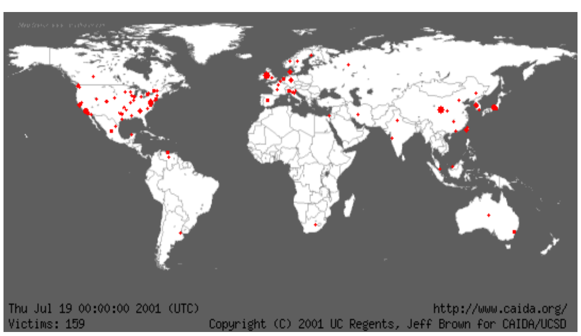
= 0x6d000009

= 0x6e000009

What does this kind of mistake look like in a popular server?



## Code Red






## Security in cs150

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

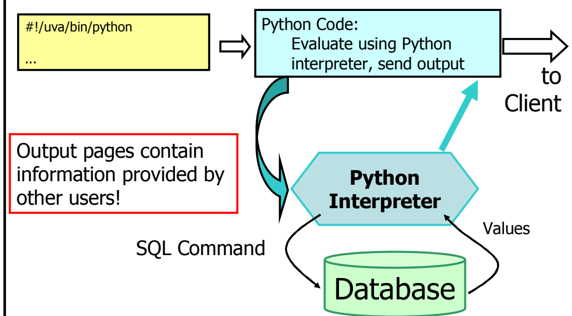
No (unless there is a bug in the underlying implementation)! Memory is managed by the interpreter, so you don't have to allocate it, or worry about how much space you have.

## Web Application Security

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



## Cross-Site Scripting






## Cross-Site Scripting Demo

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

Enter Review:

```
<script language="javascript">
function button()
{
        while (1) alert("I 0wn 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



## 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

Login does direct password lookup and comparison.

| UserID | Password |
|---|---|
| alyssa | fido |
| ben | schemer |
| dave | Lx.Ly.x |

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



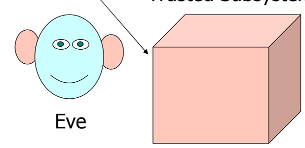
# Login Process

Terminal

```
Login: alyssa
Password: fido
```

login sends
<"alyssa", "fido">

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

Today

Later Class



# First Try: Encrypt Passwords

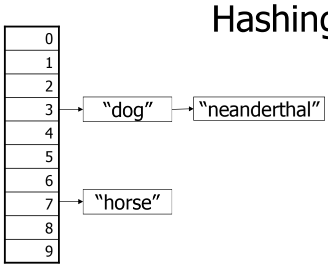
- Instead of storing password, store password encrypted with secret *K*.
- When user logs in, encrypt entered password and compare to stored encrypted password.

| UserID | Password |
|---|---|
| alyssa | $\text{encrypt}_K$ ("fido") |
| ben | $\text{encrypt}_K$ ("schemer") |
| dave | $\text{encrypt}_K$ ("Lx.Ly.x") |

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



# Hashing




$H$ (char $s$[]) = ($s$[0] – 'a') mod 10

**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.



# 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 $x * y$

Given $x$ and $y$, it is easy to calculate
$f(x, y)$ = select middle 100 digits $(x * y)$

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



## A Better Hash Function?

- $H(x) = \text{encrypt}_x(0)$
- 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

| UserID | Password |
|---|---|
| alyssa | *md5* ("fido") |
| ben | *md5* ("schemer") |
| dave | *md5* ("Lx.Ly.x") |



## 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

| | |
|---|---|
| Single ASCII character | 0.5% |
| Two characters | 2% |
| Three characters | 14% |
| Four alphabetic letters | 14% |
| Five same-case letters | 21% |
| Six lowercase letters | 18% |
| Words in dictionaries or names | 15% |
| **Other (possibly good passwords)** | **14%** |

(Morris/Thompson 79)

## Salt of the Earth

(This is the standard UNIX password scheme.)

Salt: 12 random bits

| UserID | Salt | Password |
|---|---|---|
| alyassa | 1125 | DES+$^{25}$ (0, "Lx.Ly.x", 1125) |
| ben | 2437 | DES+$^{25}$ (0, "schemer", 2437) |
| dave | 932 | DES+$^{25}$ (0, "Lx.Ly.x", 932) |

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

// We use the username as a "salt" (since they must be unique)
**encryptedpass = md5crypt.encrypt (password, user)**

| user | password |
|---|---|
| alyssa | 9928ef0d7a0e4759ffefbadb8bc84228 |
| evans | bafd72c60f450ed665a6eadc92b3647f |



## 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

```
rnd = str(random.randint (0, 9999999))
        + str(random.randint (0, 9999999))
encrnd = md5crypt.encrypt
               (rnd, str(random.randint (0, 99999)))
users.userTable.createUser (user, email, firstnames, \
                                   lastname, encrnd)
...
```

From register-process.cgi

Do you trust Pythons random number generator?



## Users and Passwords

```
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 ()

def checkPassword(self, user, password):
    c = self.db.cursor ()
    query = "SELECT password FROM users WHERE user='" + user + "'"
    c.execute (query)
    pwd = c.fetchone ()[0]
    if not pwd:
      return False
    else:
      encpwd = md5crypt.encrypt (password, user)
      return encpwd == pwd
```

From users.py (cookie processing and exception code removed)



## 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

```
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)
      ctest = md5crypt.encrypt (constants.ServerSecret + str(count) + user, \
                                 str(count))
      if ctest == 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!