Web Application Security

Raluca Ada Popa

Feb 25, 2013

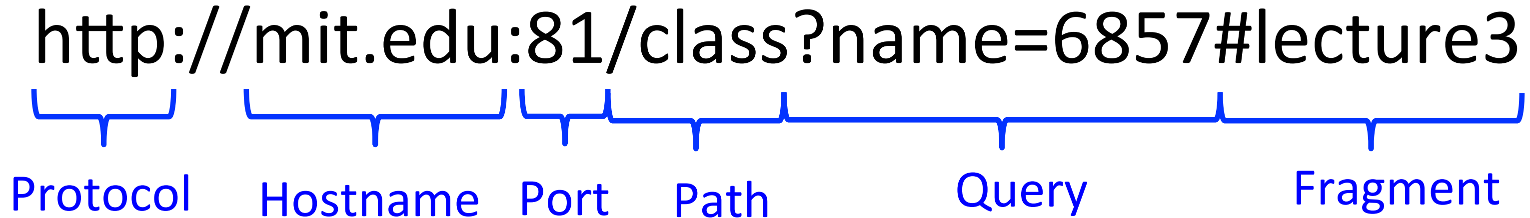
See last slide for credits

Outline

- Web basics:
  - HTTP
- Web security:
  - Authentication: passwords, cookies
  - Security attacks

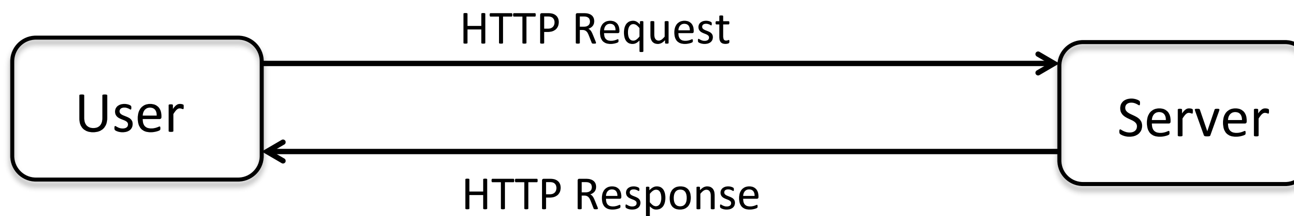
URL (Uniform Resource Locator)

- A global reference to a resource retrievable over the network



HTTP (Hypertext transfer protocol)

- The main transfer mechanism of the Web
- Used to exchange resources identified by URL between server and clients



HTTP Request

1. Method:
   - GET: get data
   - POST: put data
   - others: PUT, DELETE
2. Path
3. Headers
4. Data content

HTTP Request

Method **Path** **HTTP version**

```
GET /helloworld.html HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg,
*/*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0;
Windows 95)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Headers

Blank line

Data content – none for GET

HTTP Response

1. Status code with reason text
   - 200 OK
   - 404 not found
   - others
2. Headers
3. Data

HTTP Response

```
HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: …
Content-Length: 2543

<HTML> Hello world .. </HTML>
```

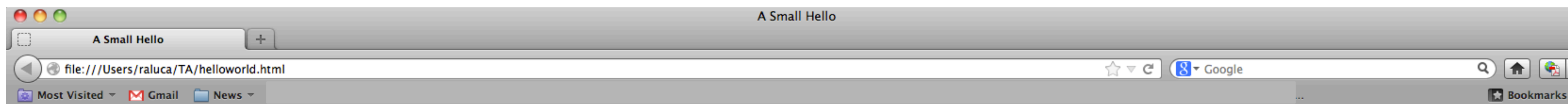
Data content

- Web page = HTML file + references

- References
  - Presentation (style): CSS
  - Multimedia: image, video, audio
  - Behavior (scripts): JavaScript
  - Behavior (plug-ins): Flash etc.

Content example

```
<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
 <HEAD>
    <TITLE>A Small Hello</TITLE>
 </HEAD>
 <BODY>
    <H1>Hi</H1>
    <P>This is very minimal "hello world" HTML document.</P>
 </BODY>
</HTML>
```



Hi

This is very minimal "hello world" HTML document.

HTTP is stateless

=

Server or client does not maintain state

- Server and client maintain state using cookies, a database, etc.

Web security

- Authentication
- Three top attacks

Goal of web security

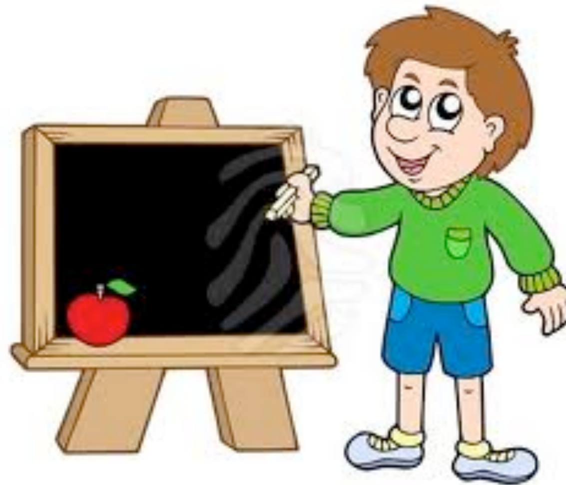
- Safely browse the web: Users should be able to visit a variety of web sites, without incurring harm:
    - No one can steal or read user's information without permission
    - No one can modify or take advantage of user's information

Authentication

Server authenticates a user U if the server checks that it is indeed talking to user U

Common method: passwords

Passwords



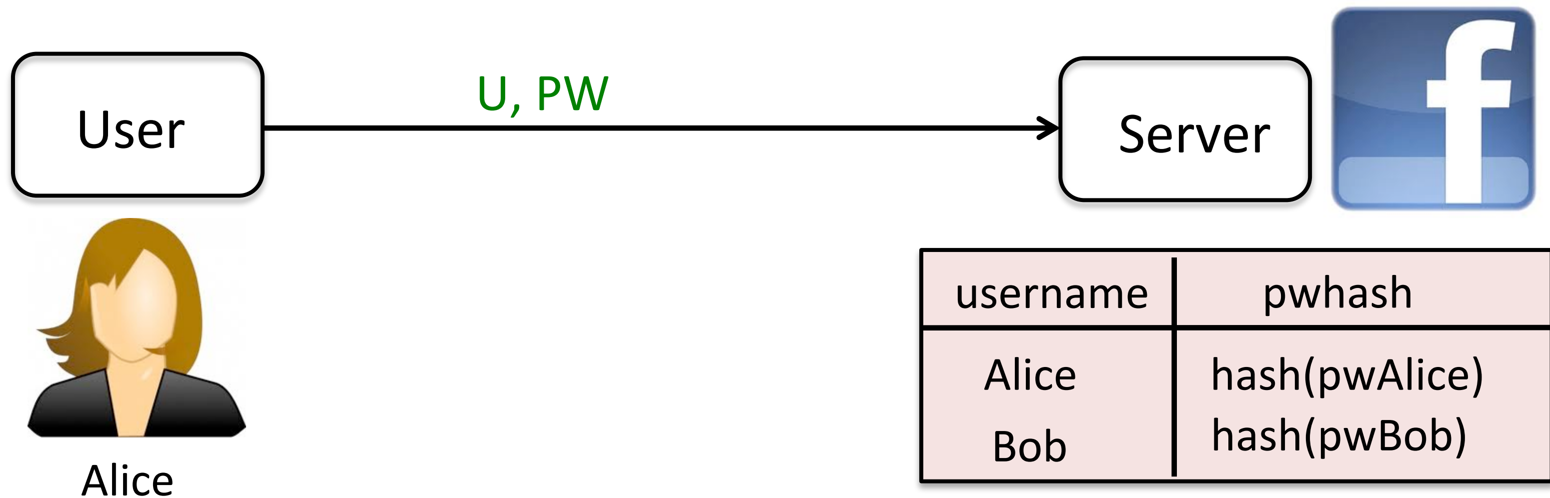
(presentation is on board, but slides posted)

Passwords

- Goal: best attacker strategy is to guess password:
  - Implemented protocol should not make it any easier to adversary

Passwords

- Big compromise if adversary steals table of passwords, so store hashes at server



Verify(U, PW):
table[U].pwhash $\overset{?}{=}$ hash(PW)

Passwords (cont'd)

- Hash should be one-way:
  - even if adversary steals table of hashes, adversary should not be able to find password

Weak passwords

- People often choose passwords from a small set:
  - The 6 most common passwords (sample of 32×10^6 pwds): 123456, 12345, Password, iloveyou, princess, abc123
  - 23% of users choose passwords in a dictionary of size 360,000,000

Dictionary attack

- Given hash(PW), adversary hashes every word from a dictionary Dict until it matches hash(PW)

- Online attack: server prevents it by using increasing delay after each incorrect password attempt

Offline dictionary attack

- Time $O(|Dict|)$ per password
- Off the shelf tools (John the ripper, Cain and Abel, etc.)
  - Scan through 360,000,000 guesses in few minutes
    - Will recover 23% of passwords

Batch Offline Dictionary Attacks

- Suppose attacker steals table T and wants to crack all passwords

username	pwhash
Alice	hash(pwAlice)
Bob	hash(pwBob)

- Builds list L containing **(w, H(w))** for all $w \in$ Dict
- Finds intersection of L and T

- Total time: **O(|Dict| + |T|)**
- Much better than a dictionary attack on each password **O(|Dict| X |T|)**

Preventing Batch Dictionary Attacks

- Use a random 64-bit salt with each hash

username	salt	pwhash
Alice	5939	hash(5939, pwAlice)
Bob	2341	hash(2341, pwBob)

- To verify (U, PW) for a user, test

  table[U].pwhash = hash(table[U].salt, PW)

- Batch attack time is now: O(**|Dict| × |T|**)

Reusing password across sites

- Resulting security is the one of weakest site

- Solution: use client side software to convert a common password pw into a unique site password pw'

  $pw' \leftarrow H(\, pw, \text{server-id} \,)$

- Required hash properties: one-wayness, non-malleability

Cookies

Cookies

= files stored by the server at the client

- maintain state

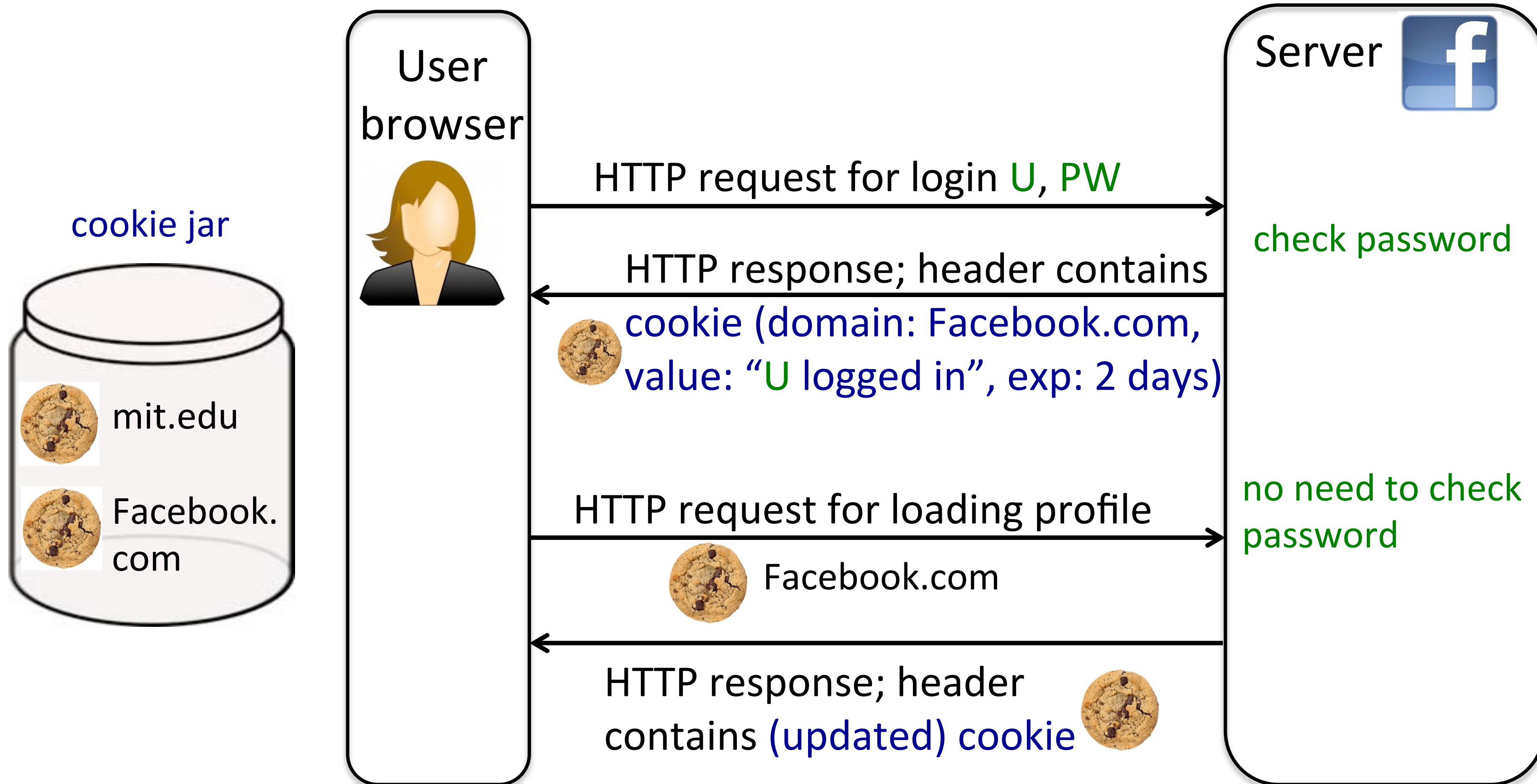
- also useful for authentication:
  - Server can remember client logged in

  Avoids sending password over the network many times

Cookie contents

- name: 6857cookie
- value: e.g., uid, number of visits
- Domain: mit.edu
- path: /courses/2013/
- expiration: in 7 days

HTTP with cookies



Browser automatically includes cookies whose domain match the suffix of URL

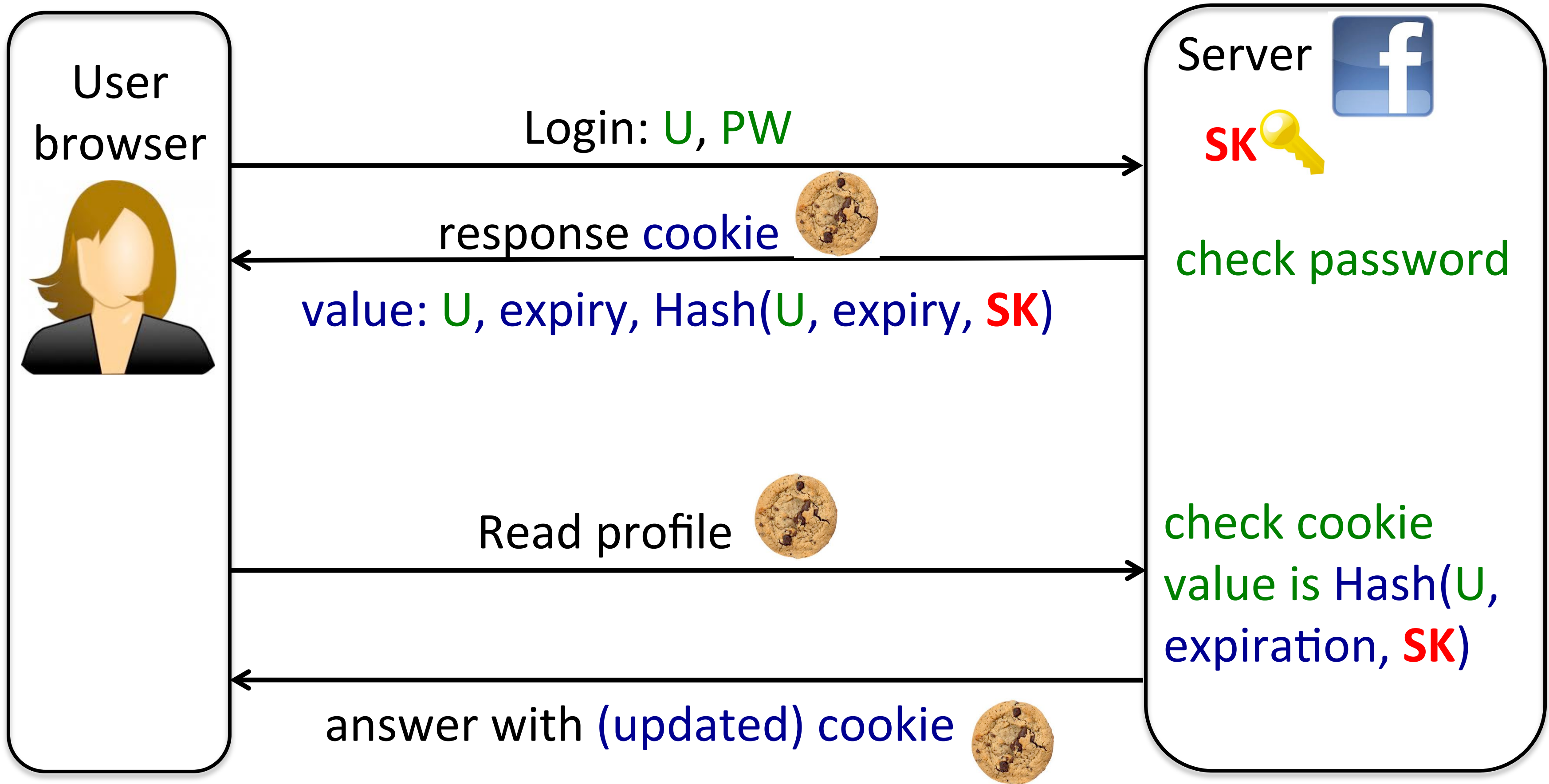
Cookies have no integrity!

- Anyone can change them, copy them, etc.

➡ Attacker can claim he is logged in to Alice's account

- Amazon attack

Fix: Unmodifiable cookies



Hash properties?

Cookie value: U, expiry, Hash(U, expiry, **SK**)

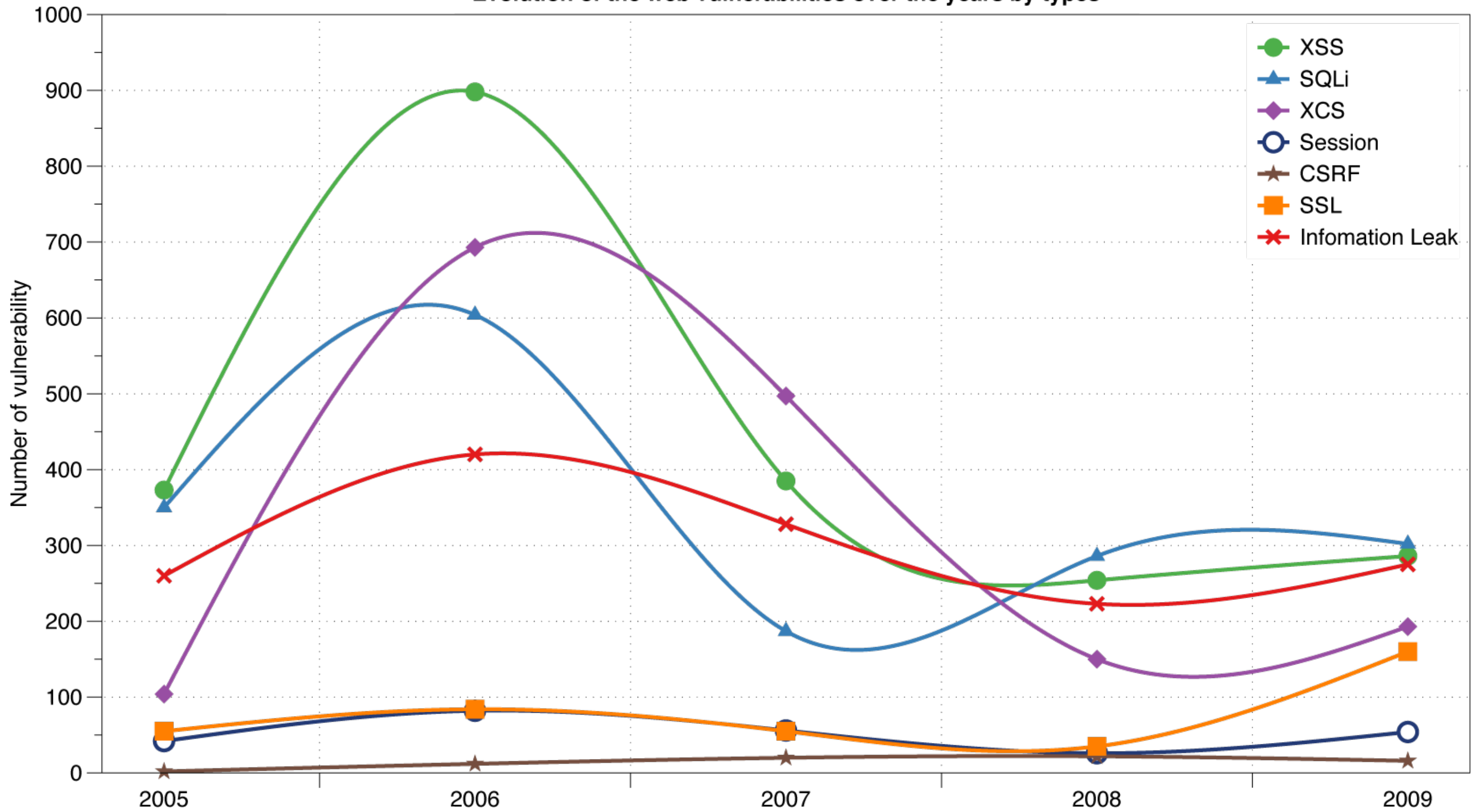
- At least one-wayness and non-malleability, but not enough. Need unforgeability.
- Would suffice if hash were a random oracle
- MACs or signatures used instead

Attacks on Web Applications

Three top web site vulnerabilites

1. SQL Injection
2. CSRF – Cross-site request forgery
3. XSS – Cross-site scripting

Reported Web Vulnerabilities "In the Wild"



Data from aggregator and validator of NVD-reported vulnerabilities

SQL Injection

- Attacker sends malicious input to server
- Bad input checking leads to malicious SQL query

Example: buggy login page

User sends uname and pw to server

Server code:

```
ok = execute(
"SELECT count(*) FROM Users WHERE user=' "
& uname
& " ' AND    pwd=' "
& pw & " '"
);

If ok login success else  fail;
```

directly from user

uname	pw
Alice	pwAlice
Bob	pwBob

Bad input

- Suppose user = " 'or 1=1 -- " (URL encoded)

- Then scripts does:

```
ok = execute("SELECT …
      WHERE user= ' ' or 1=1  -- … )
```

  – The "--" causes rest of line to be ignored.

  – Login succeeds!

- Bad news: easy login to many sites this way.

Attack affected and affects sites

- CardSystems
  - credit card payment processing company
  - SQL injection attack in June 2005
  - put out of business
  - 263,000 credit card #s stolen from database
  - credit card #s stored unencrypted
  - 43 million credit card #s exposed

Fixes

- Sanitize input: make sure SQL arguments are properly escaped

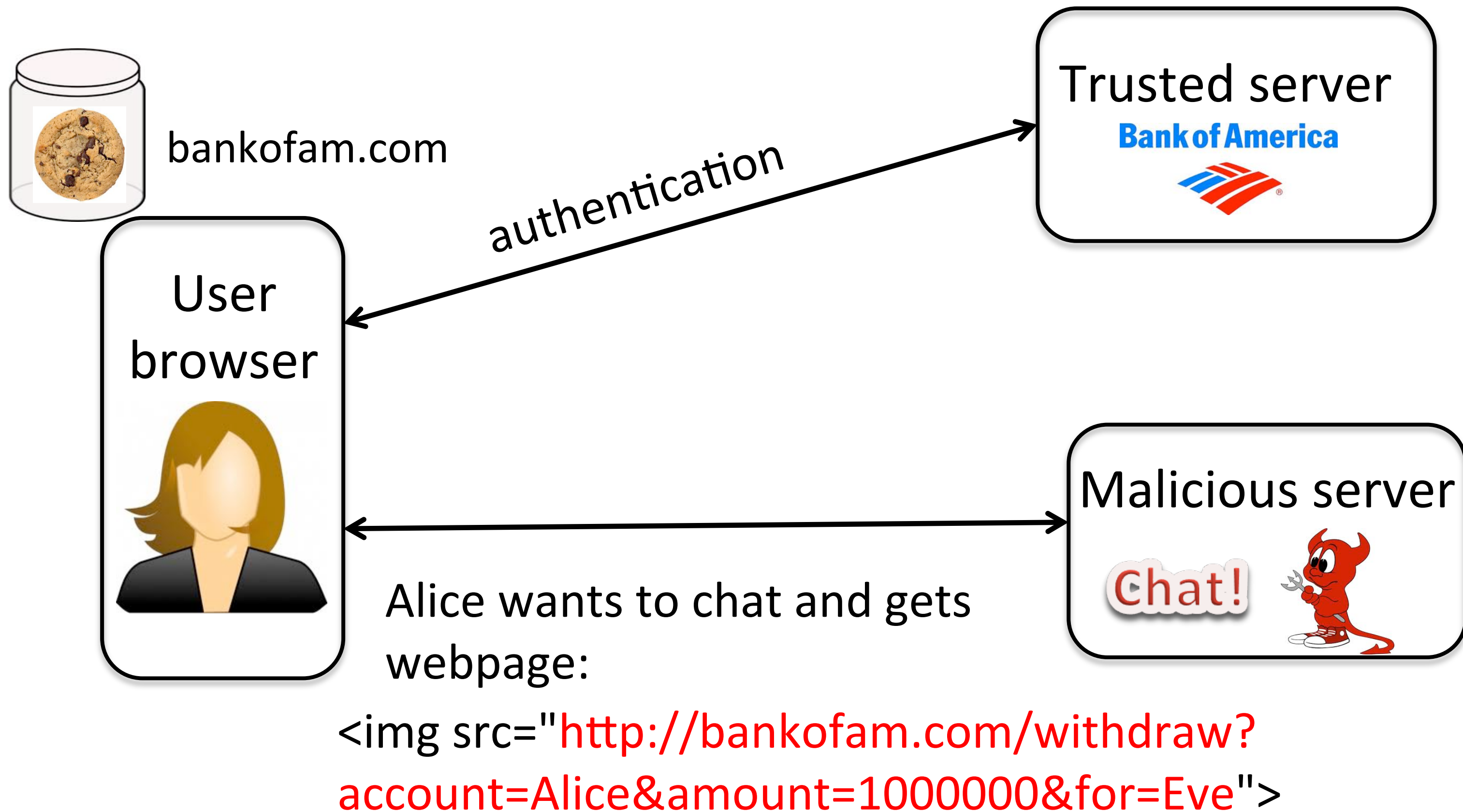
```
ok = execute("SELECT …
        WHERE user= ' \' or 1=1  --\' … )
```

  - Username does not match!

CSRF – Cross-site request forgery

- Bad web site sends a request to good web site pretending to be the browser of an innocent user, using credentials of the innocent victim

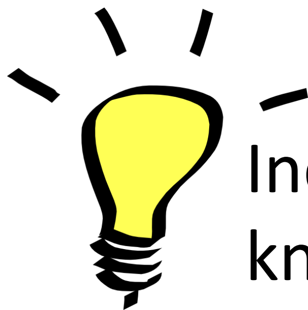
Examples



Alice’s browser wants to render image so it makes the withdrawal request automatically using Alice’s cookie!

CSRF Countermeasure

- Good server needs to ensures that user really intended action:
  - User fetched a page, filled in the form for the request, and sent the request
  - Attacker did not fetch page, sends request directly



Include random token in fetched page –not known to attacker

Random tokens

- When user fetches a page, server embeds a token in forms; server stores token for a user in a database

Webpage rendering: Recipient: Submit

Webpage code:

```
<% <form>
<input <input <input <input
</form>
<input name=“Recipient”>
<input type=“Submit”>
<input name=“99438” type=”hidden” value=“99438”>
</form>>
```

- When user sends form, token is sent to server along with user cookie

Server checks:

token from form $\stackrel{?}{=}$ token from database for user with that cookie

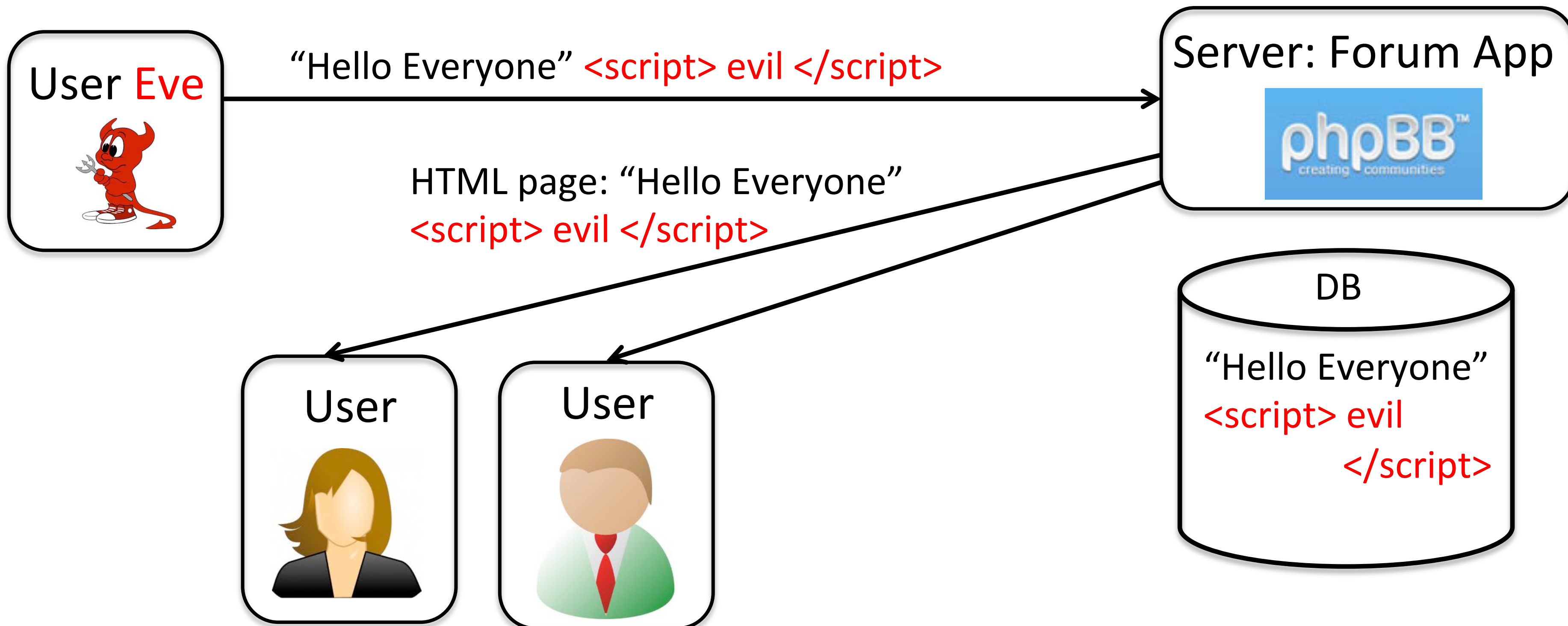
Attacker does not know token!

XSS – Cross-site scripting

– Attacked web site sends innocent victim a script that steals information from an honest web site

XSS

- Attackers sends data with script to server
- Server stores it thinking it is data and then serves it to other users



When browser renders page…

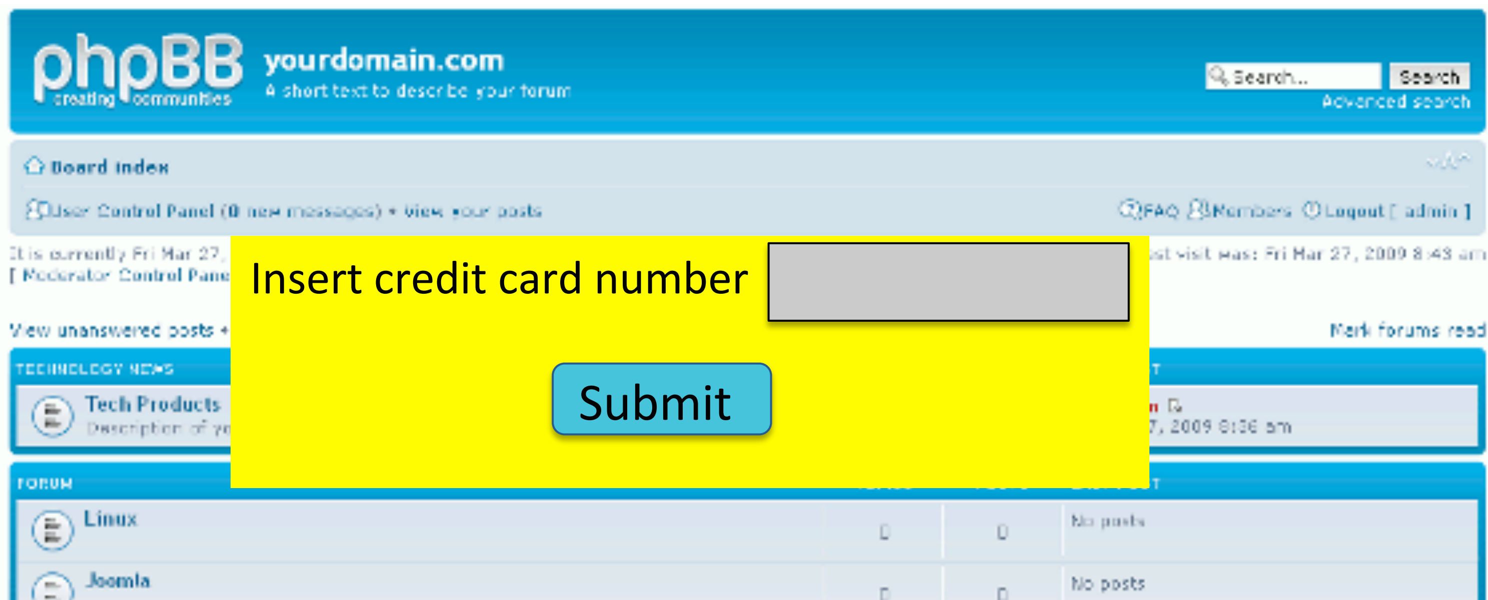
- Shows content to user

``Hello Everyone’’

- .. and executes script!

Script can …

- steal all user cookies or other credentials and send to Eve
- change the rest of the forum webpage and ask for credit card number



Fixes

Difficult to prevent, must employ a set of fixes, example:

- Server web app escapes any user-provided data before sending it to other users

<script> ⟶ <script>

Script displayed instead of run

``Hello Everyone’’
<script> evil </script>

Sum up

- Passwords and cookies used for authentication
- Three top attacks:
  - SQL injection: bad input checking allows malicious SQL query
  - CSRF: attacker makes victim user browser issue request with victim credentials
  - XSS: victim user browser runs script from attacker

Resources used for these slides

- Stanford CS155, 2012
- Victor Costan’s MIT 6.857 lecture, 2012
- Wellesley CS110, lecture M13
- MIT 6.033 lecture 22, 2012
- Book: Tangled Web