Web Security II: Cross-site and UI attacks

Chengyu Song

Slides modified from
Dawn Song, Raluca Ada Popa and Dan Boneh

HyperText Markup Language

HTML: a markup language to create structured documents that can embed

images, objects, create interactive forms, etc.

```
<html>
  <body>
    <div>foo <a href="http://google.com">Go to Google!</a></div>
    <form>
      <input type="text" /> <input type="radio" />
      <input type="checkbox" />
    </form>
  </body>
</html>
```

Web security: a historical perspective

- Similar to Internet, web is an example of "bolt-on security"
- Originally, the World Wide Web (www) was invented to allow physicists to share their research papers
  - Only textual web pages + links to other pages
  - No security model to speak of

Web security: nowadays

- The web became complex and adversarial quickly
- Web pages become very complex with embedded images, JavaScript, dynamic HTML, AJAX, CSS, frames, audio, video, sensors, VR, … from different servers
    - Today, a web site is a distributed application
- Web applications also become very diverse, news, shopping, videos, social network, banking, gaming, …
    - Attackers have various motivations

Desirable security goals

- **Integrity**: malicious websites should not be able to tamper with the integrity of my computer or my information on other web sites
- **Confidentiality**: malicious websites should not be able to learn confidential information from my computer or other web sites
- **Privacy**: malicious websites should not be able to spy on me or my activities online

How to achieve these goals?

- Reference monitor (access control)
  1. How to name/identify subject and object?
  2. What would be the access control policy?
- What about network level?
  - One layer at a time
  - TLS, DNSSEC, etc

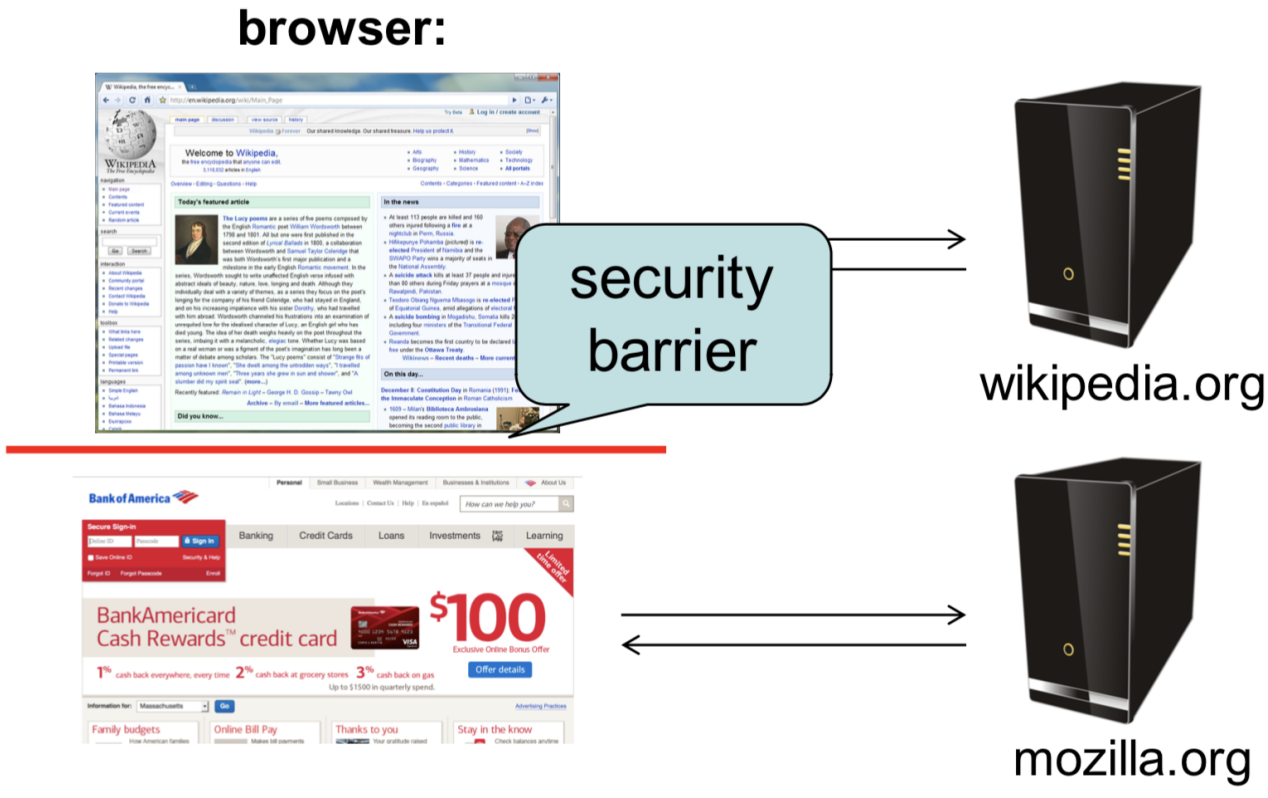
How these properties can be violated?

- Server side: injection attacks
- Client side: cross-site attacks

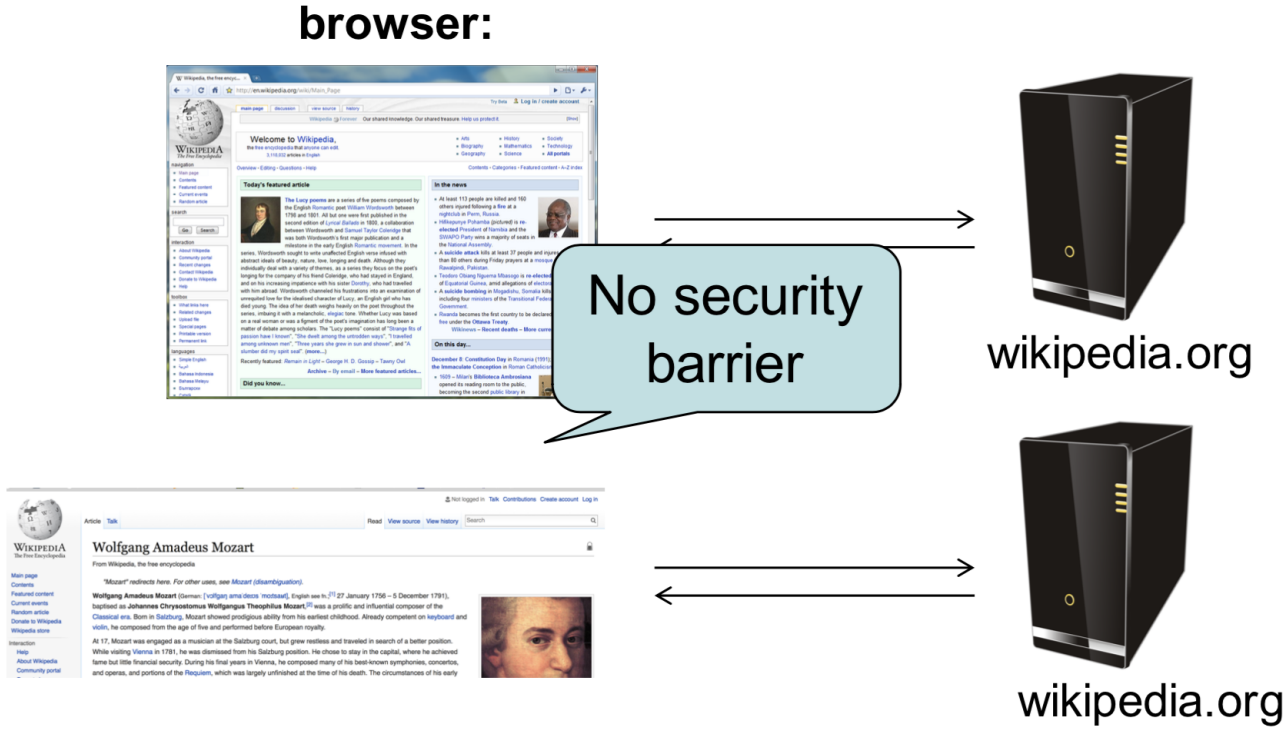
Same-origin policy

- **The most important access control policy for web applications**
  1. Each site in the browser is isolated from all others
  2. Multiple pages from the same site are not isolated

Same-origin policy: different sites

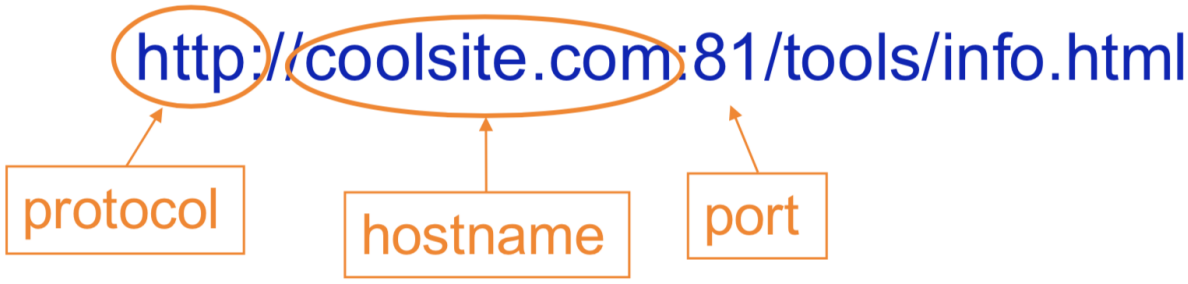


Same-origin policy: same site



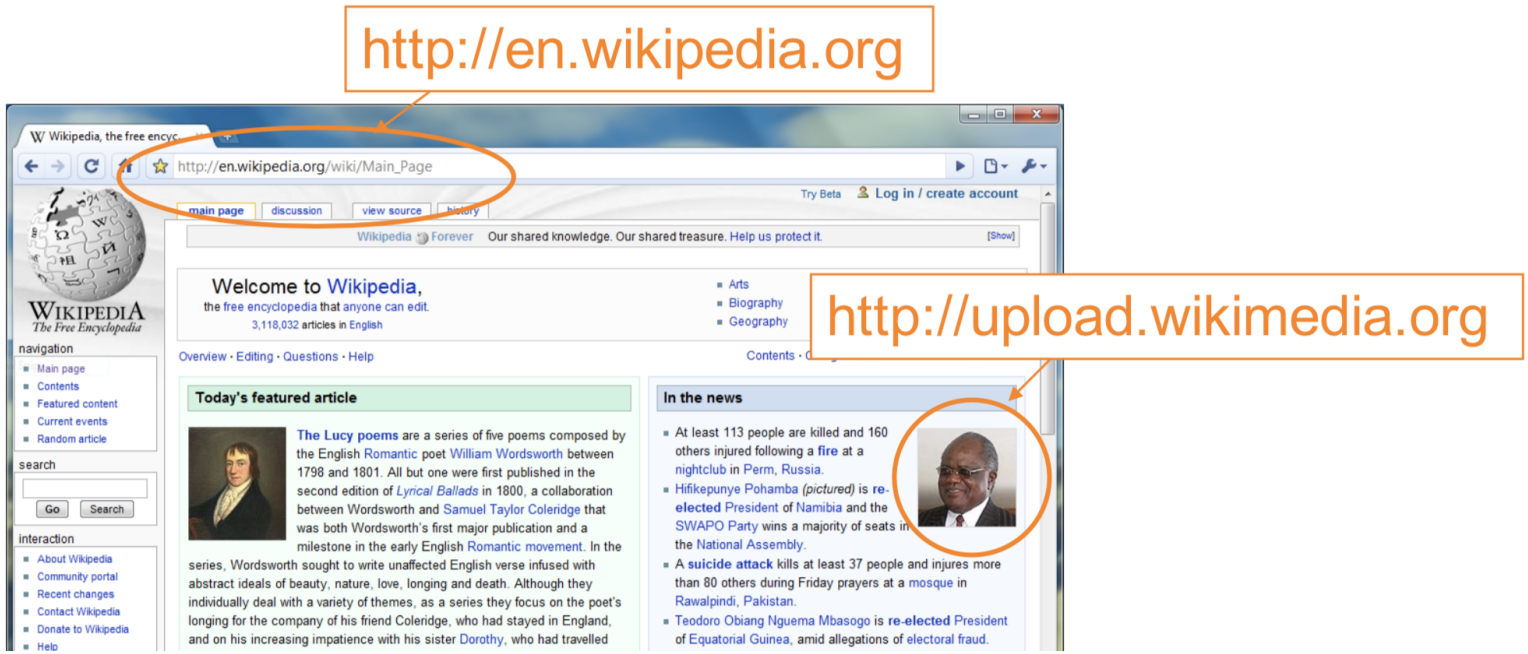
What is an Origin?

- Origin = protocol + hostname + port



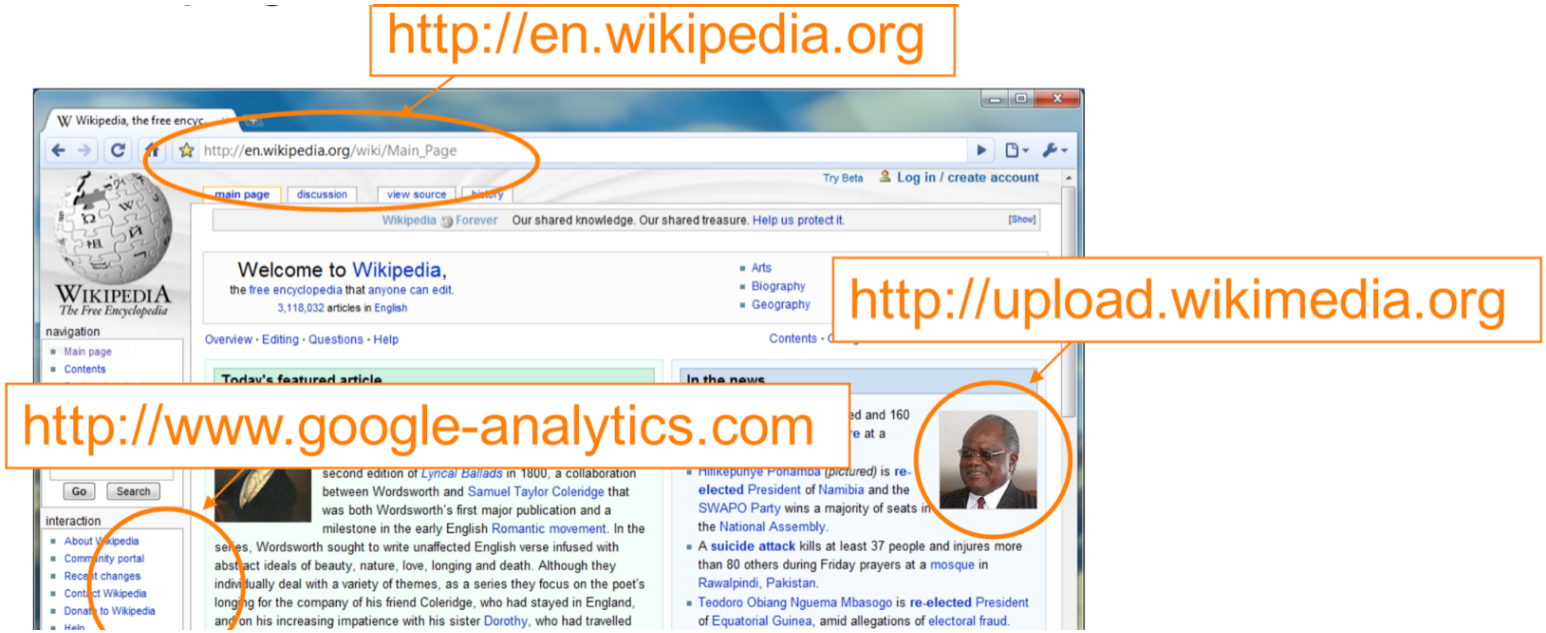
How to define the origin?

- The origin of a resource is derived from the URL it was loaded from



How to define the origin?

- Special case: Javascript runs with the origin of the page that loaded it



Exercises

Originating document	Accessed document
http://wikipedia.org/**a**/	http://wikipedia.org/**b**/
http://wikipedia.org/	http://**www.**wikipedia.org/
http://wikipedia.org/	**https**://wikipedia.org/
http://wikipedia.org**:81**/	http://wikipedia.org**:82**/
http://wikipedia.org**:81**/	http://wikipedia.org/

Exercises

Originating document	Accessed document
http://wikipedia.org/**a**/	http://wikipedia.org/**b**/
http://wikipedia.org/	http://**www.**wikipedia.org/
http://wikipedia.org/	**https**://wikipedia.org/
http://wikipedia.org**:81**/	http://wikipedia.org**:82**/
http://wikipedia.org**:81**/	http://wikipedia.org/

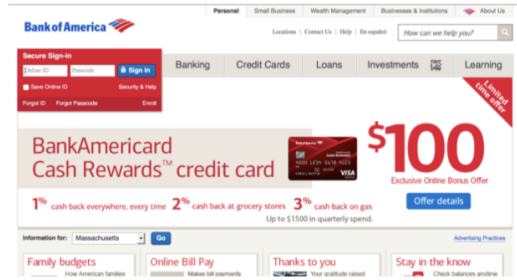
Cross-origin communication

- Similar to IPC, different origins can communicate through a narrow API:

  `postMessage`

- Receiving origin decides if to accept the message based on origin

```
postMessage
("run this
script",
script)
```



Check origin, and request!

Cross-site scripting (XSS)

- Vulnerability in web application that enables attackers to inject client-side scripts into web pages viewed by other users

Three types of XSS

- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- Type 1: Reflected
  - The attack value is 'reflected' back by the server
- Type 0: DOM Based
  - The vulnerability is in the client side code

Type 2 XSS

- Consider a form on safebank.com that allows a user to chat with a customer service associate.
    1. User asks a question via HTTP POST message: "How do I get a loan?"
    2. Server stores the question in a database.
    3. Associate requests the questions page.
    4. Server retrieves all questions from the DB
    5. Server returns HTML embedded with the question

Type 2 XSS

Assuming the query page is implemented in PHP

```
<? echo "<div class='question'>$question</div>";?>
```

Which will be rendered into

```
<div class='question'>How do I get a loan?</div>
```

Type 2 XSS

Look at the following code fragments. Which one of these could possibly be a comment that could be used to perform a XSS injection?

a. `'; system('rm -rf /');`
b. `rm -rf /`
c. `DROP TABLE QUESTIONS;`
d. `<script>doEvil()</script>`

Type 2 XSS

Look at the following code fragments. Which one of these could possibly be a comment that could be used to perform a XSS injection?

```
a. '; system('rm -rf /');
b. rm -rf /
c. DROP TABLE QUESTIONS;
d. <script>doEvil()</script>
```

```
<html><body>
...
    <div class='question'><script>doEvil()</script></div>
...
</body></html>
```

Type 2 XSS

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>
HTML Code: <div class='question'><script>doEvil()</script></div>
```

3. Victim requests the questions page

4. Server retrieves malicious question from the DB

1. Attacker asks malicious question via HTTP POST
(message: "<script>doEvil()</script>")

5. Server returns HTML embedded with malicious question

2. Server stores question in

Server

Type 1 XSS

- Consider safebank.com also has a transaction search interface at search.php
- search.php accepts a query and shows the results, with a helpful message at the top.

```
<? echo "Your query $_GET['query'] returned $num results.";?>
```

- Example: *Your query chocolate returned 81 results.*
- How can you inject `doEvil()`?

Type 1 XSS

- A request to `search.php?query=<script>doEvil()</script>` causes script injection. Note that the query is never stored on the server, hence the term 'reflected'.

```
PHP: <? echo “Your query $_GET['query'] returned $num results.";?>

HTML: Your query <script>doEvil()</script> returned 0 results
```

Type 1 XSS

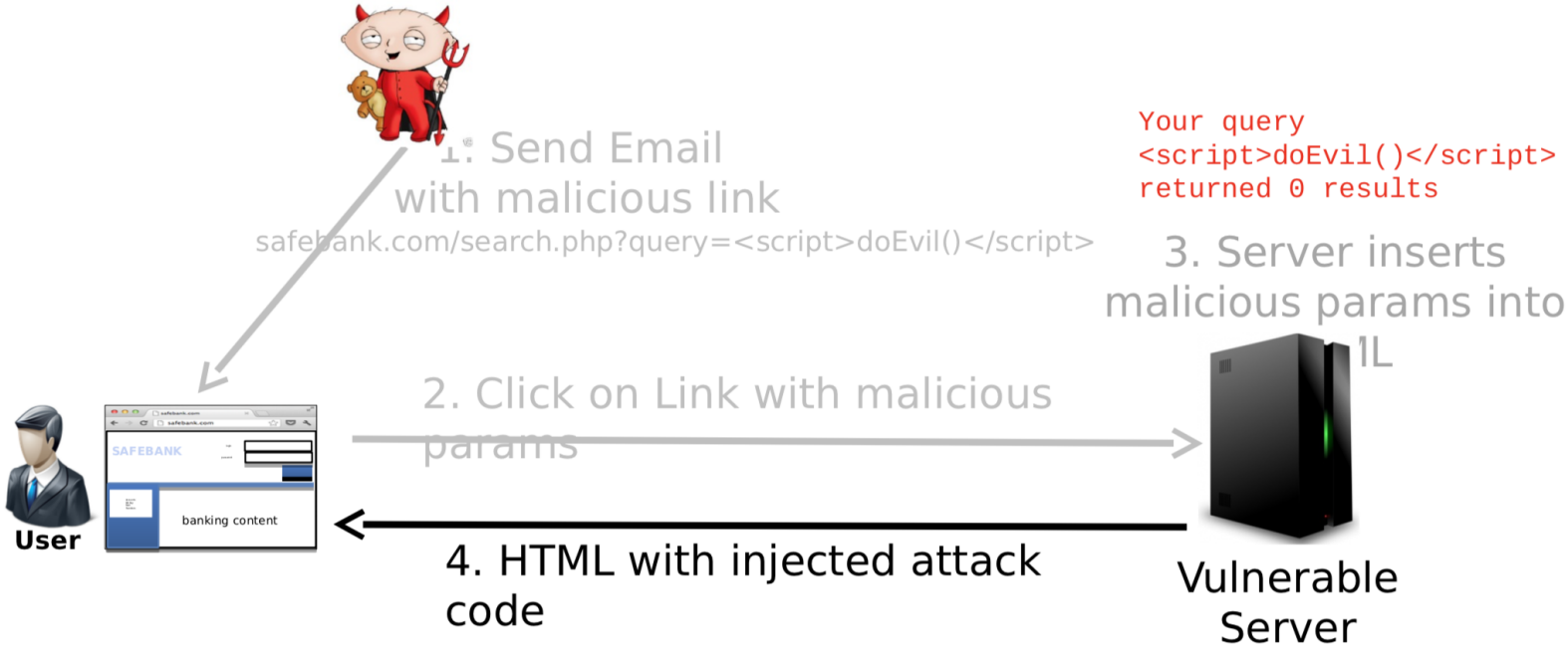
- Q: But this only injects code in the attacker's own page. The attacker needs to inject code in the user's page for the attack to be effective.

Type 1 XSS

- Q: But this only injects code in the attacker's own page. The attacker needs to inject code in the user's page for the attack to be effective.
- A: How about send to the victim an email with a malicious link?

```
safebank.com/search.php?query=<script>doEvil()</script>
```

Type 1 XSS


1. Send Email
with malicious link
safebank.com/search.php?query=<script>doEvil()</script>
Your query
<script>doEvil()</script>
returned 0 results
3. Server inserts
malicious params into
URL
2. Click on Link with malicious
params
SAFEBANK
banking content
User
4. HTML with injected attack
code
Vulnerable
Server

Type 0 XSS

- Traditional XSS vulnerabilities occur in the *server side code*, and the fix involves improving sanitization at the server side
- Web 2.0 applications include significant processing logic, at the client side, written in JavaScript
- Similar to the server, this code can also be vulnerable

Type 0 XSS

- Suppose safebank.com uses client side code to display a friendly welcome to the user. For example, the following code shows "Hello Joe" if the URL is:

`http://safebank.com/welcome.php?name=Joe`

```
Hello
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

Type 0 XSS

```
Hello
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

For the same example, which one of the following URIs will cause untrusted script execution?

a. http://attacker.com
b. http://safebank.com/welcome.php?name=doEvil()
c. http://safebank.com/welcome.php?name=<script>doEvil()</script>

Injection defenses

- Input validation
  - Whitelists untrusted inputs
- Input escaping
  - Escape untrusted input so it will not be treated as a command
- **Use less powerful API**
  - Use an API that only does what you want
  - Prefer this over all other options

Input validation

- Check whether input value follows a whitelisted pattern. For example, if accepting a phone number from the user, JavaScript code to validate the input to prevent server-side XSS:

```
function validatePhoneNumber(p){
var phoneNumberPattern = /^\(?(\d{3})\)?[- ]?(\d{3})[- ]?(\d{4})$/;
return phoneNumberPattern.test(p);
}
```

- This ensures that the phone number doesn't contain a XSS attack vector or a SQL Injection attack. This only works for inputs that are easily restricted.

Parameter tampering

- Q: Is the JavaScript check in the previous function on the client sufficient to prevent XSS attacks?

Parameter tampering

- Q: Is the JavaScript check in the previous function on the client sufficient to prevent XSS attacks?
- A: No. Attackers can handcraft the request, bypassing the JavaScript check.

Input escaping or sanitization

- Sanitize untrusted data before outputting it to HTML. Consider the HTML entities functions, which escapes 'special' characters. For example, `<` becomes `<`
- Our previous attack input

```
<script src="http://attacker.com/evil.js"></script>
```

becomes

```
&lt;script src=&quot;http://attacker.com/evil.js&quot;&gt;&lt;/script
```

Use a less powerful API

- The current HTML API is too powerful, it allows arbitrary scripts to execute at any point in HTML
- **Content Security Policy** allows you to disable all inline scripting and restrict external script loads
- Disabling inline scripts, and restricting script loads to 'self' (own domain) makes XSS a lot harder
- See CSP specification for more details

Use a less powerful API

- To protect against DOM based XSS (Type 0), use a less powerful JavaScript API
- If you only want to insert untrusted text, consider using the `innerText` API in JavaScript. This API ensures that the argument is only used as text.
- Similarly, instead of using `innerHTML` to insert untrusted HTML code, use `createElement` to create individual HTML tags and use `innerText` on each.

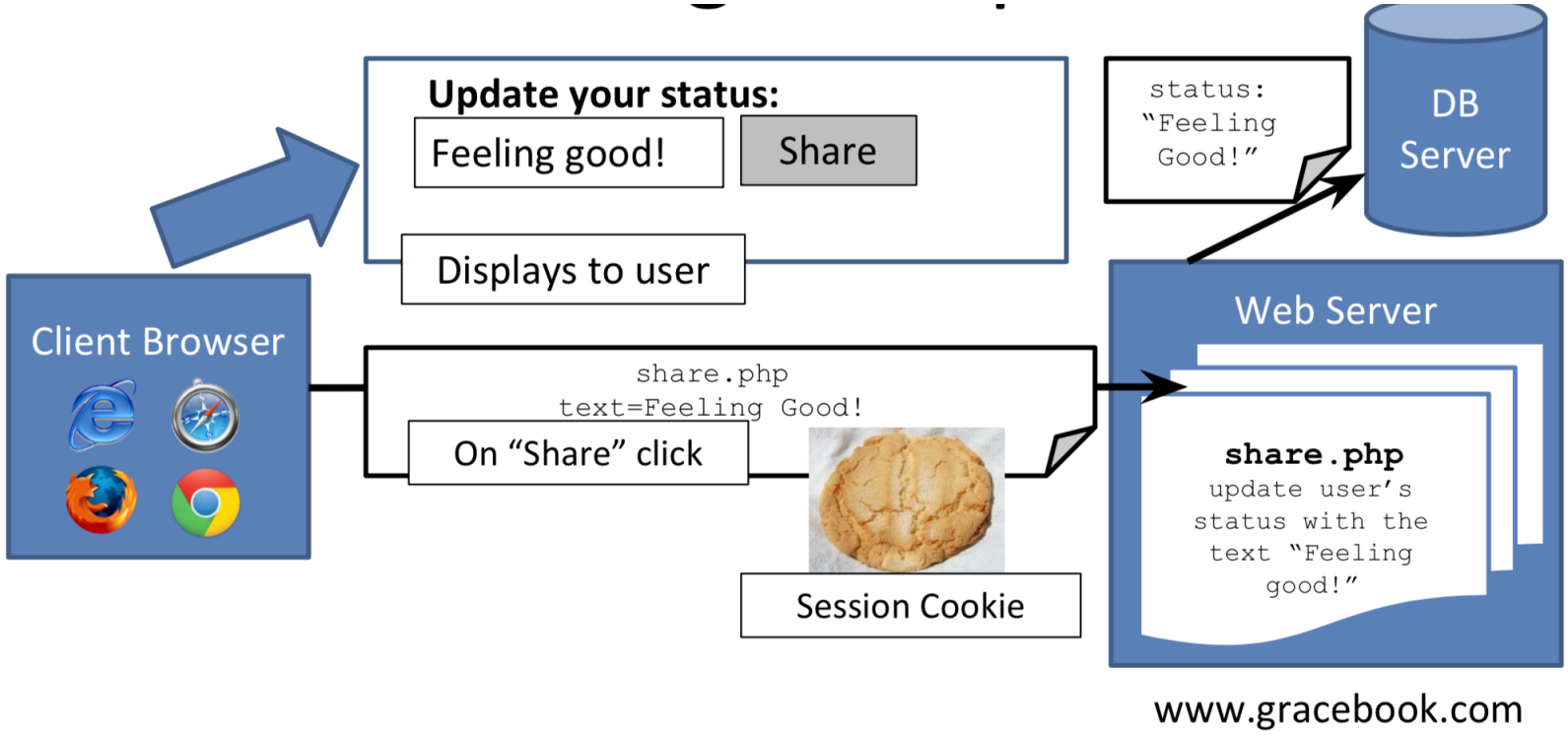
Cross-Site Request Forgery (CSRF)

- Consider a social networking site, GraceBook, that allows users to 'share' happenings from around the web.
- Users can click the "Share with GraceBook" button which publishes content to GraceBook.
- When users press the share button, a `POST` request to `http://www.gracebook.com/share.php` is made and gracebook.com makes the necessary updates on the server.

Running example

```
<html><body>
<div>
Update your status:

<form action="http://www.gracebook.com/share.php" method="post"> <inp
<input type="submit" value="Share"></input>
</form>
</div>
</body></html>
```

Running example



Network request

- The HTTP `POST` Request looks like this:

```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Referer:
   https://www.gracebook.com/form.php
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

text=Feeling good!
```

CSRF attack

- The attacker, on `attacker.com`, creates a page containing the following HTML:

```
<form action="http://www.gracebook.com/share.php" method="post"
 id="f">
<input type="hidden" name="text" value="SPAM COMMENT"></input>
<script>document.getElementById('f').submit();</script>
```

CSRF attack

- What will happen when the user visits the page?

a) The spam comment will be posted to user's share feed on gracebook.com
b) The spam comment will be posted to user's share feed if the user is currently logged in on gracebook.com
c) The spam comment will not be posted to user's share feed on gracebook.com

CSRF attack

- JavaScript code can automatically submit the form in the background to post spam to the user's GraceBook feed.
- Similarly, a `GET` based CSRF is also possible.
  - Making GET requests is actually easier: just an `img` tag suffice

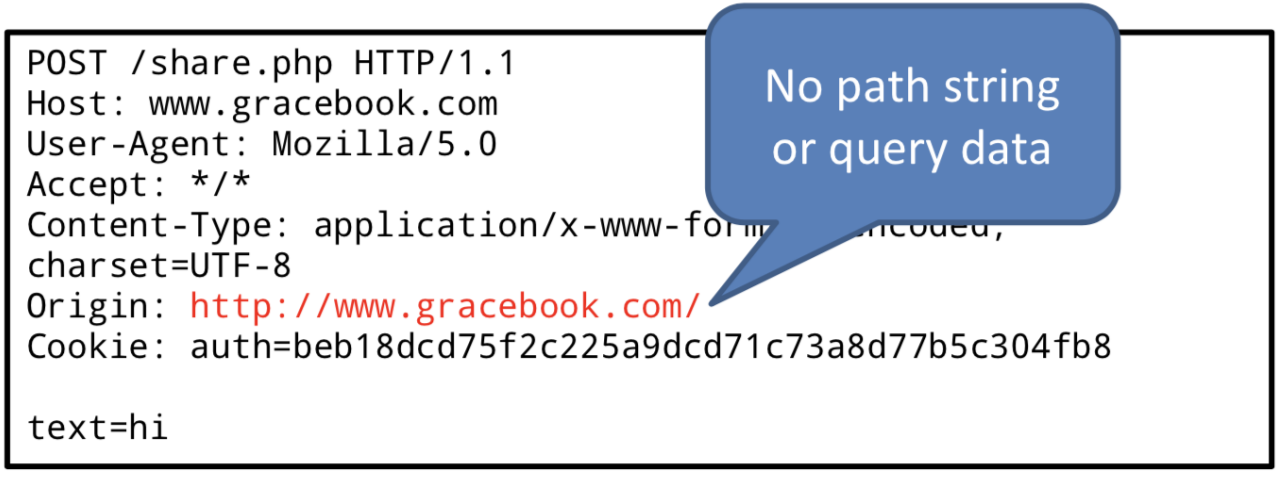
```
<img src="http://www.gracebook.com/share.php?text=SPAM%20COMMENT" /
```

CSRF defense

- Origin header
  - Introduction of a new header, similar to `Referer`.
  - Unlike `Referer`, only shows scheme, host, and port (no path data or query string)
- Nonce-based
  - Use a nonce to ensure that only `form.php` can get to `share.php`

Origin header

- Instead of sending whole referring URL, which might leak private information, only send the referring scheme, host, and port.



Nonce based protection

- Recall the expected flow of the application:
  1. The message to be shared is first shown to the user on `form.php` (the `GET` request)
  2. When user assents, a `POST` request to `share.php` makes the actual post
- The server creates a nonce, includes it in a hidden field in `form.php` and checks it in `share.php`.

Nonce based protection

The form with nonce

```
<form action="share.php" method="post">
<input type="hidden" name="csrfnonce" value="av834favcb623">
<input type="textarea" name="text" value="Feeling good!">
```

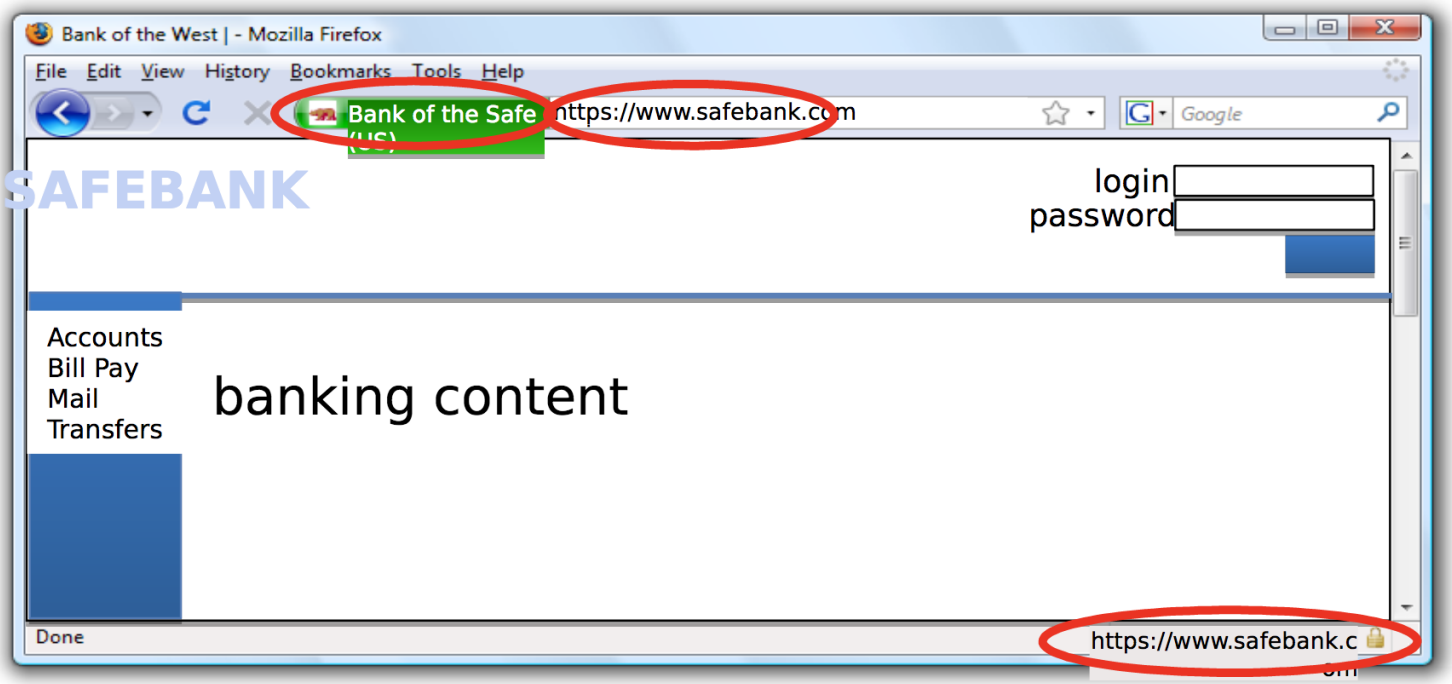
```
POST /share.php HTTP/1.1
Host: www.gracebook.com
User-Agent: Mozilla/5.0
Accept: */*
Content-Type: application/x-www-form-urlencoded;
charset=UTF-8
Origin: http://www.gracebook.com/
Cookie: auth=beb18dcd75f2c225a9dcd71c73a8d77b5c304fb8

Text=Feeling good!&csrfnonce=av834favcb623
```

Server code compares nonce

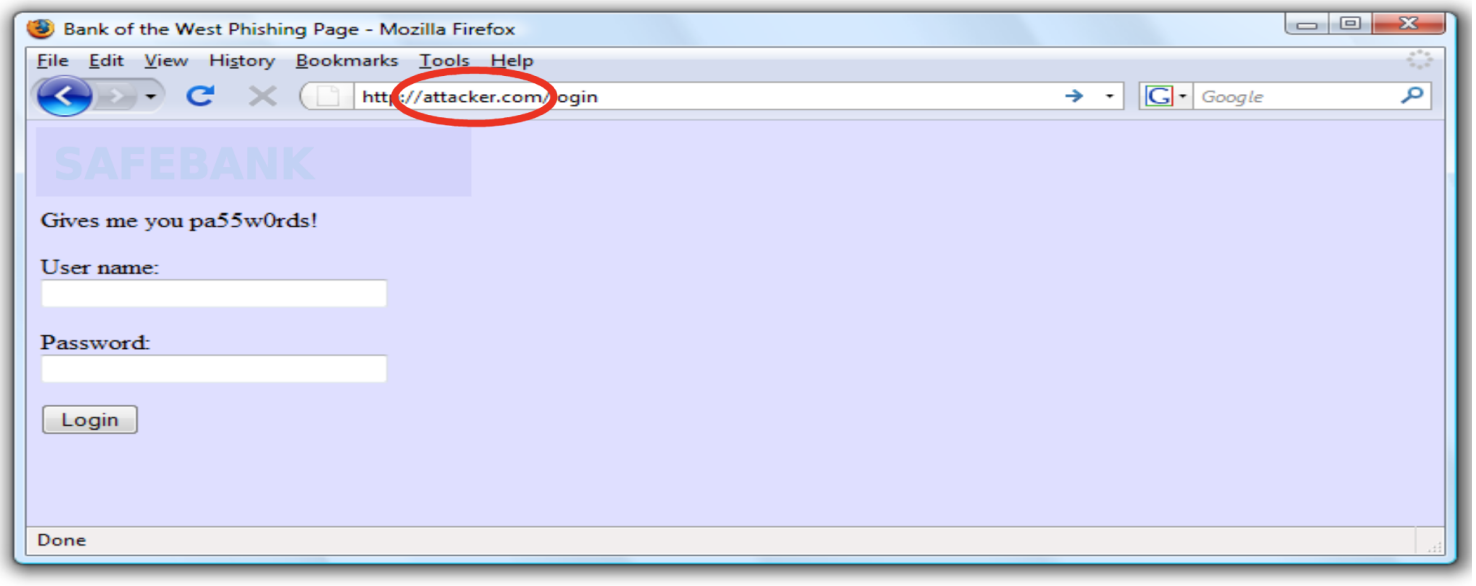
UI attacks

- Use visual tricks to lure users to perform unintended bad operations
- Address bar attack
  - Exploitation where the URL displayed in the address bar is not the one you visited
- Clickjacking attacks
  - Exploitation where a user's mouse click is used in a way that was not intended by the user

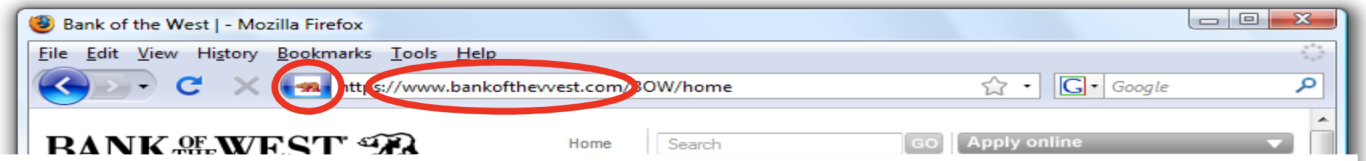
Safe to type your password?



Safe to type your password?



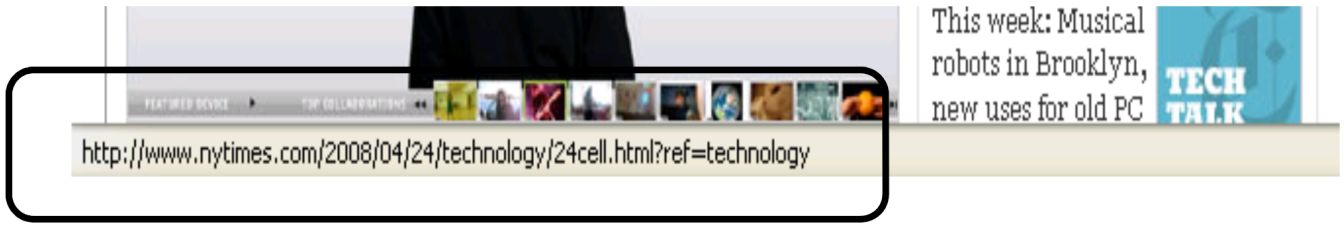
Safe to type your password?



thevvest.com/

thewest.com/

Status bar



- Trivially spoofable

```
<a href="http://www.paypal.com/"
        onclick="this.href = 'http://www.evil.com/';">
    PayPal</a>
```

Cursorjacking

- Can customize cursor!

```
CSS example:
#mycursor {
cursor: none;
width: 97px;
height: 137px;
background: url("images/custom-cursor.jpg")
 }
```

- Javascript can keep updating cursor, can display shifted cursor

Fake cursor, but more visible

Real cursor

Cursorjacking

Fake, but more visible

real

Cursorjacking