Web security
Context...

- We have seen many examples of attacks due to *insufficient input validation*
  - Buffer overflows
  - SQL injection attacks

- We continue to look at more attacks in this vein
  - Client state manipulation in web requests
    - Hidden form variables or parameters in HTTP requests
    - Cookie manipulation

Context

- We will then look at *cross-domain* attacks that involve *three* parties – the attacker and an honest client + server
  - Cross-site scripting (XSS)
  - Cross-site request forgery (CSRF)

- (XSS attacks can also be viewed as being caused by improper input validation)
Common source of flaws

- HTTP is *stateless*
  - State – whether *per-session* or *across* sessions – is often stored at the client side (i.e., cookies)
  - State is echoed back by client in future requests
  - This state is subject to manipulation!

Example web application I

- order.html
  - order form allowing user to select number of pizzas and enter credit card info
- submit_order
  - script that processes the user’s order, and generates an HTML form to be sent back to the client for verification
  - Price encoded as hidden form field

```html
<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">
</FORM>
</BODY></HTML>
```
Example web application II

- When the user clicks, the browser issues an HTTP request like
  \[ \text{GET /submit\_order?price=5.50&pay=yes HTTP/1.0} \]

- The user’s submitted request is processed by a back-end credit-card payment gateway

```plaintext
if (pay = yes) {
    bill\_creditcard(price);
    deliver\_pizza();
}
else
    display\_transaction\_cancelled\_page();
```

In pictures…

```
Web Browser (Client)  Credit Card Payment Gateway
Order 1 Pizza  Submit Order $5.50
Web Server
Confirm $5.50
```

Price Stored in Hidden Form Variable
```
submit\_order?price=5.50
```
Carrying out the attack

- Attacker orders pizza, gets order confirmation HTML page

![HTML code]

- Attacker can view the page source

```html
<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=".01">
<INPUT TYPE="submit" NAME="pay" VALUE="yes">
<INPUT TYPE="submit" NAME="pay" VALUE="no">
</BODY></HTML>
```

- And modify it!

- When form submitted, it generates the request
  
  GET /submit_order?price=0.01&pay=yes HTTP/1.0
Notes

- Even though the price variable is “hidden”, the client can find it in the HTML source in the clear
- Nothing prevents modification of the pre-populated values!
- Using POST instead of GET has the same vulnerability
- Streamline the attack using HTTP-generation tools
  - curl, Wget

Solution 1

- Store state on the server
  - Server creates a session-id for each session, and stores a table mapping session-ids to state
  - Session-id sent to client, who re-sends it in its requests

```html
<HTML>
<HEAD><TITLE>Pay</TITLE></HEAD><BODY>
The total cost is $5.50. Confirm order?
<FORM ACTION="submit_order" METHOD="GET">
  <INPUT TYPE="hidden" NAME="sid" VALUE="78272901149">
  <INPUT TYPE="submit" NAME="pay" VALUE="yes">
  <INPUT TYPE="submit" NAME="pay" VALUE="no">
</FORM>
</BODY></HTML>
```
Solution 1

- HTTP request now looks like
  GET /submit_order?sid=78272901149 &pay=yes HTTP/1.0

- Back-end processing must change:

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

- Database lookup on each request – possible DoS

Notes

- Session ids must be hard to guess!
  - Randomly chosen
  - Sufficiently long

- Time out session ids

- Delete session ids once session ends
Solution 2

- *Authenticate* client-side state
- Server verifies state sent by the client
- What is the right cryptographic tool here?

Solution 2 in detail

- Server stores random, secret key $k$
- `confirm_order` generates HTML like

```html
<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="quantity" VALUE="1">
<INPUT TYPE="hidden" NAME="price" VALUE="12">
<INPUT TYPE="hidden" NAME="TAG" VALUE="371910171983">
<INPUT TYPE="submit" NAME="pay" VALUE="yes">
<INPUT TYPE="submit" NAME="pay" VALUE="no">
</FORM></BODY></HTML>
```

What if this were missing?

where $\text{TAG} = \text{MAC}_k(\text{quantity} \# \text{price})$
(A side note)

- Note that this gives the attacker a lot of control over what strings will be authenticated by the server...

- Note that there are lots of forgeries that would be damaging for the server
  - Anything where the price is changed

- Good thing our definition of security for MACs was so strong!

Cross-domain security issues
Cross-domain security issues

- Security vulnerabilities that arise due to interactions between two different domains
  - Malicious script (pointing to different domain) inserted into webpage served by legitimate domain
  - User accessing page from legitimate domain and page from malicious domain at the same time

- For the purposes of this lecture, freely assume the attacker can get a user to access any URL of the attacker’s choice
  - Phishing, embedded links/ads/scripts/iframes, ...

Same-origin policy

- Scripts embedded in a page can
  - Read/modify the contents of that page
  - Read cookies associated with that page
  - Receive/respond to events (mouse clicks)

- *Same-origin policy*: scripts can only access properties associated with documents from the same *origin* as the *document* containing the script
  - *Origin* defined by protocol+hostname+port (not document path)
  - Http and https are different protocols
Cross-domain interactions

- Links from malicious page to legitimate page
  - Nothing can prevent this!
  - Can be a link (that the user has to click) or an iframe (that automatically loads the legitimate page, without the user noticing)
  - In latter case, same-origin policy prevents script on malicious page from reading data on legitimate page
  - But `<script src=http://legitimate.com/foo></script>` in malicious page would cause legitimate script to run in context of malicious page!
    - More later

Cross-domain interactions

- Links from malicious page to legitimate page
  - Malicious page can also initiate a POST request to legitimate page, with arbitrary parameters
  - We have already seen some of the problems that can arise here
  - Due to the way web authentication is usually handled (i.e., using a `cached` credential), any http requests will look as if they come from the legitimate user
Cross-site scripting (XSS)

- Can occur whenever an attacker can influence a script executed at a legitimate host, e.g.:
  - Dynamically generated pages (search, errors, other)
  - E.g., http://good.com/error.php?msg=an+error+occured
  - What happens if the attacker sends http://good.com/error.php?msg=<script>...</script>

Exploits using (reflected) XSS

Exploits user's trust in server
Key points…

- *Same-origin policy* is respected
  - The attacker’s script was running in the context of good.com(!), so it was able to access the cookie

- Phishing likely to succeed
  - Users only notice that the link is to http://good.com

- Using https does nothing to prevent this attack…

Stored XSS vulnerabilities

- Occurs when data submitted by a user is stored and later displayed to other users
  - Comment on blog post
  - Wiki
  - Web-based email
  - Facebook, MySpace
    - Samy worm
**Exploits using (stored) XSS**

- User logs in
- Malicious code run
- Post malicious code
- Credential sent to attacker
- Exploits user's trust in server

**Notes...**

- No need for phishing any more!
- Guaranteed that user is logged in when they run the malicious script
  - (In previous case, user may not be logged in when they click the attacker-generated URL)
Payloads for XSS attacks

- Hijack session credentials
- Site defacement
  - E.g., http://good.com/error.php?msg=We+are+going+out+of+business
- Injecting trojan functionality
  - To obtain, e.g., credit card info
- Perform actions on behalf of authenticated users
  - In an automated fashion!
  - Without leaving trace of IP address!
- More…

Cross-site request forgery (CSRF)

1. Alice's browser loads page from bad.com
2. Script runs causing evilform to be submitted with a password-change request by loading www.good.com/update_pwd with attacker-specified field

```
<form method="POST" name="evilform" target="hiddenframe" action="https://www.good.com/update_pwd">
  <input type="hidden" id="password" value="badpwd">
</form>
<iframe name="hiddenframe" style="display: none"></iframe>
<script>document.evilform.submit();</script>
```

3. Browser sends authentication cookies to good server. Honest user's password is changed to badpwd!
Notes

- Due to same-origin policy, bad.com does not have access to any data associated with good.com

- When bad.com page loaded, it executes script which sends a POST request to good.com with attacker-specified parameters
  - Browser sends all cookies for good.com along with this request!

- Malicious page cannot read user's data, but can write to user's account

Notes

- CSRF for GET requests is even easier (simply use an <img> tag with a crafted URL)

NYT:
<html>
  ...
  <img src='http://nytimes.com/email?from=414&to=kelcher@gmail.com&body=you%20are%20awesome'>
  ...
</html>

Youtube:
<img src='http://youtube.com/watch_ajax?action_add_favorite_playlist=1&video_id=[VIDEO ID]&playlist_id=&add_to_favorite=1&show=1&button=AddvideoasFavorite'/>
Notes

- Can be viewed as failure of principle of complete mediation
  - User should be required to re-authenticate before changing their password

- Also (potentially) principle of least privilege
  - User should log out of a website if not actively using it

Potential CSRF vulnerabilities

- Anywhere a client can change server-side state
  - Facebook profiles
  - Financial sites
  - Calendars, etc.
**Notes**

- XSS attacks exploit the trust a client browser has in data sent from the legitimate website
  - But attacker controls what the website sends to the client browser
  - (attackers inject client-side scripts into web pages)

- CSRF attacks exploit the trust the legitimate website has in data sent from the client browser
  - But attacker controls what the client browser sends to the website

- XSS vulnerabilities are "more general"
  - Simply inject a script that, when viewed, submits a form on behalf of the user with parameters chosen by the attacker…