Alin Tomescu, CSE408
Thursday, May 5th, Lecture #25

Cross site-scripting (XSS)

Cross site scripting attacks (XSS)

Last time we talked about the same origin policy, which governed which websites could access the cookies that were stored on your computer.

The goal for an attacker is to run code on your browser such that it thinks the code came from *bank.com.* This way the code can access *bank.com*'s cookies and wreak havoc.

Suppose *bank.com* has a customer service website where certain customers post questions and employees answer questions. If you have a problem, you go there and search through the list of problems or post your own problem.

- User x can post an input, and when you go there you will see that input/content.

The assumption is that when user x sends a question, they just type in a reasonable question like "Why did you just credit my account with a million dollars?" What happens if x asks the following question:

```
"How do I do this thing? <script>evil JS code here</script>"
```

The bank doesn't validate the input and when another user looks at the question, the code in between the script tags will execute on that user's browser. That code will have come from *bank.com*. This is the essence of XSS.

What might he do in the script?

- He might read the cookie, and send it back to himself by reposting the cookie on the *bank.com* website.
  - He can go there later and see it.
  - He could post a link to his website containing the cookie, tricking users to click on it and resulting in him getting the cookie.
  - Even better, he could make it an <img src="link to website image?cookie=value"> which will automatically send a request to the attacker's server with the cookie embedded in it, whenever someone loads that page.

This is "stored XSS."

There are also "reflective XSS" attacks. Most often occur with search boxes. When you type something into a search, you get a page "results for <whatever you typed in>." If the search query term is just part of the URL, and the webserver is not careful when it echoes the search term on the results page then an attacker can send a link to a search results website with a bunch of JS embedded in it.

This is really common in Web 2.0 apps which accept a lot of input in a lot of ways.

Blue Print

Does not solve the problem of identifying the places where user input is outputted.

Browser reads webpage, constructs the DOM -- the parse tree of the webpage. Nodes in the parse tree have different types: div, text, image. Ideally the user input node will be of type text and will not have any children. That's essentially what Blue Print ensures.

Alin Tomescu, CSE408
Thursday, May 5th, Lecture #25
Attacker provides input x to the websever, which encodes input x as $s = encode(x)$. When the webserver generates the page for the victim it generates.

```
<script>
decode(s)
dom.insert(text(s))
</script>
```

So the malicious input x is put inside a text node inside the DOM tree, and this will be safe, since it ensures that the input will merely get rendered (since it's a text node) and no JavaScript code will execute.

This is very similar to prepared statements for SQL injections.

Forced browsing bugs

Mistakes where programmers forget to perform access control inside their web applications. In some cases, if the bad guy can just get the URL of what he's trying to access then he's in.

Imagine that there's a website and there's a special page to go to administrate the blogging system like blog.com/admin.php where the programmer forgot to perform authentication.

```
require "auth.php"
```

Some programmers can forget to provide this.

Fix: source code analysis, dynamic analysis (web crawler). Turns out, there isn't a good solution to solve this problem.

Mashups

Nowadays, code from two different sources, like Facebook and Farmville, are executing in the same page. For instance, Farmville executes JavaScript within your Facebook page.

Example:

- Browser loads a page *mashup.com* and *mashup.com* is loading data from two other websites: *a.com* and *b.com*
- *housingmaps.com* used *craigslist.com* and *maps.google.com* to display a map of all the housing listings on craigslist

Think of a website that wants to display your friends on a map. It needs access to your address book, so a security policy needs to be set in place, such that even though you give that website your password, it can only look at your address book.