CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 14th, 2011

Cross-site scripting attacks

Cookies refresher

A **cookie** is a pair of values: `(domain, is_ssl_only, {name, value} pairs)`

JavaScript code runs on your browser and it can access and set your cookies. You wouldn't want Slashdot to modify the bank website with JavaScript or get its cookies. That's why the **same origin policy** was created and it has been around for a decade.

Same origin policy:

- An origin is a domain name like *amazon.com*
  - What about *www.amazon.com*? Is that the same as *amazon.com*? Browser's have different rules for making educated guesses if these two are the same. Usually, JavaScript code running on *cs.sunysb.edu* can mess around with **.cs.sunysb.edu* stuff.
- There are two types of resources in JavaScript
  - HTML
  - Cookies
  - Network
- The rule is that JavaScript code from domain d can access resources from domain d only.
  - For instance, from your bank website you can only open a network connection to your own bank

Ways to get around the same origin policy?

- cross site request forgeries attacks (CSRF)
- cross site scripting attacks (XSS)

Cross site scripting (XSS) attacks

Nowadays, a website will accept input from users and some of this input will be displayed to the user.

So this creates the possibility for an attacker to send some message to the server and then the server will send that message to other clients.

Imagine a discussion board where users post messages and all the other users see the conversation. Normally, you're supposed to type in text but a bad guy can post an evil "movie review" like the following:

```
<script lang="javascript">
      alert("lol");
</script>
```

The other client's browser will think the JavaScript code came from the legitimate domain, since it actually did. The client's browser downloaded the page and the code came with it, since it was injected there by the evil attacker.

This is called **a stored cross-site scripting attack**, because the evil JavaScript is stored on the server and later executed on the client.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 14th, 2011
Other than "lolling" in the client's face, an attacker can do more vile stuff: his script can look-up the other client's cookie and post it on the website and then later come back to steal the cookie

There are also **reflected cross-site scripting attacks**: An attacker can send you an email that contains a link that has some JavaScript. For instance, if the Google search page were vulnerable, the attacker might send you:

http://google.com/?q=<script>...</script>

If Google were dumb, then the search results page will contain with the injected JavaScript and the browser will interpret the script as if it came from Google itself. Note that in this case the script is not stored on the server.

Defenses

There are some defenses you can quickly come up with:

1. Kill all the `<script>` tag
    a. Problems: Attacker can be inventive: `<<script>script>`, also JavaScript can be embedded into onClick or onLoad handlers without `<script>` tags: `<img src="" onLoad="alert();">`
2. Wrap user input into `<pre>` or `<noscript>` tags.
    a. Problems: Attacker can put stuff like `</pre><script></script><pre>`
3. Escaping all the < and > signs in the input:
    a. Problems: You cannot have stuff like bold and italic tags.
4. Another strategy (Wikipedia) is to use a different markup language that is translated into HTML.
5. Blueprint

Blueprint

Blueprint is kind of like **prepared statements**.

One thing you can do is check the DOM tree and ensure there are no script tags, but when it goes to the browser the browser might interpret it differently and a script tag might arise.

With Blueprint, the server encodes the HTML parse tree into a string (just letters and numbers), and sends that to the browser. The browser is going to interpret the encoded string, reconstruct the parse tree (identically as it was built on the server) and display the document.

How? The server inserts the Blueprint translation JavaScript code into the response. This way the browser can interpret the encoded string so there's no need for the client to install anything on his side.

No script tags

Wrapping the user input in <pre> or <noscript> tags proves to be useless since the attacker can provide evil input like:

```
</noscript><script></script><noscript>
```

One way to fix this is to add a random nonce to the noscript tags as follows:

```
<noscript auth="rand">
</noscript auth="rand">
```

Two challenges with XSS attacks

There are two challenges that must be solved with XSS attacks:

1. Properly **recognizing bad input** and **figuring out what to do with invalid input** so it doesn't get interpreted on the client
2. Once you have a good defense mechanism, **making sure you always to use your defense mechanism.** For instance, forgetting to sanitize your data is not good.
    a. Design your software so that all input comes through a single function/point in your program where you can do all your sanitization.
    b. Build a smart compiler that tracks dataflow in your code and if it can ever find a path from your input to the output that does not go through sanitization then it warns you.

Content-sniffing attacks

Content-sniffing attacks are an example of XSS attacks that does not use text as a medium to inject the attack. This can be done with images, PDF files, etc.

Think of a website like Flickr or Wikipedia that allows you to upload an image file which will be served to other users. How can there be an XSS attack with image files since image files don't have script tags in them and they are just rendered on the script?

- You could embed <script> tags in the image's metadata that might get displayed on the website.
- You could embed <script> tags in the image file's name.
- You could make the client browser believe the image file is actually an HTML file and have it display it on the screen

How can you trick the browser?

Browser and servers do content sniffing to try and figure out what files are being passed to them.

- The "file" command on UNIX gives you info about a file's content.
- The web server's response has a Content-Type: text/html field.
- The web-server is supposed to provide an accurate MIME type.
- There are not so useful MIME types like unknown/unknown or */*.

Sometimes browsers do the content identification on their side too, since the server identification might be buggy. So the browser can take something from the server that was unknown/unknown and decide it's text/html.

A bad browser can misinterpret the content of the following post script "chameleon" file:

```
%! Adobe PostScript …
% Comment: <script>…</script>
```

The server's content-sniffing algorithm thought it was a PS file, but the browser's algorithm thought it was HTML so it displayed it and executed the JS.

Defenses:

- Don't upgrade content-types like that (from postscript to text/html)

Most files have a signature in their first bytes:

- GIFs begin with GIF8[79]a
- JPEGs begins with 4 numbers ABCD

- HTML files can begin in a lot of ways. Internet Explorer made the mistake to just look in the beginning of the file for HTML tags and then assume it's an HTML file.

The HTML content-sniffing matcher should be prefix-disjoint from other signatures, such that it would only accept whitespace before HTML tags. Since none of the other formats have whitespace as their prefix, the attack would not succeed anymore.