CS 161 Computer Security
Spring 2010 Paxson/Wagner
Discussion 3

February 09, 2010

1. **Cross Site Scripting (XSS)** To launch a reflected XSS attack, you need to get the victim to click on a well-crafted malicious link.

   (a) What are some ways an attacker can get that to happen?

   (b) What are some examples of the damage the attacker could inflict on the victim?

   (c) Why does the attacker even need to bother with XSS? Why not just host a site that has in the background a script that steals all the cookies from any other sites the user has open in their browser? Any time a user goes to the attacker's site, all their session information from other sites they are visiting will be stolen.

   Answer:

   (a) Phishing emails, chat forums.

   (b) Deface web page, steal session cookies.

   (c) Same origin policy prevents one page from accessing the methods and properties of a page from a different domain. XSS is a way around that as the script does not preserve the same origin policy.

2. **Cross Site Request Forgery (CSRF)** 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/withdraw?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 some defenses against this attack?

   Answer:

   (a) The browser sends the "img" request with the patsy-bank cookie. If Victim Vern previously logged in (and didn't log out), patsy-bank might assume it is being authorized to withdraw money by Vern.

   (b) To defend against CSRF one can inspect referrer headers or add validation via a nonce/challenge token (when sending the form include a random, unguessable value that the user must return when the form is submitted.)

3. SQL Injection

(a) Explain the bug in this PHP code. How would you exploit it?

```
$query = "SELECT name FROM Users WHERE uid = $_GET['uid']"
// now execute the query
```

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

Answer:

(a) The bug is that the `uid` GET parameter is inserted directly into the query without any sanitization. To delete the Users table, pass in the following as the `uid`:

```
0; DROP TABLE Users;
```

(b) Whitelist! A `uid` only needs digits.

4. Sanitization You are building a web application for your company Widgets, Inc. You have heard a lot about these new fangled SQL injection attacks, so you decide to work in security from the start (good use of the "Principles of Security," eh?). For the website, you are working on the shopping cart feature that allows you (and other users, if you share your cart with them) to see what you are shopping for and the quantities that you have placed in your cart. You write the following client JavaScript code for the shopping cart:

```
// Updates the shopping cart of a user. The user has already logged in
// and been authenticated.
function updateShoppingCart(user, widgetName, quantity) {
    var sanitizedName = sanitizeSQLInput(widgetName);
    var sanitizedQuantity = sanitizeSQLInput(quantity);

    var query = "UPDATE cart SET quantity = '" + sanitizedQuantity
                    + "' WHERE name = '" + sanitizedName
                    + "' AND user = '" + user + "'";

    queryServer(query);
}

function sanitizeSQLInput(input) {
    // Removes all quotes, dashes, and semicolons from the input.
}

function queryServer(query) {
    // Fancy function that sends the SQL query to the server to be
    // executed directly.
}
```

What problem(s) are there with this code?

Answer: There are a few major problems with this code.

- The sanitization is incomplete. Although it bans quote characters, dashes, and semicolons from the input, there are other things that can go wrong. It might be better to have a whitelist of acceptable input, but even that may not be sufficient. Perhaps a different solution, such as prepared statements, are in order.
- `widgetName` and `quantity` are sanitized, but `user` is not.
- This only deals with SQL Injection. You have forgotten about other problems, such as XSS. This may not immediately seem like a problem, after all, it's just a shopping cart. However, imagine that shopping carts are shared between customers on this website and that, internally, the quantities are stored as strings. What would happen if you set `quantity` to the value:

  ```
  <script>/* insert bad things here */</script>
  ```

  This would result in a stored XSS when others look at your cart.
- Most importantly, this is client side code. This is a validation error in that the server should never trust anything that comes from the client because the client can always be malicious. Imagine that the user is a bad guy who rewrites the JavaScript so the assignment of `sanitizedName` is, instead:

  ```
  var sanitizedName = widgetName
  ```

  In this case, all the value of sanitization has been lost. Worse yet, a malicious client can directly call `queryServer` to send any query to the server to be executed directly, without sanitization of any kind.

5. Authentication

The typical way of keeping track of a user's session ID is to store it within a cookie. In an effort to support old browsers without cookies, some websites instead include the session ID within the URL. For example, at `foobar.com`, users may click links with URL's like `http://foobar.com/?sessionid=1234567`.

(a) What attack do you see on this scheme?

(b) Suppose that problem is fixed, and using our clever, new scheme, `foobar.com` establishes new sessions with session ids based on a hash of the tuple `(username, time of connection)`. Is this secure?

Answer:

(a) The main attack is known as "session fixation." Say Adam the Attacker establishes a session with `foobar.com`, and the session id is set to 1234567. Adam sends a link `http://foobar.com/?sessionid=1234567` to Victor the Victim. If Victor logs in through this link, he will have a session established with that id. Now Adam knows Victor's session id and can log in as Victor.

Of course, if the session id is ignored by `foobar.com` when Victor clicks on the link, and a new session id is established, this problem disappears. Even better, just use cookies since all modern browsers support them!

(b) No. This is, in fact, a common problem with session ids. Although an attacker may not be able to guess the precise time, he very well may be able to guess within a few seconds. Certainly it cannot be assumed that the username is unknown. Thus, given enough chances (perhaps a few thousand), an attacker can probably guess the correct session id.