Paxson
Spring 2011

CS 161
Computer Security

Discussion 5

February 23, 2011

Question 1 ***Cross Site Request Forgery (CSRF)*** **(7 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/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 possible defenses against this attack?

Solution:

(a) The `img` tag causes the browser to make a request to `http://patsy-bank.com/withdraw?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 it is being authorized to withdraw money by Vern.

(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 only correct way to fix this today is to use a nonce to bind the requests to the form. For example, if a request to `http://patsy-bank.com/withdraw` 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. Upon submitting the form, the random token is sent to `http://patsy-bank.com/withdraw` and Patsy-Bank can then compare the token received with the one remembered and allow the transaction to go through only if the comparison succeeded.

The modern and more flexible way to protect against CSRF is via the `Origin` header. The `Origin` header a request includes a list of sites that were involved in the creation of the request. So in the example above, the `Origin` header would include the chat forum in the `Origin` header and Patsy-Bank is going to drop this request since pages on the chat forum are untrusted. This approach

is more flexible because unlike the nonce solution above, you can allow multiple sites to cause the transaction. For example, Patsy-Bank might trust `http://www.trustedcreditcardcompany.com` to directly withdraw money from a user's account. This is a use case that the nonce based solution doesn't support cleanly. Currently, many modern browsers support the `Origin` header but there is still a sizeable chunk of users with browsers that don't support it.

Question 2 ***SQL Injection*** **(7 min)**

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

```
$query = "SELECT name FROM users WHERE uid = $_GET['uid']";
// Then execute the query.
```

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

Solution:

(a) The bug is that the `uid` GET 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) 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 `$_GET['uid']` to an integer.

The underlying issue, though, is that data can be interpreted as a command. The solution to this general issue is separate the parsing of the query from the execution (when the data is supplied). *Pepared 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' => $_GET['uid']));
```

The first line defines the SQL expression with a placeholder `:user` that is substituted with user input in the second line. Note that the substituted input is *not* parsed as SQL anymore as this already happend 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 ***Session Fixation*** **(6 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) What attack do you see on this scheme?

(b) Suppose that problem is fixed, and using our clever, new scheme, `foobar.edu` 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` and receives a session ID of 42. Then the attacker manages to make the victim's browser visit `http://foobar.edu/browse.html?sessionid=42` (maybe through a `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 sessionid 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) sessionid. For example, if the victim types in his username and password at `http://foobar.edu/login.html?sessioid=42`, then the sessionid 42 would be bound to his identity. In such a scenario, the attacker could impersonate as the victim on the site. This is uncommon nowadays, as most login pages reset the sessionid to a new random value instead of reusing an old one.

(b) The proposed fix is not secure since it solves the wrong problem as discussed above. The correct fix is to not set sessionID via URL parameters. Use cookies instead.