Paxson
Spring 2013

CS 161 Computer Security

Discussion 6

February 27, 2013

Question 1 ***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.

It is also possible to check the `Referer` header sent along with any requests. This header contains the URL of the previous, or referring, web page. Patsy-Bank can check whether the URL is `http://patsy-bank.com` and not proceed otherwise. A problem with this method is that not all browsers send the `Referer` header, and even when they do, not all requests include it.

Another problem is that when Patsy-Bank has a so-called "open redirect" `http://patsy-bank.com/redirect?to=`*url*, the referrer for the redirected request will be `http://patsy-bank.com/redirect?to=....` An attacker can abuse this functionality by causing a victim's browser to fetch a URL like `http://patsy-bank.com/redirect?to=http://patsy-bank.com/transfer`..., and from patsy-bank.com's perspective, it will see a subequent request `http://patsy-bank.com/transfer`... that indeed has a `Referer` from patsy-bank.com.

The modern and more flexible way to protect against CSRF is via the `Origin` header. This works by browsers including an `Origin` header in the requests they send to web servers. The header lists the 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. Patsy-Bank will then drop the request, since it did not originate from a site trusted by the bank (an instance of *default deny*). This approach is more flexible because unlike the token solution above, you can allow multiple sites to cause the transaction. For example, Patsy-Bank might trust `http://www.trustedcreditcardcompany.com` to directly transfer money from a user's account. This is a use-case that the token-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*** **(10 min)**

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

```
$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) 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) 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 ***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.