Solution of Exercise Sheet 5

1 SQL Injection

Consider a website *foo.com* that stores authentication cookies for its registered users in their browsers (after a successful login) using the following PHP code[1]:

```
1  $randomness = mt_rand();
2  $token = hash('sha256', $randomness.$password);
3  $arr = array($username, $token);
4  $loginCookie = base64_encode(serialize($arr));
5
6  setcookie("login", $loginCookie);
```

where `$username` and `$password` are the correct username and password of the user who has logged in. The token `$token` is also stored in the website's database. Upon an HTTP request, the website's server checks whether the client issuing the request is logged in as a registered user as follows.

```
1  /* (1) get cookie data */
2  $loginCookie = $_COOKIE["login"];
3  $arr = unserialize(base64_decode($loginCookie));
4  list($username, $token) = $arr;
5
6  /* (2) authenticate user */
7  if (!$username or !$token) { /* not logged in */ }
8  else {
9    $sql = "SELECT * FROM users WHERE " .
10          "username = '$username' AND token = '$token';";
11   $result = pg_query($sql);
12   if ( pg_num_rows($result) > 0 ) {
13     echo "Successfully logged in: ".pg_fetch_result($result, '
         username')."!";
14     /* successful login */
15   }
16   else { /* authentication failed */ }
17 }
```

(4 points)

(a) Specify values for `$username` and `$token` such that the code snippet starting at comment (2) would output the message "*Successfully logged in: administrator!*", even though you have no idea what the stored token of the user `administrator` on the website may be. You may assume that user names on the website are unique and that there indeed exists a registered user `administrator` in the database.

[1] For more information on Base64, please refer to `https://en.wikipedia.org/wiki/Base64`.

Solution:
`$username = "administrator'; --"` and `$token = "dontcare"`.
This leads to the query:

```
SELECT * FROM users WHERE
username = 'administrator'; --' AND token = 'dontcare'
```

This query will exactly return the unique row whose value in the column `username` is `administrator`. The comment `--` causes the rest of the query to be ignored, including the token check. Note that therefore the value for `$token` does not matter *as long as it is non-empty*. Otherwise, the condition `!$token` would be true, and you would be considered as not logged in.

(6 points) (b) Now, considering the entire above code snippet (starting at comment (1)), how would you craft a cookie that would allow you to authenticate as the user `administrator`? Describe the values you have to pick for the `name`, `value` and `domain` fields of the cookie (for the `value` field, explain how to compute the desired value). Note that you can edit your own cookies using appropriate plugins, e.g. *Cookie Inspector* for Chrome, or using browsers with direct support for cookie editing, e.g. Firefox.

Solution:
We need to create a cookie with the following properties:

- The `domain` is the domain of the website (i.e. foo.com), exactly as when the website sets a honest cookie.
- The `name` is `login`.
- The `value` can be computed by:
  `base64_encode(serialize(array("administrator'; --", "dontcare"))));`

(10 points) (c) In the next step, we want to fix this vulnerability using prepared statements.

1. Look up the PHP functions used here to send a query to the (PostgreSQL) database (*pg_query*).
2. Look up the existing PHP functions to use prepared statements instead of a raw query. As we are still working with PostgreSQL, they also start with `pg_`.
3. This is the part that actually gives you the points. Rewrite the given code to use prepared statements instead of a direct query. You do not have to change any function operating on the result, this is just about protecting the query.

Solution:
In order retrofit the program to use prepared statements, we first have to

prepare an SQL query to fix its structure and make it impossible to change it with user input.
PHP offers the pg_prepare function that takes two arguments as an input: The name to assign for this statement and actual SQL query, where the data input is marked with a $ sign. In our case, the statement looks like this:

```
pg_prepare("query", "SELECT * FROM users WHERE " .
   "username = $1 AND token = $2 ;");
```

After preparing the statement, we can execute it with our data:

```
$result = pg_execute("query", array($username, $token));
```

As `pg_execute` returns the same result as the original `pg_query`, there is no need to adapt the remaining code, so the result looks like this:

```
1  /* (1) get cookie data */
2  $loginCookie = $_COOKIE["login"];
3  $arr = unserialize(base64_decode($loginCookie));
4  list($username, $token) = $arr;
5
6  /* (2) authenticate user */
7  if (!$username or !$token) { /* not logged in */ }
8  else {
9    pg_prepare("query", "SELECT * FROM users WHERE " .
10   "username = $1 AND token = $2 ;");
11   $result = pg_execute("query", array($username, $token)
       );
12   if ( pg_num_rows($result) > 0 ) {
13     echo "Successfully logged in: ".pg_fetch_result(
         $result, 'username')."!";
14     /* successful login */
15   }
16   else { /* authentication failed */ }
17 }
```

2 Reflected vs stored Cross-Site Scripting

In the lecture, we have learned about two different flavors of Cross-Site Scripting, namely reflected and stored XSS.

(4 points) (a) Both need a different kind of preparation before the actual attack can happen. Assume you already have found the vulnerabilities and created the XSS code. How do you 'deliver' it to your victim? Give a brief explanation for reflected and stored each. Who (user, website) is targeted at which point in time and why?

Solution:

For reflected XSS we require the user to click on a carefully crafted link in order to trigger the attack, so we need to somehow convince her to do so. This is mostly accomplished by sending out phishing email where the attacker pretends to be another entity (bank, co-worker, ...) and tries to convince the user to click on the provided link. Often, the prepared link is concealed, so the user does not notice it is malicious.

To prepare a stored XSS attack, the attacker directly targets the vulnerable website to store the attacker code in the database. Later, when user visit the website, the attacker code that was stored before will appear as a regular part of the delivered website and execute in the victim's browser.

(6 points) (b) For this task, consider the perspective on an attacker that found exploits for both kinds of vulnerabilities in a messaging board web application. Assume you only want to exploit exactly one of those. As an attacker, which one do you most likely choose if you want to attack ...

- ... as many victims as possible without caring about the actual individuals.
- ... a well-defined set of targets that you have carefully chosen.

For each of those two scenarios above, state whether you would rather use reflected or stored XSS and give an explanation why this is the better choice.

Solution:

Many victims In case the attack tries to maximize the amount of victims, e.g. when it collects credentials or other personal information, using stored XSS on a public messaging board is the more promising approach since we do not require users to read their emails, fall for the phishing and click a prepared link. It suffices if they visit the webpage that was attacked before.

Targeted attacks In case we want to target a specific set of victims, reflected XSS is preferable in this scenario since the attack only works for those that click on the carefully crafted attack URL. Regular website visitors will not be attacked, so chances are higher the attack stays undetected.

3 Combining stored and reflected XSS

Consider a website *bar.com* that allows login via GET parameters, which means you can provide username and password via URL. The following snippet of *index.php* shows the login process:

```php
$username = base64_decode($_GET['username']); // get
    username from URL
$password = base64_decode($_GET['password']); // get
    password from URL
if (!$username or !$password) { /* not logged in */}
else {
  $sql = "SELECT * FROM users WHERE username = '"
  . $username . "' AND password = '" . $password . "';";
  $result = pg_query($sql);
  if (pg_num_rows($result) > 0) {
        echo "Welcome " . pg_fetch_result($result, '
           username') . "!";
    /* successful login */
  }
  else {
    /* authentication failed */
  }
}
```

If, for example, a registered user *foo* exists and has chosen *bar* as the password, the following URL can login this user ...

```
http://bar.com/index.php?username=Zm9v&password=YmFy
```

... because `Zm9v` is the base64 encoding of *foo* and `YmFy` is the base64 encoding of bar.

You may assume that you can register new users and choose their passwords.

(3 points) (a) Identify the code line that provides the reflection necessary for an reflected XSS attack and explain why this line of code makes an attack possible.

Solution:

The line responsible for this vulnerability is

```php
        echo "Welcome " . pg_fetch_result($result, '
           username') . "!";
```

The username is reflected back by writing it to the webpage that is generated for the user. This is exploitable by choosing the username to be javascript code.

(15 points) (b) Describe an attack that will execute Javascript code to open an alarm window in the victims browser. What preparations do you have to do before sending out a link to a victim? How does the attack URL look like?

Solution:

The general idea is to choose a username that is actually executed as javascript code. As it needs to be embedded into the webpage, we choose ...

```
<script>alert('PWNED!');</script>
```

... to be our username. In order to reach the reflection point, we register the above username with some password we know, e.g. *known_pw* (stored XSS).

Now we can craft the exploit URL by providing base64 encoded versions of our attack code (username) and *known_pw* as a password (reflected XSS).

```
base64("<script>alert('PWNED!');</script>") ->
    PHNjcmlwdD5hbGVydCgnUFdORUQhJyk7PC9zY3JpcHQ+
base64("known_pw") -> a25vd25fcHc=
```

So the final attack URL now looks like this:

```
https://bar.com/index.php?username=
    PHNjcmlwdD5hbGVydCgnUFdORUQhJyk7PC9zY3JpcHQ&
    password=a25vd25fcHc=
```

(2 points) (c) Beside the apparent XSS vulnerability, the login mechanism is very insecure and badly designed for a variety of reasons. For example, it is a horrible idea to provide username and password in cleartext in an URL.
Give at least one example scenario where logging in via URL leaks the used credentials.

Solution:
One example is a webproxy that caches requests to websites. In this case, whoever has access to the proxy can simply read your credentials and impersonate you.
Another example is sharing links to websites via social media or messengers. If you blindly share a webpage, the complete URL will be used and therefore whoever receives this message also receives your credentials.