Software Security – Assignment 2: Analysis of a Web Application

Information: I also scanned the hacked web application using Nessus (http://www.tenable.com/products/nessus) in the web pages initial state and afterwards again when the security holes of the application where fixed. The scan results are provided in the same zip file. Besides that I also uploaded a diff file created with git that shows the the difference between the initial state of the web application and the state after it was fixed. One exception for this is that the databaseConnect1() function was already created before the git folder was initialized, and therefore it doesnt show the difference compared to the original state correctly.

Attack timeline

1st:

First the attacker tried to find a login screen, which could be found in the url/login.php. After this the attacker tried to login to the system. There are two different ways how the attacker can have logged in:

1. The attacker used SQL injection: A valid login can be made by putting an arbitrary username, and in the password field the following (without “ signs): “ haha' or 'x'='x “. This grants the attacker access to the adminstration zone of the web page.
2. The attacker could also just have guessed the username and passwords. The username that the attacker used to post the animal rights pictures was “admin” and the password “password” (in case the attacker did not change theses. Anyway the password being used is one of the worst and could be easily guessed.

2nd:

The attacker used the profile tab to change the name of the admin to First Name: “Worst Admin” and Last Name: “EVAR!!!!”. The attacker could also just have listed other users by modifying the url provided:

/profile.php?id=1 – gives the admin

/profile.php?id=2 – gives Gordon Brown

/profile.php?id=0 – gives test

This way the attacker could have modified any account for the website. Also the passwords are provided in plain text which makes the job even easier.

3rd:

After being able to login, the attacker tested if he really is able to make a post to the frontpage of the website (in case the test-post was made by the attacker and not the real admin).

4th:

The attacker tested if he is able to do x-site-scripting by posting a javascript to the content which worked also (this can be seen by opening the “you got haxord” news an javascript alert box comes up).

5th:

The attacker tried to upload other files than pictures to the article which worked also (see Got that shell!). The attacker uploaded a file called shell.php. It can be opened by entering the uploaded files url which is /uploads/shell.php. The shell.php provides many interesting stuff for the attacker such as a reverse shell and really detailed information about the hacked system such as OS, PHP Version, loaded Apache modules, users and many more. This information can be used to find new vulnerabilities in the hacked web server.

Explain if the attackers could have gone beyond defacing the website and what they might have managed to do.

1. know user accounts → start bruteforcing passwords and gain root (which would be possible since the root pw is "password")
    1. if they had gained root access they could have installed rootkits and do basically whatever they want
2. Even without getting the root access to the system they basically had all the web page in their control meaning they could have adjusted it for their needs and this way attack people visiting the web site.
    1. For example: infect users with malicious code to make the zombie computers and add them to attackers bot network
    2. Fool the users in giving some private information about them in the name of Business Worldwide Solutions
3. Get information from all user accounts including plaintext passwords
    1. Since the sql database root username and password where visible in the source code (in businessPage.inc.php)

Categorize the vulnerabilities acording to the OWASP Top Ten. (These may also give you hints, in case you have missed some vulnerabilities.) Fix the web app, so all holes are closed. Try to keep your solution as simple as possible and explain what you did & where to fix the holes.

1. Injection

The application included several places for possible SQL injection. This was due to the use of old unsecure way of running SQL queries with mysql_query()-function. This problem occurred in all files doing SQL queries (login.php, backend.php, profile.php, businesspage.inc.php), from which probably the most critical one was the login.php, which eventually allowed hackers access to the websites adminstration functionality. In the pictures below the corrections to the SQL-queries are shown, all parts (that do not include user input in queries) do not cause an injection thread but were corrected to use the new more secure and preferred way of handling SQL-queries in PHP.

```
-        $qry = "SELECT * FROM `users` WHERE user='" . $user . "' AND password='" . $pass. "';";
-
-        $result = @mysql_query($qry) or die('<pre>' . mysql_error() . '</pre>' );
+        $pdo = databaseConnect1();
+        $stmt = $pdo->prepare('SELECT password FROM users WHERE user = :user');
+        $stmt->execute(array('user' => $user));
+        $correct=0;
+        foreach ($stmt as $row) {
+            if (password_verify($pass, $row[0])){
+                $correct = 1;
+                break;
+            }
+        }

-        if( $result && mysql_num_rows( $result ) >= 1 ) {       // Login Successful...
+        if( $correct == 1 ) {   // Login Successful...
```

Illustration 1: SQL-Injection - login.php

```
              $published=gmdate("Y-m-d H:i:s");
-             databaseConnect();
-             $query = "INSERT INTO article (headline,content,created,published) VALUES ('$title','$content','$published','$name');";
-             mysql_query($query) or die('<pre>' . mysql_error() . '</pre>' );

+             $pdo = databaseConnect1();
+             $stmt = $pdo->prepare('INSERT INTO article (headline,content,created,published) VALUES (:title, :content, :published, :name)');
+                         $stmt->execute(array('title' => $title, 'content' => $content, 'published' => $published, 'name' => $name));
              }
```

Illustration 2: SQL-Injection - backend.php

```
-
-        databaseConnect();
-        $result = mysql_fetch_row(mysql_query("SELECT user_id FROM users where user='".$pUsername."'"));
-
-        $businessSession['id'] = $result[0];
+
+    $pdo = databaseConnect1();
+        $stmt = $pdo->prepare("SELECT user_id FROM users where user=:uname");
+        $stmt->execute(array('uname' => $pUsername));
+        $result;
+    foreach ($stmt as $row) {
+        $result=$row;
+        break;
+    }
+
+    $businessSession['id'] = $result[0];
```

Illustration 3: SQL-Injection - businessPage.inc.php

```
$pdo = new PDO('mysql:host='.$_BUSINESS[ 'db_server' ].';dbname='.$_BUSINESS[ 'db_database' ].';charset=utf8', $_BUSINESS[ 'db_user' ],
$pdo->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
return $pdo;
```

Illustration 4: SQL-Injection - connection establishment to database businessPage.inc.php

```
function fetchArticles() {
-    databaseConnect();
-    $result = mysql_query("SELECT headline, created, published, id FROM article");
-
-    $articles = '<lu>';
-
-    while($row = mysql_fetch_row($result)){
-
-        $curauth = mysql_query("SELECT first_name, last_name FROM users WHERE user_id =".$row[2]);
-        $name = mysql_fetch_row($curauth);
-        $articles .= "<li><h3>$row[1]<br /><a href=\"showArticle.php?id=$row[3]&comments=0\">$row[0]</a></h3><p>$name[0] $name[1]</p></li>";
-    }
-
+        $pdo = databaseConnect1();
+        $stmt = $pdo->prepare("SELECT headline, created, published, id FROM article");
+        $stmt->execute();
+        $articles = '<lu>';
+        foreach ($stmt as $row) {
+            $curauth = $pdo->prepare("SELECT first_name, last_name FROM users WHERE user_id = :uid");
+            $curauth->execute(array('uid' => $row[2]));
+            foreach ($curauth as $name) {
+                $articles .= "<li><h3>$row[1]<br /><a href=\"showArticle.php?id=$row[3]&comments=0\">$row[0]</a></h3><p>$name[0] $name[1]</p></li>";
+            }
+        }
    $articles .= '</lu>';
    return $articles;
}
```

Illustration 5: SQL-Injection - businessPage.inc.php

```
 function fetchArticle($id, $commentson){
-    databaseConnect();
-    $result = mysql_fetch_row(mysql_query("SELECT id, headline, content, created, published FROM article where id=".$id));
-
-    $curauth = mysql_fetch_row(mysql_query("SELECT first_name, last_name FROM users WHERE user_id =".$result[4]));
-
-    $comments = mysql_query("SELECT user, comment FROM comments WHERE articleid =".$id);
-    $num_comments = mysql_num_rows($comments);
-
+
+
+        $pdo = databaseConnect1();
+        $stmt = $pdo->prepare("SELECT id, headline, content, created, published FROM article where id=:id");
+        $stmt->execute(array('id' => $id));
+        $result;
+        foreach ($stmt as $res) {
+                $result = $res;
+                break;
+        }
+
+        $stmt = $pdo->prepare("SELECT first_name, last_name FROM users WHERE user_id = :uid");
+        $stmt->execute(array('uid' => $result[4]));
+        $curauth;
+        foreach ($stmt as $res) {
+                $curauth = $res;
+                break;
+        }
+
+        $comments = $pdo->prepare("SELECT user, comment FROM comments WHERE articleid = :id");
+        $comments->execute(array('id' => $id));
+        $num_comments=0;
+        foreach ($stmt as $res) {
+                $num_comments++;
+        }
```

Illustration 6: SQL-Injection - businessPage.inc.php

```
 function postComment($message, $name, $id) {
-    databaseConnect();
-    $query = "INSERT INTO comments (comment,user,articleid) VALUES ('$message','$name','$id');";
-   mysql_query($query);
+    $pdo = databaseConnect1();
+    $stmt = $pdo->prepare("INSERT INTO comments (comment,user,articleid) VALUES (:msg, :name, :id)");
+        $stmt->execute(array('msg' => $message, 'name' => $name, 'id' => $id));
 }

 function getAuthors() {
-    databaseConnect();
-    $result = mysql_query("SELECT user_id, user FROM users;");
-
+    $pdo = databaseConnect1();
+    $stmt = $pdo->prepare("SELECT user_id, user FROM users");
+        $stmt->execute();
+
     $users = "<select name='user'>";
-
-    while($row = mysql_fetch_row($result)){
+
+    foreach($stmt as $row){
         $users .= "<option value='" . $row[0] . "'>" . $row[1] . "</option>";
     }
```

Illustration 7: SQL-Injection - businessPage.inc.php

```
+               if( $correct == 1 ) {   // Login Successful...
+
+
+                   $stmt = $pdo->prepare("SELECT first_name, last_name, user, password FROM users WHERE user_id = :uid");
+                   $stmt->execute(array('uid' => sessionGrab()['id']));
+                   $num_results=0;
+                   foreach ($stmt as $row) {
+                       $num_results++;
+                   }
+
+                   $pw_hash = password_hash($_POST['txtPass'], PASSWORD_DEFAULT);
+                   if($num_results > 0) {
+                       $stmt = $pdo->prepare("UPDATE users SET first_name=:txtFirst, last_name=:txtLast, user=:txtUser, password=:password WHERE user_id = :uid
+                       $stmt->execute(array('txtFirst' => $_POST['txtFirst'],'txtLast' => $_POST['txtLast'],'txtUser' => $_POST['txtUser'],'password' => $pw_ha
+                   } else {
+                       $stmt = "INSERT INTO users (first_name,last_name,user,password) VALUES (first_name=:txtFirst, last_name=:txtLast, user=:txtUser, passwor
+                       $stmt->execute(array('txtFirst' => $_POST['txtFirst'], 'txtLast' => $_POST['txtLast'], 'txtUser' => $_POST['txtUser'], 'password' => $pw
+                   }
+               }
                }
 $profile = "";
+$pdo = databaseConnect1();
+$stmt = $pdo->prepare("SELECT first_name, last_name, user, password FROM users WHERE user_id = :uid");
+$stmt->execute(array('uid' => sessionGrab()['id']));

-if(isset($_GET['id'])) {
-    databaseConnect();
-    $profile = mysql_fetch_row(mysql_query("SELECT first_name, last_name, user, password FROM users WHERE user_id =".$_GET['id']));
+foreach ($stmt as $row) {
+    $profile=$row;// do something with $row
+    break;
 }
```

Illustration 8: SQL-Injection - businessPage.inc.php

As seen in the code samples above, instead of running the SQL queries directly on the database with the users input – a more sophisticated way is used: prepared statements and parameterized queries with the help of PDO.

Besides SQL-Injection one OS-Command injection flaw existed. The flaw and solution are shown in the picture below.

```
+
         $menuHtml .= '</ul>';
         $city = 'Vienna';
         if(isset($_POST['btnSubmit']))
-            $city=$_POST['txtCity'];
+            $city=filter_var($_POST['txtCity'], FILTER_SANITIZE_STRING);
         $menuHtml.="Get weather for your city:";
-
+
         $menuHtml.="
-
+
     <form enctype=\"multipart/form-data\" method=\"post\" name=\"userupdate\" onsubmit=\"return validate_form(this)\">
-
+
     <input name=\"txtCity\" type=\"text\" size=\"30\">
     <input name=\"btnSubmit\" type=\"submit\" value=\"Submit\" onClick=\"return checkForm();\"><br/>
     </form>";
-
-        $cmd = shell_exec( 'echo Weather report for:' . $city . ': ;shuf -i 0-40 -n 1; echo °C ' );
-
+
+        $cmd = "Weather report for:".$city.": ".rand(0,40)." °C";
```

Illustration 10: OS Command injection - businessPage.inc.php

This flaw could be exploited by putting “;” in the city name field then adding some OS commands that would be executed on the target system. For the functionality whished to achieve with the OS commands there also exists php functionality that can achieve that – so as a solution for this flaw I removed the OS Command execution and used php. Also the possibility for XSS was removed by filtering the city input.

2. Broken Authentication and Session Management

The first very big flaw in this section was having totally insecure passwords for the user accounts, but also for the server itself (for example username: admin password: password). These kind of flaws should be fixed by educating the users/adminstrators of the application about the problems with insecure password. Also some password rules (minimum length, must include special characters etc) could be implemented by the website itself.

Another big flaw in this part was, that the passwords stored in the database are not hashed correctly, but stored in plain text format. In this way if the attacker gets his hands on the database with the user data he can easily see the username and password combinations. This is even worse if the users use the same passwords/usernames on other web pages which are also exploited when this happens. To fix this I upgraded the servers PHP version to 5.5. This version of PHP provides functionality to hash passwords securely (including salt values).

To check login password against the password hash stored in the database the following functionality was used (in login.php):

```
if (password_verify($pass, $row[0])){
```

Illustration 11: password hash verification - login.php

This build in php function checks whether the given password ($pass) matches the has stored in the database ($row[0]).

Storing the passwords in hash format in the database the profile.php had to be updated:

```
$pw_hash = password_hash($_POST['txtPass'], PASSWORD_DEFAULT);
```

Illustration 12: password hashing - profile.php

This $pw_hash variable is then stored to the database instead of the plain text password. The password_hash is a php5.5 core function which uses a password hash using a strong one-way hashing algorithm. The second parameter defines which algorithm should be used for hashing (PASSWORD_DEFAULT means bcrypt algorithm, and its designed to change over time when stronger algorithms are invented).

The default password storage size on the mysql database was too small, so the size for it had to be made bigger. I used 255characters which is also preferred in the PHP documentation.
The size was updated with the following command: `ALTER TABLE users MODIFY password VARCHAR(255);`

In the picture below the mysql database is shown to represent the difference between hashed password and plain text passwords in the database:

```
mysql> select * from users;
+---------+------------+-----------+---------+--------------------------------------------------------------+
| user_id | first_name | last_name | user    | password                                                     |
+---------+------------+-----------+---------+--------------------------------------------------------------+
|       0 | test       | test      | test    | test                                                         |
|       1 | master     | admin     | admin   | $2y$10$NYRBJvS2cA3xBTP9eX/81u6DzqLkOngXtsaY5rC52aZ8d4DbwcJDy |
|       2 | Gordon     | Brown     | gordonb | gordi1                                                       |
+---------+------------+-----------+---------+--------------------------------------------------------------+
3 rows in set (0.00 sec)
```

Illustration 13: password hashed vs plaintext

One more bad security fault was that the server didnt have any timeout for the users session, meaning the user stayed logged in if he did not properly logout but just closed the window. This was solved by invalidating the users session if it hadn't been active for a certain time (15min in this case). Also the users session id is regenerated every 15 minutes, to avoid attacks on the users session (such as session fixiation).

```
+if (isset($_SESSION['LAST_ACTIVITY']) && (time() - $_SESSION['LAST_ACTIVITY'] > 900)) {
+    // last request was more than 15 minutes ago
+    logout();
+}
+$_SESSION['LAST_ACTIVITY'] = time(); // update last activity time stamp
+
+if (!isset($_SESSION['CREATED'])) {
+    $_SESSION['CREATED'] = time();
+    $_SESSION['TOKEN'] = md5(uniqid(rand(), true));
+} else if (time() - $_SESSION['CREATED'] > 900) {
+    // session started more than 15 minutes ago
+    session_regenerate_id(true);    // change session ID for the current session an invalidate old session ID
+    $_SESSION['CREATED'] = time();  // update creation time
+}
```

Illustration 14: Session timeout - businessPage.inc.php

3. XSS

XSS scripting occurred on the web page when the user posted news with some malicious javascript code in it. This was fixed by sanitizing the user input in the backend.php.

```
$content = filter_var($_POST['mtxContent'], FILTER_SANITIZE_STRING);
```

Illustration 15: filter user input - backend.php

The filter_var with the FILTER_SANITIZE_STRING basically removes different html tags from the user input inorder to prevent XSS.

4. Insecure Direct Object Reference

In the web application it was possible to go through all user profiles by editing the following link http://host/profile.php?id=2. The user was able to just change the number in the id field and afterwards was able to access other users information and even change their passwords and usernames. This was prevented by not trusting the user input for the id, meaning the profile.php is only loaded for the id of the logged in user. Besides that also a password box was added to the profile.php page so in order to make changes to user data the current password would have to be entered (in the beginning the password was shown in plain text when opening this page – this functionality was also removed).

```
 $profile = "";
+$pdo = databaseConnect1();
+$stmt = $pdo->prepare("SELECT first_name, last_name, user, password FROM users WHERE user_id = :uid");
+$stmt->execute(array('uid' => sessionGrab()['id']));

-if(isset($_GET['id'])) {
-    databaseConnect();
-    $profile = mysql_fetch_row(mysql_query("SELECT first_name, last_name, user, password FROM users WHERE user_id =".$_GET['id']));
+foreach ($stmt as $row) {
+    $profile=$row;// do something with $row
+    break;
 }
+
```

Illustration 16: insecure direct object references - profile.php

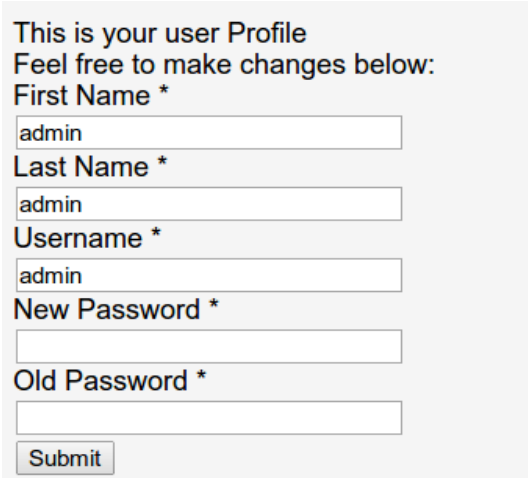

Illustration 17: web page profile.php

```
-    Password *<br /><input name=\"txtPass\" type=\"text\" size=\"30\" maxlength=\"32\" value=\"". $profile[3] . "\"><br/>
+    New Password *<br /><input name=\"txtPass\" type=\"password\" size=\"30\" maxlength=\"32\" value=\"\"><br/>
+    Old Password *<br /><input name=\"txtOldPass\" type=\"password\" size=\"30\" maxlength=\"32\" value=\"\"><br/>
+    <input type=\"hidden\" name=\"CSRFToken\" value=\"".$_SESSION['TOKEN']."\">
```

Illustration 18: profile.php old password

```
if (password_verify($_POST['txtOldPass'], $row[0])){
```

Illustration 19: profile.php verify old password

5. Security Misconfiguration

Server software was out of date (11 security updates available). This added a thread as an attacker could just have googled attacks against these vulnerable versions in order to hack the system. This was solved by upgrading the system accordingly (sudo apt-get update && sudo apt-get upgrade).

Also sensitive information was shown in the web applications source code (mysql username and password). This kind of information should never be included in the source code but added to an additional configuration file. Besides that root access was used for the sql queries made – better approach would have been creating a specific profile for the use in the webpage with the minimum priviledges possible.

The server provided error messages when doing an invalid SQL query and showed them to the user. This was fixed by simple not showing the mysql.error messages to the user.

```
mysql_query($query) or die('<pre>' . mysql_error() . '</pre>' );
```

6. Sensitive Data Exposure

Passwords were not hashed (See 2. Broken Authentication and Session Management).

Besides that the server also did not use TLS/SSL to encrypt the traffic between the client and the server. This means the passwords are transmitted to the server in cleartext and could be sniffed if an attacker was for example in the same open WLAN as the victim user. To overcome this problem the server should use https for the communication with the clients.

7. Missing Function Level Access Control

These vulnerabilities were not found in the web application.

8. CSRF

An attacker could make a malicious website and try to trick user to post stuff through that to the target website. This could have basically happened in profile.php and also in backend.php. To prevent this a random session token is created during the beginning of each session. This token is added to all the forms to be submitted as hidden values, and to be able to use these forms the froms hidden value has to match the session token.

```
+if (!isset($_SESSION['CREATED'])) {
+    $_SESSION['CREATED'] = time();
+    $_SESSION['TOKEN'] = md5(uniqid(rand(), true));
```

Illustration 20: Session token created on session startup businesspage.inc.php

```
if (isset($_POST['btnSubmit']) && $_POST['CSRFToken']==$_SESSION['TOKEN']) {
```

Illustration 21: the hidden_value has to match the session token for the submit to be valid (backend.php & profile.php)

```
<input type=\"hidden\" name=\"CSRFToken\" value=\"".$_SESSION['TOKEN']."\">
```

Illustration 22: Adding hidden_value to the form (backend.php & profile.php)

9. Using Components with Known Vulnerabilities

See Section 5. Security Misconfiguration.

10. Unvalidated Redirects and Forwards

The website included redirects for the external references shown in the articles. This way an attacker could have used the web page to attract a victim user to his own malicious website. This was fixed by removing the redirect.php file totally, and instead of using redirects to show the externa web pages just normal links were used.

```
-                   $content.="<br /><a href=\"redirect.php?url=". $_POST['txtRef'] ."\">External Reference </a>";
+                   $content.="<br /><a href=\"".$_POST['txtRef']."\">External Reference </a>";
```

Illustration 23: backend.php redirect vs ordinary linking

OTHER

When a user uploads a picture to be included to the article it is not verified if the file to be uploaded actually is a picture. This vulnerability was also used by the hackers to upload the shell.php file to the server. To avoid this the file type has to be verified before accepting the uploaded file.

```
                    $target_path = WEB_PAGE_TO_ROOT."uploads/";
                    $target_path .= basename( $_FILES['uploaded']['name']);
-                   if(!move_uploaded_file($_FILES['uploaded']['tmp_name'], $target_path)) {
-
+
+                   $imageData = @getimagesize($_FILES['uploaded']['tmp_name']);
+                   if($imageData === FALSE || !($imageData[2] == IMAGETYPE_GIF || $imageData[2] == IMAGETYPE_JPEG || $imageData[2] == IMAGETYPE_PN(
+                   $html .= '<pre>';
+                           $html .= 'Your image was not uploaded - wrong file format.';
+                           $html .= '</pre>';
+               }
+
+                   else if(!move_uploaded_file($_FILES['uploaded']['tmp_name'], $target_path)) {
```

Illustration 24: backend.php - upload validation

The file type verification is done by using the getimagesize PHP core function, which returns the size of an image. It also informs if the file provided in the parameters is not an image. For this solution I also added that only JPEG, GIF and PNG images are accepted, but this can be modified according to the companies needs.

Another rather severe bug is provided by the designCheck.php which provides a platform for testing. Basically the url injected for example as below,

/designCheck.php?sample=/etc/passwd

this would show the content of the /etc/passwd file on the web page. Basically an attacker could this way read arbitrary files from the file system which are not read protected. As a solution for this I would delete the whole testing section from the live system, as the vulnerability is quite severe and the advantage of this implementation is rather questionable.