SECURITY ASPECTS OF A PHP/MYSQL-BASED LOGIN SYSTEM FOR WEB SITES

Version 1.0, 23 Feb 2004

Author: Ole Kasper Olsen, ole.olsen@hig.no

Abstract

Many feel the need to protect something which they have made available on the Internet by typically a username and password-based login control. Having identified the need for such protection, many will typically download the needed script from an Internet script database.

Having downloaded the script, it's usually a simple matter to implement it and get it to work. And if it seems to work, it works, right? Not quite. What this article aims to do, is to explain and discuss the security aspects of most PHP session and MySQL based login systems.

This article does not detail how one can program such a login system, but full featured PHP code is enclosed. This code will be used as a starting point for discussion about various security-related topics. Thus, the focus of this article is mostly aimed at the underlying logic and technology of the code presented, and it is assumed a certain level of PHP knowledge, although it is mostly not necessary to grasp the security implication of various techniques.

Security on the Internet is all about making the weakest link in the chain as strong as possible. An inhumanly difficult task, but this article will hopefully give you some pointers to how you can reinforce *some* of those weak links.

1 Network Security and Server Administration

Network security and related subjects are somewhat out of the scope of this article, and will not be discussed in great detail. There are, however, some things anyone who wishes to set up a secure Web site needs to know about, especially considering that the Internet and the Web is an inherently insecure place to store sensitive material, and good server side code will sadly not be enough.

First, any communication between a client computer (i.e., the Web surfer's computer) and the server is unencrypted. Even though passwords are displayed with asterisks as you the user types them in, they are sent over the net in plain text. Thus, if an attacker is eavesdropping on your communication link, the password will be extremely vulnerable. To combat this, you will need to be able to encrypt data between the client and server. The most straight-forward way of doing this on the World Wide Web, is by utilizing SSL[1]/TSL[2] encryption, which provides symmetric key exchange via public key cryptography and subsequent symmetric encryption of the communication link[3].

[1] Secure Sockets Layer – http://wp.netscape.com/eng/ssl3/draft302.txt
[2] Transport Layer Security – the successor to SSL – http://www.ietf.org/rfc/rfc2246.txt
[3] See section 4.3.2.2.2 for more incentives for protecting the communication between client and server.

Also take care if you are setting up your own server and are not a seasoned IT technician. Many people don't read the documentation good enough, and subsequently they don't know that MySQL is installed with a blank administrator password by default. Don't make the same mistake with your servers. Further, is your server publicly available on the Internet? Does it need to be? If the Web server is the only machine which requires access to the database server, you can keep it behind a firewall and restrict access only to the Web server.

2 Technology Introduction

Enclosed you will find an archive with some example source code in PHP (hereafter referred to as "the script"), which will be the basis of the entire article. The script is based around a MySQL database which contains data about the users of a secured Web site, each user stored with a username, password and some other important personal data. The script can validate a user by asking for username and password, and then test that combination against the information in the database. If the provided user credentials exist in the database a HTTP session[4] for the user will be created, and relevant information about the user and the session will be saved in session variables. This will allow the user to access protected pages. The user can subsequently log out if she so wishes. The script can also add new users to the database.

3 Database

Before we start looking at the script, there are a few important concerns about the database server and the way we are to store information about the users which need to be dealt with.

The first which needs to be decided is in which form we are to store the users' passwords in the database.

3.1 Storing Passwords

Never, ever store users' passwords unencrypted on a database server, unless you have an extremely good reason to do so. Doing so makes it extremely easy for an attacker who compromises your server to get away with your users' passwords. Additionally, do you for certain know who has access to the database? In many cases, the database server is part of a Web hosting package, and there's no telling who has access to read the database contents. If the server is yours and you are certain you are the only one with access to its contents the situation is somewhat different, but you still have the scenario where an attacker is able to breach your defences. There's also the element of being able to tell your users that their passwords are stored securely.

Intuitively, many would say that the solution would be to encrypt the passwords using standard symmetric encryption like 3DES or AES[5], and then decrypt them for a

[4] http://www.ietf.org/rfc/rfc2965.txt
[5] http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

short period when comparing what the user wrote to what is actually stored. Symmetric encryption requires a key for encryption and decryption. This key would have to be placed in a readable form on the machine doing the encryption and decryption (i.e., the Web server). Equipped with this key, an attacker (in many cases the system's administrator which obviously always knows the key) can easily decrypt the passwords.

Additionally, symmetric encryption use heavy algorithms and is very slow. The most common solution to this problem, on the other hand, is extremely fast and efficient.

And that solution is the use of cryptographic hash functions. A cryptographic hash function will take a message of totally arbitrary length (for example a password) as input and produce a fixed length output which is often called a message digest. For our uses, the most interesting characteristic of a cryptographic hash functions is that it is what is called *preimage resistant*. In plain English, this means that for any given message digest, it is computationally infeasible (i.e., it would take too long) to find the original message. The hashing of a message can be seen as a one-way procedure.

The security of a hash function is by large given by its message digest's length, because the only viable approach for an attacker to reverse the hash (i.e., find the original message from the message digest or find a message which hashes to the same message digest) is to carry out some form of brute force attack. If we assume the message digest to be 160 bits long, this task has the complexity of 2^{160}. That is, in the worst case, an attacker has to try 2^{160} different messages before she finds one which hashes to the same message digest as the password in question. With today's PC efficiency she'll still be searching for a match several millennia from now.

One such cryptographic hash function is SHA-1 (Secure Hash Algorithm, first revision)[6], which is the one being used in the script, as you'll see later on.

4 Script

Even though quite limited in lines of code, the session-based login script is actually quite full of security concerns.

4.1 Common Pitfalls

We first start up by introducing some common pitfalls which you will find in most of the script's files.

4.1.1 Including Files

A very common way to compartmentalise and organise PHP code is to divide the code up into different files, and then include those files into the file being currently executed. For example, in the enclosed script you will find the code for connecting to a database in the file "includes/db.inc.php". Subsequently, every file which needs database connectivity will have to include that file to gain access to the function

[6] http://www.itl.nist.gov/fipspubs/fip180-1.htm and http://www.ietf.org/rfc/rfc3174.txt

which connects to the database. This is done by using PHP's `include` or `require` construct[7]:

```
require("includes/db.inc.php");
```

What these constructs do, conceptually, is pasting the content of the included file into the file which the include statement is being run from. All this is done server side, so what is the security implication?

Well, the problem isn't with the constructs themselves, but with careless naming of the files which are to be included. For some odd reason, it is very common to name the files after their intended use, not their content. Thus many people use the extension ".inc" on all their included files out of old habit. The security issue here is that someone might be able to guess the filename of the include-file on the Web server. That's perfectly fine if all the file contains is some pure HTML which you want to insert somewhere on your site via PHP or other technologies. But what if the file contains your database connection function, along with the needed database address and logon details as it does in the enclosed script? If that is the case, some lucky attacker just have to guess the filename, and she will have everything she needs to break into your database server by just calling up the file in her Web browser. At least if you give your include-files the ".inc" extension.

The solution to this problem is twofold: Either move the file out of the public HTML directory or name it with the ".php" extension so that the file is parsed by PHP before shown to the user. When the file is parsed by PHP it will most likely not print out anything, because the only content is a function which is never called (in that file).

4.1.2 Connecting To the Database Server

In the script, the database connectivity is provided by the function `db_connect` in the file "includes/db.inc.php". This file is included into any other file which needs database connectivity, and the implementation is as follows:

```
1   <?php
2   function db_connect() {
3      $host     = "sql.mysite.com";
4      $user     = "aabbcc";
5      $password = "ZuPerP45s!";
6      $database = "mysite";
7
8      $link_id = @mysql_connect($host, $user, $password)
                  or die($errmsg);
9      @mysql_select_db($database, $link_id) or die($errmsg);
10     return $link_id;
11  }
12  ?>
```

[7] The only difference between `include` and `require` is that where `include` will give a warning if the file was not found, `require` will stop the execution of the script.

On line 2 we declare the new function. The address and login credentials for this specific database server we declare on lines 3 through 6.

Further, line 8 makes the actual connection to the database using the credentials detailed in the lines above. The `mysql_connect` function returns a pointer to the connection if everything went well. We store this in the variable `$link_id`. If `mysql_connect` fails, PHP will kindly print out that it could not contact the database server, and then goes on to list the actual host address of the database server. Is this information we would like a potential attacker to know? Absolutely not! Therefore, to suppress the error message from PHP, we write a "@" (in PHP it's called the "Error Control Operator"[8]) in front of the function call. Then we supply our own, more generic error message via the `"or die"` construct which will run whenever the preceding function returns FALSE. Alternatively you could redirect the user (using `header("Loctaion: [newURL]")`) to a nice and helpful error page. Just don't give away too much information.

On line 9 we repeat the procedure with the error control operator, as we don't want an attacker to know which database on the database server we are using.

4.2 Registering New Users

To register a new user, we'll need an HTML form, which is to be found in "register_user_form.html". The file "register_user_action.php" is the part of the script which handles the form data. The form uses the POST method to transfer data to the script (i.e., the server on which the script resides). Using POST ensures that the data is passed in the HTTP header, rather than passed in the URL which is marginally more insecure.

First and foremost, we need to check to see whether or not the file was called in a correct manner, i.e. that it was called from the HTML form. This is done on line 14 in "register_user_action.php":

```
14  if(isset($_POST["submitUser"])) {
```

"submitUser" is the name of the submit button in the form, and by checking that it was passed to the script we can be fairly sure that the form was submitted correctly, and that no one has opened the file from somewhere else[9].

Further, on lines 15 through 19, the script checks to see whether the user has forgotten to fill out some of the fields, that both of the password fields have equal values and that the email address entered is valid. This is pretty straight forward, and can be implemented in a number of ways.

---

[8] http://www.php.net/operators.errorcontrol

[9] Many people will be able to construct and send their own HTTP headers with POST data, but this check is mostly just a user friendly gesture to prevent running of the script if no data is passed along. For added security, you should also check whether or not the user came from a safe page.

On lines 23 through 27 we are building up the SQL statement which is going to insert the new user into the database. There are some extremely important pitfalls to be aware of here.

4.2.1 SQL Injection Attacks

As you might know, the apostrophe (') is a reserved character in SQL as it's used as a string delimiter. Below is an example of bad (yet common) programming practice which will potentially endanger your system, where POST data from a form is used without any input validation.

```
if(isset($_POST["submit"])) {
  $insertStatement = "INSERT INTO users (username)
                      VALUES('" . $_POST["username"] . "')";
  mysql_query($insertStatement);
}
```

If the user provides a username like "John's son", `$insertStatement` would be parsed to contain the following:

```
INSERT INTO users (username) VALUES('John's son');
```

This will fail, because the SQL engine considers the string to be finished when it reaches the second apostrophe. The expected character after that is a closing parenthesis. However in our example, there's more text. The insertion will then fail. In a perfect world with no evildoers, this might not mean much, except that the user will have to come up with a username without an apostrophe. But what if an attacker can exploit this by providing a specially crafted username?

```
INSERT INTO users (username) VALUES('john'); TRUNCATE users;');
```

As you can see, the attacker provided the username "john'); TRUNCATE users;"[10]. In essence, the attacker has now divided up what was one SQL statement into three. In most cases the SQL engine will now try to execute the first statement (everything up to and including the first semicolon) and successfully insert the user "john" into the database. Then it will try the next statement ("TRUNCATE users"), which will empty the table "users" of **all** data. Finally, it will try the last statement which will fail.

As you can see, failing to correctly validate the input might put your entire database content at risk. We'll need to find a way to make sure that there's no way to prematurely terminate a string. We can do this by inserting an escape character (in SQL, this is a backslash (\)) in front of potentially dangerous characters. Luckily, PHP has a built in function for doing this: `addslashes`.

[10] This attack requires some knowledge about the SQL statement (to be able to match the closing parenthesis and number of input values), but a staggering amount of pages will actually print out the entire SQL statement if something fails (like when someone types in an apostrophe in some input box), making things *extremely* easy for attackers.

4.2.2 Cross Site Scripting

There's also another security hazard to consider, namely that which is commonly called cross site scripting. For example, an attacker can insert malicious JavaScript code or even PHP code rather than a name in the input field. This will have no effect when we store it in the database, but if you have for example a page which lists all users of the site, the code will be embedded into the HTML/PHP document, and executed. To combat this we'll have to either convert some special HTML characters (like "<" and ">") into harmless character entity references[11] or remove them completely.

4.2.3 The Resulting Code

Taking all the above into consideration, here's a quick tour of the resulting code.

```
24  $insertStatement.="'" .
          addslashes(htmlspecialchars($_POST["username"])) . ", ";
```

We append the username to the insert statement only after we have removed any possibility of cross site scripting (using `htmlspecialchars`[12]) and SQL tampering (using `addslashes`).

Next we have the password. As discussed in section 3.1, storing passwords in plain text is a risky business, and usage of a cryptographic hash function was advised. PHP has built in support for SHA-1 and MD5. MD5 was developed by Ronald Rivest at RSA in the early 1990s, and has lately been proven to have certain weaknesses[13] and is not recommended for use in new systems, which is why we'll use SHA-1 here. SHA-1 produces a 160 bit message digest whereas MD5's message digest is only 128 bits long.

```
25  $insertStatement .= "'" . sha1($_POST["password1"]) . ", ";
```

You might wonder why `addslashes` hasn't been used here. That is because PHP's `sha1` function is implemented in such a way that it will always return a string with exclusively hexadecimal numbers (0 – 9 and A – F), i.e. no characters which can be dangerous in an SQL statement.

The email address we have already checked with the `validateEmail` function on line 18, so we know it's free of hazardous characters. We'll just append it to the insert statement without any further ado.

```
26  $insertStatement .= "'" . $_POST["email"] . ")";
```

On line 27, the user's full name is handled the same way as we handled the username previously on line 24.

---

[11] HTML 4.01 Character Entity References http://www.w3.org/TR/html4/sgml/entities.html
[12] If you wish to strip away all HTML and PHP code, use the PHP function `strip_tags` instead.
[13] http://www.uni-mannheim.de/studorg/gahg/rweis/rgp/md5/dobbertin.ps

We then execute the insert statement to store the user in the database using the `mysql_query` function on line 28. Also here, make sure not to give away too much information in the error message in case the insertion should fail. Printing out the entire SQL statement or using the `mysql_error` function is **never** preferable, except when you are developing and/or debugging the script.

4.3 Logging in the User

The act of logging in the user may be the most crucial part of any secure Web site. There are a number of things which may go wrong, luckily though—many of which we can protect ourselves against.

4.3.1 More on SQL Injection Attacks

All parts of a secure database driven Web site which accepts input from the user is susceptible to SQL Injection Attacks. Thus, not only is the registration of the user vulnerable, but also the login sequence can contain serious security vulnerabilities. SQL Injection Attacks were discussed in detail in section 4.2.1, so an example will suffice here.

Suppose in your code you fail to validate user input, and later have

```
$stmnt = SELECT login_id FROM users WHERE user = '" .
       $_POST["username"] . "' AND password = '" . $_POST["password"];
```

Consider an attacker who wants to access the user jdoe's account. She provides a username which looks like this: "jdoe'; #". She then provides a blank password. Will she be able to gain access to the "jdoe"-account? Let's have a look at the username she has crafted. The last character in the username is "#". This may look innocuous enough, but that character is used in SQL for commenting out the rest of the code on the same line. The SQL statement, after it's parsed will look like this:

```
SELECT login_id FROM users WHERE user = 'jdoe'; #' AND password = ''
```

What has the attacker done? She has successfully circumvented the site's password check, because the part of the SQL statement which pairs the submitted password with the password in the database is commented out. All an attacker needs to know is a valid username. More often than not, usernames are publicly known.

The solution to this form of SQL Injection Attack is like in section 4.2.1—to properly validate input from users. As shown before, this can be easily done with the PHP function `addslashes`.

4.3.2 Sessions

The HTTP protocol which is used for communication between Web browsers and Web servers is a stateless protocol. This means that it doesn't maintain state between

requests. In other words, the HTTP protocol doesn't provide any features we can use for maintaining that a user is logged in.

The solution to this problem is sessions. A session enables a Web server[14] to retain information about a specific user between document requests. The information is stored on the server, so in theory it should be tamperproof as long as the server itself is not compromised. Each session currently existing on the server is given a unique identifier. The identifier is then sent to the user's Web browser, and is usually stored in a cookie. Whenever the user accesses a document on the server, the session identifier is automatically sent along with the document request[15]. If the requested document relies on data stored in a session, the server will make use of the session identifier to find the correct data in the correct session.

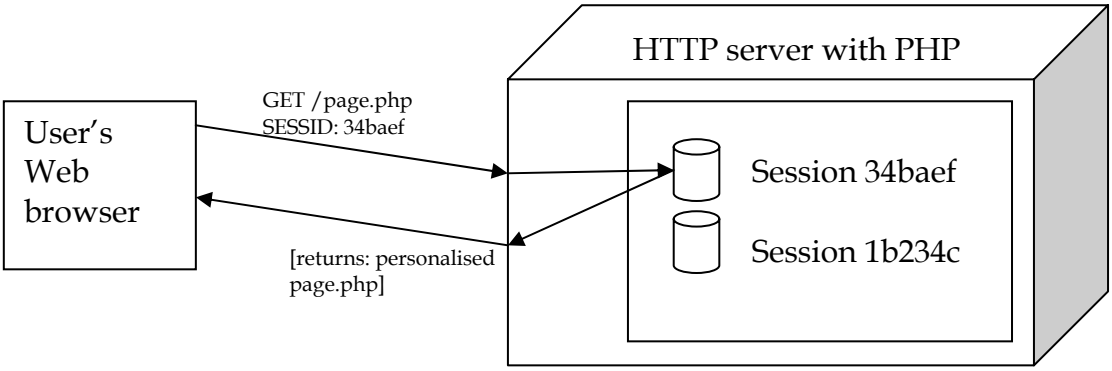

Figure1 - Web browser sends session identifier on each request to the server

4.3.2.1 The Vulnerable Session Identifier

As sessions are used for identifying users who are logged in, they are often the easiest way into a protected site for an attacker.

As described in section 4.3.2, the user is given a unique session identifier upon login. There are two main classes of session attacks and both focus on the session identifier. One is session hijacking where an attacker manages to learn a user's session identifier. The attacker can then successfully hijack the session by setting the session identifier in a cookie in her own Web browser or by other means provide the session identifier with each document request to the server. The other class of session attacks is called fixation attacks, where the attacker manages to fix or preset the session identifier before the user even logs in. As you can see, both classes of attacks target the session identifier. The session identifier is the most vulnerable part of any session based login system.

[14] In an Apache/PHP environment, it is PHP which provides the session functionality.

[15] If the user's Web browser doesn't support cookies, or the user has turned off the browser's cookie functionality, the session identifier can be transmitted to the server via the URL. This is generally not recommended as it makes the session identifier so much more accessible and visible. It is also extremely easy to get hold of the session identifier by launching a social engineering attack [2] against the user.

If an attacker can find the session identifier, she will be able to gain the same access as the victim, as long as the victim hasn't logged out, making the server properly destroy the session (see section 0). To gain access to protected areas of a Web server, all the attacker needs to do is provide the session identifier with each document request.

Remember, even though it's the server which assigns session identifiers, it cannot self control to whom it has given it to.

4.3.2.2 Session Hijacking

There are three main ways to hijack a session by uncovering the session identifier. These are by ways of interception, prediction and brute force.

4.3.2.2.1 Interception

An adversary who can monitor the network traffic between a targeted user and server can easily steal a session identifier as it travels over the network in clear text.

The best, and perhaps only, way to combat this problem is by using encrypted communication between the user and the server[16]. Remember though to set the cookie's security bit[17], so that if the user requests a document on the server which is not protected by encryption, the session identifier will not be sent.

4.3.2.2.2 Prediction

Prediction is perhaps most severe in situations where the attacker has legitimate access to a system, but wants to gain access to someone else's session. The attacker can then log on and off until she has a fair number of session identifiers which she can analyse. If the session identifiers have been badly chosen, a seasoned attacker will most probably be able to crack the scheme behind the session identifiers, and subsequently she will be able to craft the session identifier of any user. Often, the session identifier is just a monotonously incrementing number (i.e., a counter of some sort), other times it may be a self-devised encryption algorithm. Do not flatter yourself by thinking you can devise a good encryption algorithm.

To combat prediction one needs to use random or seemingly random session identifiers. In later versions of PHP the default session identifier is a 128 bit long cryptographically strong pseudorandom value calculated using MD5[18].

If you are serious about Web security, you may also bind the session identifier to something unique to the user (most often the best choice is the user's IP address as seen by the server) for better security. One can for example utilise an HMAC for doing this. An HMAC is very similar to a standard cryptographic hash function, only it takes two inputs, namely a message which can be the user's IP address, and a

---

[16] This is best done by utilising SSL or TLS as mentioned in section 1.

[17] In PHP, this can be done via the function `session_set_cookie_params` or the `session.cookie_secure` ini setting.

[18] In PHP v5 you can choose to use SHA-1 in place of MD5.

secret key only known by the server. The result will be a session identifier which is seemingly a random number, but at the same time the server will be able to do a control to find out whether the session is owned by the user or not.

4.3.2.2.3 Brute Force

Brute force guessing of session identifiers can only be done if the attacker can approximate the session identifier (to limit the exhaustive search) or if the developer of the secure Web site has been incredibly short sighted (by for example making the session identifier be a number between 1 and 10,000).

By using a cryptographically strong pseudorandom number with length of at least 128 bits as the session identifier, it will be computationally infeasible for an attacker to do a brute force attack on the session identifier.

4.3.2.3 Session Fixation

The fixation attack, as first described in [1], differs from the three previously mentioned forms of attacks in that it doesn't focus on obtaining the victim's session identifier, but rather making the victim use a session identifier defined by the attacker.

The attack exploits the fact that if the user already has a session identifier, a new session identifier will in many cases not be constructed during login. Rather than providing the user with a new session identifier, the server will just take the provided session identifier and create a new session with the same identifier.

If the attacker can set a cookie in the victim's Web browser which contains a session identifier which she already knows, gaining access to the victim's account will be elementary. The most convenient way for an attacker to set a cookie with a session identifier in the victim's Web browser is probably by cross-site scripting.

As a fixation attack relies on the Web server using an already provided session identifier, we'll have to force a regeneration of the session identifier during the login sequence if we are to thwart this scheme. The attacker will then not be able to gain access to the identifier, except by other means as explained in section 4.3.2.2.

4.3.3 The Resulting Code

For logging in, a user will have to go through an HTML form as the provided in "user_login_form.html". As with the registering, this form also uses the POST method to make the data available to "user_login_action.php", which will be doing the authentication work.

The first lines of code are pretty much the same as the one we saw when dealing with registering the users. On line 10 we check to make sure the form was submitted, and on lines 11 and 12 we make sure the fields were properly filled. After that we connect to the database and build up an SQL statement for fetching the user from the database on lines 13 through 18.

If you ever wondered how we were to match the hashed password in the database to the user-provided password, you'll find the solution on line 18. We need to hash the password the user submitted to be able to match it against the one already in the database. In other words, we are matching two hashed passwords against each other.

```
18  $loginQuery .= " password = '". sha1($_POST["password"]) . "'";
```

If we find a match in the database, i.e. the SQL statement returns exactly 1 row of data, we can begin the sequence of logging in the user. This is done on lines 26 through 30.

```
26  session_start();
```

The PHP function `session_start` creates a session if it doesn't already exist and prepares a cookie with the session identifier.

As explained in section 4.3.2.3, an attacker may launch a session fixation attack against a user. To hinder this, we'll have to regenerate the session identifier during the login, and modify the cookie which is to be sent to the user. `session_regenerate_id` will take care of this[19].

```
27  session_regenerate_id();
```

Next, we store some information we find in the database about the user in the session. Such information is useful if we later want to personalise the pages (full name and email address), or fetch more data from the database (login ID).

```
28  $_SESSION["user"] = new User($userData->login_id, $userData->name,
                                 $userData->email, $userData->username);
```

On the next two lines, we store the user's IP address and we timestamp the session, for use later on when controlling access.

```
29  $_SESSION["IP"] = $_SERVER["REMOTE_ADDR"];
30  $_SESSION["timestamp"] = time();
```

The user is now considered logged in by the system.

4.4 Protecting Content and Controlling Access

To make access control easy, all functionality associated with the login check is put in the file "includes/session.inc.php" which can then be included in every protected page.

[19] `session_regenerate_id` was introduced in PHP v4.3.2.

The easiest way to check if a user has access to a protected part of the site is simply to check if the session is properly set, and that it has some of the content which were set during login. In the script we first check to see if the global PHP session variables (in the array `$_SESSION`) are set. This is done simply to check if the expected "user" object is where we put it during the login sequence. If it is not, we stop execution of the script.

```
5   if(!isset($_SESSION["user"])) {
6      die("Not logged in");
7   }
```

Further we do a simple check to make sure that the "user" object is properly initialised. We do this by making sure the login ID is a number.

```
8   if(!is_numeric($_SESSION["customer"]->loginID)) {
9      die("Not logged in");
10   }
```

We have now assured that the owner of this session has properly gone through the login routine. As explained in session 4.3.2, we do not however know that the current user really is the owner of the session, as the session identifier is pretty vulnerable. Therefore we saved the user's IP address when she logged in, and on line 11 we compare the current IP address against the IP address we already have. If they mismatch, the user is logged out and asked to log in again[20].

```
11  if($_SESSION["IP"] != $_SERVER["REMOTE_ADDR"]) {
12     header("Location: user_logout.php");
13  }
```

Then we have the timestamp. If the timestamp is not older than 30 minutes (1,800 seconds), we reset the timestamp to the current time (to give the user another 30 minutes of browsing). If the timestamp is older than 30 minutes, we do as with mismatched IPs: we log out the user, and ask her to log in again.

```
14  if((time() - $_SESSION["timestamp"]) > 1800) {
15     header("Location: user_logout.php?timeOut=1 ");
16  }
17  else {
18     $_SESSION["timestamp"] = time();
19  }
```

4.5 Logging out the User

The logout sequence is extremely crucial. As previously explained, an attacker may gain access to a session by somehow obtaining the session identifier. If the session data on the server is not properly deleted when the user chooses to log out, the

[20] It is to be noted, however, that many ISPs will route their user's requests through different proxy servers, giving them a seemingly different IP based on which proxy server they have been routed through. As a consequence, legitimate users may be locked out. Such are the trade offs of security.

attacker may continue to use the session identifier ad infinitum (unless we have been smart enough to timestamp the sessions so that we can enforce a time limit).

So, it's important to destroy all data related to the session when the user logs out. This is done on lines 4 through 9 in `user_logout.php`.

```
5   $_SESSION = array();
```

This line effectively eradicates the entire content of the `$_SESSION` array by reinitialising it. This means that any data we had stored in the user's session is gone, including IP address and timestamp. We can no longer validate the user against the session. Therefore the owner (or any other with access to the session identifier) will not be allowed access to protected pages.

We then tell PHP to destroy the session. In theory, there will be no trace of the session left on the server after we've run the following line.

```
7   session_destroy();
```

We'll also delete the session identifier cookie on the user's Web browser. After all, it's redundant at this point.

```
9   unset($_COOKIE[session_name()]);
```

The user is now successfully logged out.

5 References

[1] KOLŠEK, MITJA – Session Fixation Vulnerability in Web-based Applications, 2002
[2] MITNICK, KEVIN – The Art of Deception, 2002