

James Frew DaAD PHP Assignment 1

by James Frew

Submission date: 04-Dec-2022 05:54AM (UTC+0000)

Submission ID: 190911421

File name: 31653_James_Frew_James_Frew_DaAD_PHP_Assignment_1_460452_931816492.docx (110.88K)

Word count: 1039

Character count: 5318

DATABASE-DRIVEN WEBSITE APPLICATION

ACCA5026

James Frew

Student University of Wales Trinity St David, Swansea, Wales

2108660@student.uwtsd.ac.uk

I. SQL Injection

SQL Injection occurs when code is injected into the system using an input field on a website. Attackers can use lines such as `" or ""="` to bypass security systems – usernames, passwords, etc. `" or ""="` changes the initial query from `"SELECT * FROM users WHERE username LIKE $input AND BINARY password LIKE $input2"` to `"SELECT * FROM users WHERE username LIKE $input" or ""=" AND BINARY password LIKE $input2" or ""="`. The initial query would be used to only allow the user to log into their account if the password and username were recognised by the database and matched. `" or ""="` now makes it so that if the user enters nothing (`"`), or if nothing = nothing (`"="`) – which is always correct – then the user gains access without actually knowing either username or password. `" OR 1=1"` would also work here. [1]

Similarly, in areas such as search features, `" OR 1=1"` can be used to show all records, rather than just the specified search.

Most dangerously of all attackers could use lines such as `“; DROP TABLE tableName”`. This would delete the whole table from the database and could be performed as many times as they like, to effectively wipe the entire database of all tables and records. [2]

The way to prevent this is by using a prepared statement – `"SELECT * FROM users WHERE username LIKE ? AND BINARY password LIKE ?"` where the input variables are replaced with `?`. This allows the variables to be forced as purely string (or whichever desired) values, making it so that `" or ""="`, or `"OR 1=1"` will be taken as text, rather than logic. The program would then read the entries, find that they weren't present in the database and refuse access to the user. [3]

1) Implemented Defence Against SQL Injection

(See Appendix A for screenshot)

First, the user's inputs from username and password are taken from the form. The query then prepares to check against the "users" database an input for "username" and an input for "password". No actual values are yet inputted into the query, just theoretical ones.

A statement is then prepared, and parameters bound to it. These parameters are the two string values (denoted by `"ss"`) and are taken from `$un` (username inputted) and `$pw` (password inputted).

The query is then executed, and the result recorded. If no records are found, the user is told that their entries are incorrect. Otherwise, the user has logged in correctly.

Prepared statements are used on every input field of the database.

II. Spam Bots

Spam bots (botnet) are automated robots that keep submitting data to your site, in order to overload it and cause it to crash. [4] This can be as simple as repeatedly making new user accounts or spamming a comments section with inappropriate comments. When it's only a few bots, they don't cause much of an impact on the site, but deploying hundreds can lead to a denial of service attack and the site going under.

Fortunately, there are multiple ways of countering spam bots. Honeypots are potentially the simplest method of defence and involve including decoy entry fields that only bots can see. As bots view the website code rather than the actual webpage, a text field on a form can be styled so that's its hidden from users, but bots will still see it. The bot will input data into this field whereas a human wouldn't be able to. When the form posts its data, if the hidden field has a value in it, the website can block the actions of the bot, as it knows that a human wouldn't have been able to input data. [5]

Some bots are clever and can detect the hidden styling and class assigned to the input field. This can be overcome by changing the name of the class and moving the styling to an external style sheet.

Another method for defending against spam bots is using a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart). CAPTCHAs work by presenting the user with pictures and being told to select all the pictures of a certain scene/object. Appendix

CAPTCHAs can also have text written in a very "stylised" fashion, which can unfortunately make it very difficult for people with dyslexia to read. The most advanced of CAPTCHAs will even record the movement your mouse made as the page was loaded and before you clicked on it. As bots usually will just move instantly to their desired location, the CAPTCHA recognises these movements as too fast or precise to be human and can block the bot. [6]

Additional random questions can be used to further throw off bots. This involves asking the user a simple question (such as basic mathematics or very common general knowledge questions,) and having them enter the answer in a textbox. Mathematical should be asked using words rather than numbers – e.g., "What is one plus four?", rather than "what is 1+4?" as this will make it harder for the bot to guess the answer.

2) Implemented Defence Against Spam Bots

(See Appendix B)

The user account creation form has an additional honeypot text input to catch spam bots (line 52). This is made invisible to humans by giving it the {display:none; visibility:hidden} style

(See Appendix C)

This has been implemented on the account login and account creation pages as these are most likely to be targeted by bots.

```

55     $un = $_POST["uName"];
56     $pw = $_POST["pWord"];
57
58
59     $query = "SELECT * FROM users WHERE username LIKE ? AND BINARY
password LIKE ?";
60
61     $stmt = mysqli_stmt_init($link);
62     mysqli_stmt_prepare($stmt, $query);
63     mysqli_stmt_bind_param($stmt, "ss", $un, $pw);
64     mysqli_stmt_execute($stmt);
65     $result = mysqli_stmt_get_result($stmt);
66
67     $numrows = mysqli_num_rows($result);
68
69     if($numrows == 0)
70     {
71         echo "Username or password incorrect<hr>";
72         ?>
73         <a href = "login.php">Try again</a>
74         <?
75     }
76     else
77     {
78         echo "Welcome Back ".$un;

```

A.

```

47 <div class="text">
48 <form action = "checkregister.php" method = "post">
49 Email: <input type = "mail" name = "eMail" maxlength = "100" required><br>
50 Username: <input type = "text" name = "uName" maxlength = "30" required><br>
51 Password: <input type = "password" name = "pWord" maxlength = "30" required
><br>
52 <input type="text" name="extra" class="human">
53 <input type = "submit" value = "Submit">

```

B.

```

3 <style>
4 .human{display:none; visibility:hidden}
5 </style>

```

C.

References

- [1] C. Ping, "A second-order SQL injection detection method," in Dec 2017, Available: <https://ieeexplore.ieee.org/document/8285104>. DOI: 10.1109/ITNEC.2017.8285104.
- [2] (). *SQL Injection*. Available: https://www.w3schools.com/sql/sql_injection.asp [Accessed 22/11/ 2022].
- [3] A. S. Sai Lekshmi and V. S. Devipriya, "An emulation of sql injection disclosure and deterrence," in Jul 2017, Available: <https://ieeexplore.ieee.org/document/8076787>. DOI: 10.1109/NETACT.2017.8076787.
- [4] (). *Botnet*. Available: <https://www.w3schools.in/ethical-hacking/botnet> [Accessed 22/11/ 2022].
- [5] Y. Yang, N. Vljic and U. T. Nguyen, "Web bots that mimic human browsing behavior on previously unvisited web-sites: Feasibility study and security implications," in Sep 2015, Available: <https://ieeexplore.ieee.org/document/7346921>. DOI: 10.1109/CNS.2015.7346921.
- [6] Shardul Vikram, "Using Novel Image-Based Interactional Proofs and Source Randomization for Prevention of Web Bots." , 2011.

James Frew DaAD PHP Assignment 1

GRADEMARK REPORT

FINAL GRADE

83/100

GENERAL COMMENTS

Instructor

WEBSITE

Database structure and design - A large number of table, suitably normalised 10/10

Range of PHP demonstrated -

Search is good - I hadn't initially realised that the second page was a sub-search with the category: you could signpost this to the user more clearly, but the code is sound.

Paged results are implemented to allow growth of the site in the future. this is good.

Signup is good with thorough validation. Login is also well managed.

Time is used. Edit is complex due to the number of fields, Delete also used.

Although the site looks simple on the surface, there is a lot going on here with joins etc due to the large number of database tables being managed.

43/50

Overall integration of website - A bit sparse - I know you prefer development to design, but perhaps a

few pictures in the header and some consideration of the colour scheme :) 7/10

REPORT

Investigation/analysis of two security issues -
SQL Injection is covered well with suitable examples.

Spam bots are OK - you might want to give some screenshots/code examples of where the problems lie to illustrate your text. 13 /15

Mitigation of security issues in your site - Shown well generally, but a bit limited - there are other areas in your site where prepared statements are(or could be) implemented. A few more examples would be helpful, but good on the whole. 10/15

General Comments: Even though this is only a short report, all academic reports should have an Introduction and a Conclusion - even if a short one. Overall, I can see that you've put a lot of effort into this assignment - well done.

TOTAL: 83/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4
