Manual Testing of SQL Injection Vulnerabilities in an Online Student Database System Using Same Channel Strategy

Mu’awuya Dalhatu1, Aliyu Sani Ahmad2 and Moshood Abiola Hambali3
Department of Computer Science, Federal University Wukari, Nigeria.
Email: 1dalhatu@fuwukari.edu.ng, 2alally_ahmad@yahoo.com, 3hambali@fuwukari.edu.ng
*Corresponding Author

ABSTRACT
In the recent time, the dependency of universities, businesses and organisations on online databases are increasing in both the developed and developing countries like UK and Nigeria. So also the security challenges and threats are increasing. The most common and dangerous online database threat is SQL injection attacks and followed by XSS. SQL injection attack is a challenge to any online database that process data based on user inputs. This research investigates one of the most important online database threats. The research reviews related work, design and implement a secured student database application for testing the SQL injection. The research findings show that there is no single technique, policy or procedure that is powerful enough to prevent all SQL injection attacks. However, the research recommends the proper implementation of techniques such as sanitising inputs and using bind variables.

Keywords: Online database, SQL injection, Threat, Mitigating SQL Injection, XSS

1. INTRODUCTION
Despite the acceptance, role, and importance of computer applications especially in universities, many universities, businesses and organisations especially in developing countries like Nigeria are still using manual systems to manage records. This may be due to the threats associated with the online databases or applications. Most small or newly established universities and businesses may seem to use the manual system effectively. However, many organisations such as the universities grow rapidly each year; therefore, they require automated systems such as a database system in order to reliably manage records by eliminating redundancy and improving accessibility, security, and privacy.
Database systems could be considered online or offline. Access to an online database system is usually granted over a network via an application interface. This database system is more challenging to protect because both authorised and unauthorised users have access to the application interface that serves as a door to the database.

Web applications and online databases in particular, are facing one or more threats. Many universities, companies and organisations in both developed and developing countries like United Kingdom and Nigeria have in one way or the other happened to be a victim of such attacks [1].

There are different threats to both the web applications and online databases that include Structure Query Language (SQL) injection, Cross Site Scripting (XSS), cookie poisoning, cookie stealing, cross site request, JavaScript hijacking, and Domain Name System (DNS) rebinding [2]. SQL injection and XSS are directly related to online databases which could be accessed through an interface or a web application interface in particular.

According to OWASP [3], SQL injection and XSS are the most common threats to online database systems. The report lists SQL injection as the most common threat, followed by password cracking while XSS is the third most common, globally. This may be applicable to developed countries like the United Kingdom and the United States of America, where most of the research on database security came from. However, there is an urgent need for researchers to focus on developing countries like Nigeria where the online databases have even more threats because they have only been in use for the last two decades. For example, National Information Technology Agency (NITDA) [4] is the first agency established by Federal Government of Nigeria in the year 2001 that is responsible for researching, monitoring, standardization and evaluating of information technology. Similarly, Sowala and Abubakar [5] stated that policies, regulations and techniques for mitigating and preventing threats to online databases and applications are newly established in Nigeria. They further stated that attacks on online databases are threats to national security, the economy and the integrity of the country as a whole.

Furthermore, Awoleye, Ojuloge and Siyambula [6] conducted a research on the security of 64 websites maintained by the Nigerian Government. Their research states that 42.2% of the websites are not safe from XSS threats, 31.3% are vulnerable to SQL injection threats. Their findings also state that 37.5% are vulnerable to password cracking. This study conforms to the 2013 OWASP [3] report discussed above.

Several research such as [6], [8], [5] and [9] consider non-technical solutions or policies such as the Nigeria Cybercrime Act 2013, Establishment of online database security standard, Enactment of the Nigeria’s Cybercrime Act 2015, the establishment of Nigeria National Computer Emergency Response Team (ngCERT), and public awareness as the strategy for minimising the online crimes.

However, several studies such as [15], [1], [7] and [10] have suggested different technical solutions and recommendations for dealing with SQL injection and/or XSS. Some of these recommendations are running a server with minimal privileges and implementation of input validation technique. The use of a four-tier approach that requires implementation of different security policies or techniques at system layer, web server layer, database layer and application layer as a means of protecting the entire system from different type of attacks. Others are parameterized queries, smart configuration of database management system, and stored procedures.

These recommendations show that the researchers believe that government policies, rules and regulations can play a major role in minimizing attacks on online database systems in Nigeria. This will help in some cases. However, non-technical policies or legal implications will not prevent attacks because of the challenges associated with tracing the origin of the attacks and identifying who the attackers are. The increase in cybercrimes and applications hacking especially in Nigeria may be associated with lack of strong technical solutions that can prevent attacks. When both the technical and non-technical solutions are used together, the technical solutions will be for preventing attacks while the non-technical ones will be for punishment of convicted attackers.

Therefore, there is a need to investigate and suggest a simple and reliable technique for addressing SQL injection attacks for developing countries like Nigeria. This is necessary because most of the online database and web application developers in Nigeria are non-professional and hardly consider the issues of security. The difficulty in considering or implementing minimal security requirements may be due to a lack of strong security awareness, the complexity and cost of existing solutions, rushing to meet deadlines or omission in the customer’s requirements.

This research focuses on SQL injection attacks because these are the most common and dangerous threats as identified in the related work section. The research also considers the design and implementation of a secured online database system for managing students’ activities.
such as registration, checking result and storing staff and courses record (see appendix).

II. LITERATURE REVIEW

Knowing how to properly detect, prevent or trace the sources of attacks on online databases requires one to have an in-depth understanding of most of the possible vulnerabilities, threats, and the different types of attacks and techniques used by the attackers to achieve their aims or goals. Understanding the intents of attackers is essential in making a research on online database security [11]. Halfond, Viegas, and Orso [12] identified nine possible attacks intent which include Identifying injectable parameters; Deleting, updating or inserting of data; identifying a database type and version; Escalating privilege and Extracting information.

Extracting information is one of the most common attack intents that requires an attacker to determine database schema, bypass detection and authentications [28]. The attack also requires the use of different techniques such as SQL injection for data extraction. Shar and Tan [16] define a SQL injection as a code injection approach that is used to attack online databases with weak or incorrect input validation techniques. Attackers append the malicious code to the pre-defined SQL statements that are sent to the database server for processing. The malicious codes can be appended to Universal Resource Locator (URL) and any input field that can be used to return output from the databases [14]. Sadeghian, Zamani, and Ibrahim [7] agreed with Shar and Tan [16] that SQL injection vulnerabilities exist in any online database that processes data based on users input and it can easily be successful where the inputs are not well sanitized and filtered. Sadeghian, Zamani, and Ibrahim [7] further categorised a SQL injection into three: in-band, out-of-band and inferential.

The in-band category may be considered simple because the attacker receives output from the page where the attack is launched while the inferential category is considered more challenging because it requires more time and does not return output information to the attackers directly [17]. In inferential SQL injection, attackers explore the vulnerabilities by using logical operations that will be evaluated as true or false. The attacker monitors the behaviour of the system based on logical operations.

2.1 SQL Injection

There are different SQL injection techniques and strategies that are being used by attackers to compromise the security of online databases. Sadeghian, Zamani, and Ibrahim [7] identified encoding technique, whitespacing technique, comment technique, case and variation techniques that are effective in attacking online database systems. Basta and Zgola [18] agree with the stated techniques and further identified conditional, union and concatenation techniques. Other techniques or attacks include tautology, piggy-backend query, illegal/incorrect logical query [12], [13], [14] and time delay that helps an attacker to monitor the successfulness or otherwise of executed malicious codes based on the time taking by a server to response [15]. Some of these techniques are briefly explained below:

2.2 Union

UNION operator is used to return data from two or more tables. It is a very effective operator in attacking online database systems because it allows an attacker to extract data from the unintended table by attaching their malicious queries to the legitimate and predefined query. According to Randhe and Mogal [19], UNION operator can be used to bypass authentication and retrieve data from other tables.

Assuming a system requested a student to enter their student ID in order to view their result. The following query will be executed if a student enters 12345 as their student ID.

SELECT * FROM result WHERE student_id= 12345;

However, the query looks like the following if a UNION operator is attached:

12345 UNION SELECT NULL, NULL FROM information_schema;

Moreover, UNION operator required all the query statements to select the same number of columns with the same datatype in the same order. However, NULL value is used to bypass datatype checking because it has no datatype and can be used for unknown values. NULL can also be used to determine the number of columns.

2.3 Tautologies

According to Gollmann [2] and Nadeem et al., [26], tautology is one of the simplest SQL injection techniques for injecting malicious codes. Tautology is always true and conditional. Examples of tautologies are 1=1, 1<2, 1 like 1 or 10/2=5. Randhe and Mogal [19] state that tautology is mainly used in bypassing authentication and retrieving or updating information.
For example, instead of providing correct student_id (say 12345), an attacker decides to input ‘123 or 2=2 - - ‘ which will be evaluated as follow:

```
SELECT * FROM result WHERE student_id = 123 or 2=2--
```

The second part of the ‘where’ clause will always be evaluated as true which will lead to returning of all the results in the table. The ‘--’ symbol is a comment that causes the remaining code execution such as verifying password to be ignored.

### 2.4 Encoding

Encoding is a process of changing characters from one standard to another in order to bypass detection. Sadeghian, Zamani, and Ibrahim [7] state that URL encoding, Unicode/UTF-8, hexadecimal encoding is commonly used encoding standard. Others include ASCII and `char()` function that takes an integer to return a string [25]. The encoding technique can be used where SQL keywords or whitespaces are not allowed.

For example, the following are the hexadecimal equivalent encoded for URL:

```
“OR I LIKE I” is equivalent to %6f%72%20%31%20%6c%69%6b%20%31%20%6f%72%20%32%3d%32%2d%2d%20%65%6e%74%5f%69%64%20%31%32%33 %54%72%61%20%73%74%75%64 %65%6e%74%5f%69%64%20%3d%32%3d%32%2d%2d%20
```

```
“UNION SELECT * FROM result WHERE student_id = 123 or 2=2--"is equivalent to:
percent65%6e%69%6b%20%31%20%6c%69%6b%20%31%20%6f%72%20%32%3d%32%2d%2d%20%65%6e%74%5f%69%64%20%31%32%33 %54%72%61%20%73%74%75%64 %65%6e%74%5f%69%64%20%3d%32%3d%32%2d%2d%20
```

### 2.5 Time Delay

Time delay technique employs by an attacker where no useful error message is returned to them. The ‘wait for delay’ SQL command in MySQL and ‘pg sleep ()’ function in Oracle [18] allow an attacker to predict the success or otherwise of their malicious codes [15]. For example, information about a password may be obtained if a `username` is known:

```
if (username="abc" and (Len (password) < 10)){wait
for delay 00:00:30;} or if (username="abc" and (IsNumeric (password))){wait
for delay 00:00:30;}
```

### 2.6 Concatenation and Comment

Both the SQL single and multiple line comments can be used to get malicious code executed successfully. The commenting characters can also be concatenated for execution. For example, if an attacker input ‘or’ 2 ‘=’ 2 - - in a given `username` field of a vulnerable system; the system will probably give access to the attacker and grant them the privilege of the first user in the authentication table. For example, the following statement may bypass detection.

```
SELECT * FROM result WHERE username = 'something' OR '2' = '2' -- AND password = anything;
```

```
OR SELECT * FROM result WHERE /**/ OR /**/ WHERE /**/ username = 'something' or '2' = '2' -- and pass = 'anything';
```

### 2.7 Mitigating SQL Injection

To properly mitigate or prevent SQL injection attacks, one has to understand that different attacks may require different prevention techniques. As Sadeghian, Zamani, and Ibrahim [7] stated, there is no single protection technique that is strong enough to stop all SQL injection attacks. Similarly, Basta and Zgola [18] argue that there is no exact way of preventing them. However, they further argue that an approach that combines several strategies is more effective. This suggests that combining different strategies add more complexity and protection. Furthermore, Shar and Tan [20] broadly identify three categories of SQL injection defence strategy: defensive coding, vulnerability detection, and runtime prevention. Some of the mitigation and prevention techniques are briefly explained below.

#### 2.7.1 Secure Programming

Deepa and Thilagan [21] recommended the use of secure programming because it enables developers to implement secure coding during the applications’ development. Similarly, Sharma and Jain [22] highlighted the importance of secure programming in preventing blind SQL injection attacks that are difficult to prevent with other techniques.

Moreover, Sharma and Jain [22] identified two common principles of secure programming. The first principle suggested the use of a generic or customised error message. A Customised message hides the details of errors generated by both legitimate and malicious queries. This can be achieved by handling the errors internally or providing generic error messages, such as ‘invalid username/password’ instead of stating why access is denied. The second principle is filtering user inputs. However, according to Gupta et al., [23], secure
programming alone does not guarantee full protection against SQL injection attacks. This may be due to the flexibility of the SQL language that leads to different types of SQL injection attacks.

2.7.2 Parameterized Queries

Parameterized query is also known as a prepared statement. This avoids concatenating user inputs directly in the SQL statement by using a concept of a placeholder [24]. Different studies such as [22], [10], and [7] have recommended the use of parameterized queries in preventing SQL injection attacks. Furthermore, a parameterized query is capable of preventing different types of SQL injection attacks if dynamic queries are properly replaced by prepared queries. Moreover, Qian et al., [10] identified two advantages of parameterized queries. The first advantage is that user inputs are treated as a literal value not as a code to be executed. The second is that client servers ensure that only correct data types and length are acceptable. However, the major drawback with parameterised query is that it can only be used for data values and not all user inputs are data values [18].

2.7.3 Validating Data

Despite the flexibility of SQL injection techniques to avoid filtering, there is still the need to have a technique to filter and validate user inputs in order to increase the complexity of the security for repelling attacks as early as possible.

According to Shar and Tan [20], identifying a datatype mismatch can be used to detect malicious queries or data by checking if the input is numeric or string. Shar and Tan [20] further identified ‘white list’ and ‘black list’ as the two filtering techniques. The ‘white list’ is used by developers to accept only known and expected characters while the ‘black list’ is used to reject malicious characters such as ‘or’ or a ‘=’ sign. However, Basta and Zgola [18] and Morgan [15] argue that the ‘white list’ approach is more effective and saves time because a developer can easily list the good characters they expect, in contrast to listing all the possible combination of malicious characters as required in the black list approach.

Furthermore, Qian et al., [10] and Nayak and Sharma [28] list some suggestions for mitigating SQL injection attacks that include: no assumptions should be made about user inputs; content and length of string inputs should be verified; appropriate limitations should be enforced on user inputs.

2.7.4 Escaping

Escaping SQL keywords and malicious characters is another means of mitigating SQL injection attacks, programming languages such as PHP provide mysql_real_escape_string() function for escaping potential malicious keywords and characters [24], [27]. Furthermore, Sharma and Jain [22] recommended the escaping of comments characters, query delimiter, character data string delimiter, and SQL keywords in user inputs.

However, Shar and Tan [20] argue that escaping has some drawbacks due to common mistakes made by developers and the challenges associated with identifying what to escape. They however recommended the use of built-in escaping functions over customized ones. Similarly, Golmann [2] suggested the use of PHP built-in functions such as is_numeric (), setType () and addslashes () functions for numeric data. This is important because escaping numeric data may not protect the system properly.

2.7.5 Stored Procedures

Shar and Tan [20] and Sharma and Jain [22] recommended the use of stored procedures to replace dynamically generated queries so that online databases cannot be accessed directly. Furthermore, Qian et al. [10] recommended the use of stored procedures for validating user inputs. According to Shar and Tan [20], stored procedures make injecting malicious code highly challenging to attackers. However, Mukhejee et al. [24] state that attackers can make use of vulnerable stored procedures to compromise the system and escalate their privileges. Moreover, Morgan [15] state that there are several stored procedures for displaying local drives, security mode of a server, or killing a process that are vulnerable on online database systems.

III. TESTING PLAN

This section discusses the method or strategy that will be used to test the security of the online database before a solution is proffered based on the detected vulnerabilities.

According to Basta, Zgola and Bullaboy [18], SQL injection can be tested using observational, multichannel or same channel strategy. The same channel is chosen for this research work. In this case, the same page will be used for communication between the server and client. The steps are briefly outlined below.

1. The first step is to identify an entry point into the database in order to bypass the login screen
or extract information such as results. The entry point could be any input field or URL that returns output.

2. The second step is to determine the expected datatype of the identified entry point. This is required because numeric and non-numeric input have different criteria and are processed differently. For example, a non-numeric input requires a single quote while a numeric does not.

3. The third step is to employ a particular type of attack for the initial testing. The testing will start with the error based then union and follow by blind SQL injection. The first step in the error based is to input data such as a single quote that may generate a system error message.

4. The fourth step is to review the code manually to fix errors or vulnerabilities and implement appropriate mitigation techniques such as bind variable, validating inputs and returning a generic error message.

Table 1 summarises the test plan and their respective outcome.

### 3.1 Implementation

An online student system is developed in order to test and identify SQL injection vulnerabilities, investigate the effectiveness of existing solutions and check how strong the recommended or implemented solutions are.

The software consists of the back-end and front-end. The back-end is implemented using Oracle database system (Oracle Database 12c) while the front-end is implemented using HTML5, Bootstrap framework, JavaScript and CSS (Cascading Style Sheets). The Connection between the front-end and back-end is made using PHP programming language. Secure programming practice, bind variable, validating user inputs and customised error messages are used in mitigating SQL injection attacks. The online application is with the researchers.

However, some of the restrictions on Oracle will, however, affect the security testing plan because Oracle database does not allow execution of two SQL queries in a single statement. For example, the following statement is not executable on Oracle, but it is executable on MySQL. Such restriction makes Oracle less vulnerable to SQL injection.

```
SELECT * FROM result; UPDATE result set grade='A'
WHERE student_id=123 and grade='F';
```

### 3.2 Testing

This section discusses the basic SQL injection test that was carried out manually. The testing starts by identifying an injectable point followed by injectable datatype; then reviewed the code and implemented appropriate defence techniques.

The first step starts by inserting a single quote in the username field (Fig. 1), the system generate an error message as a result of inputting a single quote as shown in Fig. 2. The error message generated in Fig 2 reveals at least three important information: the first information is the system is vulnerable to SQL injection attacks. The Second information is the back-end uses Oracle database system (due to ORA- 01756 :) and the third information is the scripting language is PHP (due to oci_parse function). Similarly, Fig. 4 shows an error message generated by an empty input (see Fig. 3) can also help an attacker to compromise the system security. This test shows how powerful a single quote character is and how important error messages are to an attacker or tester.

The first step in securing the system from SQL injection attacks is to handle error messages internally and make sure that an empty input is not passed to the database. This is achieved with the code fragment shown in Fig. 5. The system returns "Enter username/password" for both single quote and an empty input as shown in Fig. 6 and 7 respectively. It also returns "invalid username/password" for invalid input as shown in Fig. 8.

The system now handles single quote and empty username/password input the same way.
Mu'awuya Dalhatu, Aliyu Sani Ahmad and Moshood Abiola Hambali (2019), Manual Testing of SQL Injection Vulnerabilities in an Online Student Database System

Fig. 1. Student Login with Single Quote

Fig. 2. System Error Message Generated by a Single Quote.

Fig. 3. Student Login with Empty Input

Fig. 4. System Error Message Generated by an Empty Input

Fig. 5. Code Fragment that Return Error Message for Single Quote and Empty Input
Table 1. Test plan and expected outcome. (NOTE: Where 12345 represents student ID, and File0.php represents a file name on the server)

<table>
<thead>
<tr>
<th>s/no</th>
<th>Type of test</th>
<th>Injection Point</th>
<th>Input</th>
<th>Expected Output</th>
<th>Vulnerable</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Error based</td>
<td>Login screen</td>
<td>‘ ‘</td>
<td>Enter username/password</td>
<td>No</td>
<td>It is not vulnerable due to customised error messages.</td>
</tr>
<tr>
<td>2</td>
<td>Tautology</td>
<td>Login screen and checking result</td>
<td>12345 or 1=1--</td>
<td>Invalid username/password</td>
<td>No</td>
<td>It is not vulnerable due to input validation.</td>
</tr>
<tr>
<td>3</td>
<td>UNION</td>
<td>Extracting result</td>
<td>12345 UNION SELECT student_id, module_code, grade FROM result</td>
<td>No result is returned</td>
<td>No</td>
<td>It is not vulnerable due to input validation and bind variable</td>
</tr>
<tr>
<td>4</td>
<td>Blind</td>
<td>Login screen</td>
<td>12345 and 1=1 &lt;1</td>
<td>Invalid username/password</td>
<td>No</td>
<td>It is not vulnerable because the same error message is returned in both cases.</td>
</tr>
<tr>
<td>5</td>
<td>Blind</td>
<td>URL</td>
<td>File0.php%20and% 201=1 File0.php%20and% 201&lt;1</td>
<td>file ‘not found on this server’</td>
<td>No</td>
<td>It is not vulnerable because the same error message is returned in both cases.</td>
</tr>
<tr>
<td>6</td>
<td>Constraints violation</td>
<td>Inserting and deleting</td>
<td>unique constraints violation, integrity constraints violation,</td>
<td>System error message</td>
<td>Yes</td>
<td>It is vulnerable because system error messages are return in both cases</td>
</tr>
</tbody>
</table>
The second step is to determine the injectable datatype. This is easy to be identified in a student system like this, because most student systems use student ID as a username and an alphanumeric as a password. However, it is important to test that practically. Figure 9 shows the error message generated for a string (abc123) input which indicates that the username should be numeric.

The next step is to use a tautology technique to bypass authentication. A tautology technique is then used to compromise the system’s login as shown in Fig. 10. Such vulnerability can be eliminated by validating user inputs and returning “invalid username/password” for invalid datatype as shown by the code fragment in Fig. 11. Validating the user input is implemented with

```php
if (!is_numeric($username)) {
    return "invalid username/password"; // hiding error generated by invalid data
}
```

Fig. 10. Student Login with Tautology Input

‘isNumeric()’ PHP function. This makes the system secured from both the tautology attack and an invalid datatype as shown in Fig. 12 and Fig. 13

Fig. 11. Code Fragment that Return Error Message for Invalid String and Tautology Input
3.3 BIND VARIABLE

Bind variable is another reliable and easy technique for preventing different SQL injection attacks. It uses the concept of a place holder similar to parameterised query discussed in section 5.2. It is implemented in this research in order to increase the level of security as suggested by [18].

SQL statement is first prepared without the user inputs and later treated the inputs as laterals as shown in Fig. 14.

A bind variable is particularly important in preventing the UNION and tautology attacks.

IV. DISCUSSION AND CONCLUSION

This research identifies several threats to online databases and considers SQL injection as the key threat, followed by XSS. The research also shows that existing solutions against SQL injection attacks are effective. However, one has to understand what techniques or policies to use, where to use and how to use it properly. Moreover, the security testing shows how significant generic error messages are, and how effective the implemented solutions such as sanitizing user inputs and bind variable are against different SQL injection attacks such as tautology, union, blind and error based.

The research findings show that there is no single technique, policy or procedure that is powerful enough to prevent all SQL injection attacks. However, the research recommends the proper implementation of techniques such as sanitising inputs and using bind variables at least for students’ and businesses’ websites.
V. RECOMMENDATION AND FUTURE WORK

The research work focuses on first order injection attacks. The research recommends the proper validation and sanitisation of inputs, secure programming, stored procedure and bind variable in dealing with SQL injections especially the first order one. However, according to Sharma and Jain [22], second order and literal injection attacks exist. Therefore, there is a need to investigate such attacks also to secure the system from them.

Adequate security provided by one aspect of a system such as the network, database or web application will not be sufficient for the entire system while security failure in one aspect may compromise the entire system. Therefore, there is need to test online applications with automated tools because most attacks are carried out automatically. There is also a need to test and protect the applications against web applications threats such as stealing cookie and session. Moreover, implementing other security measures such as password hashing, privilege and auditing will be of security benefits especially when a database is compromised.

VI. REFERENCES


BIODATA

Mu’awuya Dalhatu is a lecturer at the Department of Computer Science, Federal University Wukari, Nigeria. He obtained a B.Sc. Computer Science from Bayero University Kano, in 2011 and M.Sc. Computer Science from the University of Wolverhampton, United Kingdom in 2016. His research interests are Database Security, Web Designs, Distributed and Embedded Systems.

Aliyu Sani Ahmad obtained his bachelor degree in ICT from Federal University of Technology Yola in 2009 and Master of Science in Computer Science at University of Greenwich (UK). He is presently working as lecturer at the Department of Computer Science, Federal University Wukari, Nigeria. Previously, he worked with Centre for Continuing Education Consultancy Services, College of Agric. as Resource Person between years 2011 to 2012. During his Master program, he worked with Skills Hive as Web Application Developer, UK. His current research interests are Artificial Intelligence.

Moshood Abiola Hambali, is a lecturer in the Department of Computer Science, Faculty of Pure and Applied Sciences, Federal University Wukari, Wukari-Taraba State, Nigeria. He graduated with a B. Sc. (Hons) in Computer Science from Al-Hikmah University, Ilorin; M. Sc. in Computer Science Department from the University of Ilorin and presently, PhD Scholar at University of Ilorin, Nigeria. He had published over 20 articles in both local and international journal. His research interest includes Data mining, Bio-Informatics and Information Security.
APPENDIX

Home Page

Admin Login Form

Administrative options

Student's Result Displayed
Mu'awuya Dalhatu, Aliyu Sani Ahmad and Moshood Abiola Hambali (2019), Manual Testing of SQL Injection Vulnerabilities in an Online Student Database System

© 2019 Afr. J. Comp. & ICT – All Rights Reserved
https://afrjcict.net

Student’s Registration for Academic Data

Module Registration form for Admin

Segment of a Vulnerable Code for Retrieving Student’s Result

Segment of a Code for Retrieving Student’s Result Using Bind Variable