iBibliography SQL Injection Solutions

Alexander Tse, Reuben Heredia, Mohammed Taher, Jordan Chiou,
Electrical and Computer Engineering, University of British Columbia

***Abstract*—Web applications in today's cyber-world can be victims of SQL injection attacks. These attacks can allow the hacker access to confidential information stored on the databases of the web applications. In order to test for vulnerabilities, this paper analyzes the security of Dr. Konstantin Beznosov's web application, iBibliography, and proposes two possible solutions to secure the application. However, these solutions may not necessarily the best solutions and this paper also examines other possible solutions that may protect the web application more securely.**

***Index Terms* – SQL Injection, iBibliography, Acunetix Vulnerability scanner**

I. Introduction

Structured Query Language (SQL) is a standard programming language for database driven web applications. SQL is used to retrieve, modify and manage data in Relational Database Management Systems (RDBMS) [1]. SQL based databases are also vulnerable to SQL injection based attacks, are easy to find and are exploited by attackers [2], [3]. There are many ways to reduce the vulnerability of web applications by adopting countermeasures against SQL injection.

This paper presents the solutions designed to prevent SQL injection attacks by taking Dr. Konstantin Beznosov's online bibliography database, iBibliography (iBib), as a test case. iBib is web based application based on an open source software called WebBiblio. It provides access to various papers, journals and other publications authored by Dr. Konstantin Beznosov and his colleagues, and other team members of Laboratory for Education and Research in Secure Systems Engineering (LERSSE).

Manuscript received December 1, 2008
Tse (email: tsetsealexander@gmail.com)
R.Heredia (email:rohbit@gmail.com)
M.Taher (email:mstaher@gmail.com)
J.Chiou (email: jordanchiou@hotmail.com

A visitor to the website is able to search for publications based on keywords, author, subject etc. iBib is composed of SQL based database called MySQL (a RDBMS owned by Swedish Company MySQL AB, a subsidiary of Sun Microsystems [4]) in the backend and PHP in the front end.

II. SQL Injection

SQL injection is a technique that exploits the security of a web application that uses SQL based databases [1]. An attacker passes malicious SQL code into strings which are passed to the SQL server for execution. The most common method of SQL injection is to directly insert malicious code into user input variables (e.g. login pages, search pages) [5]. This type of attacks allows the attackers to [6]:

1. Spoof Identity
2. Delete, add or modify data
3. Allow complete or partial disclosure of data and subsequently procure sensitive information
4. Execute OS commands

III. iBibliography Search Function and its Vulnerabilities

iBib provides the user with two channels to interact with the database. In addition to searching the database, registered users are allowed to add, remove or modify publications after they have logged into the system via the login page. Non-registered users and visitors to the website can only use the search page to query for publications based on keywords, author, subject etc. Since the search page allows any user to access the input variables, we have analyzed the search page for SQL injection vulnerabilities.

A user can search for publications using either the basic search on the home page or can choose to do an advanced search. When a user enters a keyword in any of the search pages and submits the form, the information is taken to the *biblio_search2.php* page, which does all the processing and communication with the actual database. Once the database returns the search results to *biblio_search2.php*, the returned information is displayed to the user. The structure of the search page is shown in Figure 1.

Figure 1 - Structure of the search pages in iBibliography

A simple search for a single quotation mark ' produces a simple error page saying "Error counting bibliography search results". This is an SQL Injection attack but does not give any confidential information away. The source of the webpage gives us more detail about the error messages giving us more information about the database such as data table and field names as shown in screenshot in figure 2 below.

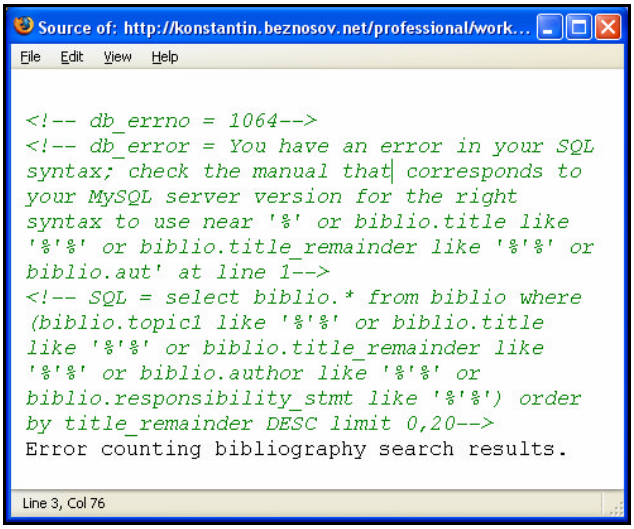

Figure 2 – The source code of the error message

We used a popular Web application vulnerability scanner Acunetix to search for vulnerabilities with iBib. This scanned served as our benchmarking tool for checking the effectiveness of our solutions. Our initial scan of iBib using Acunetix showed 199 SQL injection and 42 blind SQL injection vulnerabilities. From Acunetix we were able to identify three variables that were prone to SQL injection errors and had highest manipulation count. The variables are:

- *searchText* – the variable used when user uses the search page
- *sortBy* – the variable used for *biblio_search2.php* to sort results returned by the database
- *page* – the variable used for number of pages of search results

Other variables that were identified were on

- login.php
- bibtex.php
- view.php
- biblio_view.php

IV. Our Design Solution

As mentioned in previous section, the search page had the three most affected variables: *searchText*, *sortBy*, and *page*. Since they could contain malicious SQL code, we implemented filters in the two pages which pass the

variables: *search_form.php* and *biblio_search2.php*. By doing so, we achieve some level of defense in depth, as a malicious user has two barriers to overcome.

A. *Function isAlphaNum in search_form.php*

The function implemented in *search_form.php* is a JavaScript function called isAlphaNum(field). The function checks the search-input field for meta-characters and displays an alert every time the user enters a meta-character, i.e. (‘ “ -- . / etc.). Basically, a meta-character is any punctuation character. This function is a mechanism to keep the hacker frustrated every time he or she tries to enter punctuation needed to inject the malicious query.

```
function isAlphaNum(field)
{
        var re = /^[a-zA-Z 0-9]*$/;
   if (!re.test(field.value))
   {
        alert("Please enter only
        alphanumeric letters.");

        return true;
   }
   else
   {
        return false;
   }
}
```

By far and away the best approach is to identify which characters you wish to permit, so the *isAlphaNum(field)* function first defines the list of allowed characters in the variable *re* and then compares the list with the value of the *field* [7]. If this is the case, it allows the user to continue entering their search; however, if the user enters any meta-characters, it will alert the user as seen in Figure 3. The way it is implemented in the form is that it will check the field every time a key is released using the *onkeyup* field; and, it will check before submitting the form since *isAlphaNum(field)* is implemented in both the Enter and Submit Form Javascript functions. The following HTML code excerpt displays how the function is implemented into the form.

```
<input type="text" name="searchText"
size="30" maxlength="256"
onkeyup='isAlphaNum(this)'
onkeypress='return
entsub1(event,this.form)'>
```

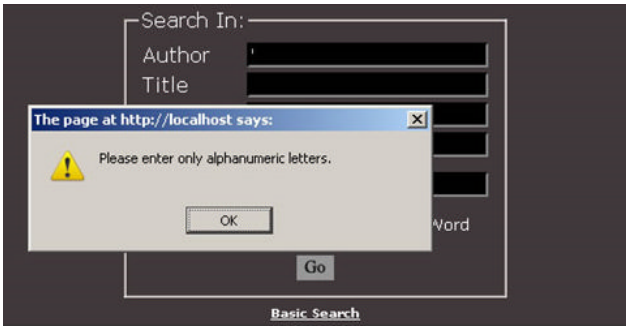

Figure 3 – Users are alerted when typing meta-character

B. *Function eregi_replace in biblio_search2.php*

As shown in Section III, the majority of the vulnerabilities were found in the page, *biblio_search2.php*. The solution we implemented in this page and various other pages in iBibliography utilized the Portable Operating System Interface (POSIX) Regular Expression (Regex) function:

```
eregi_replace ( string $pattern ,
string $replacement , string $string
)
```

This function scans *string* for matches to *pattern*, and replaces the matched text with *replacement* [9].

The function mainly searches for [:punct:] – i.e. meta-characters – and removes them by replacing them with a NULL character. If a meta-character separates two letters or numbers, the removal will append the two like so:

INITIAL STRING:	**hello’ or 1=1;--**
MODIFIED STRING:	**hello or 11**

Since a user can still bypass the *search_form.php* and submit their (possibly) malicious search string, the aforementioned code serves the primary defense against a persistent user.

In *biblio_search2.php*, this function was used as follows:

```
/****************SEARCHTYPE*****/
if( eregi("[a-zA-Z]+",
$_POST["searchType"]))
     {
          $searchType =
          eregi_replace("[[:digit:][:
          punct:]]+", "",
          $_POST["searchType"]);
```

```
        }
        else
        {
        return false;
        }
```

In the previous code, the search field *searchType* is being searched for characters that are either digits or punctuation and replacing them with a NULL character. However, the next code excerpt cannot use [:punct:]:

```
 /***********SORTBY*************/
if( eregi("[a-zA-Z_]+",
$_POST["sortBy"]))
        {
              $sortBy =
              eregi_replace("[^a-zA-
              Z_]+", "",
              $_POST["sortBy"]);
        }
        else
        {
        return false;
        }
```

This code cannot use the Regex pattern [:punct:] to search for punctuation because the variables in *sortBy* have an underscore (_) such as *publication_year*. In this case, we must search for any characters that are NOT alphabetic and is not the underscore character. As such, by inserting ^ before the search pattern "a-zA-Z_", eregi_replace() will ignore those characters and search for all other characters.

V. Results

After implementing all our solutions for *biblio_search2.php* and all the other pages with SQL injection problems, we rescanned our website using Acunetix and found no SQL vulnerabilities associated with the search pages. The screenshot of the scan is shown in figure 4.

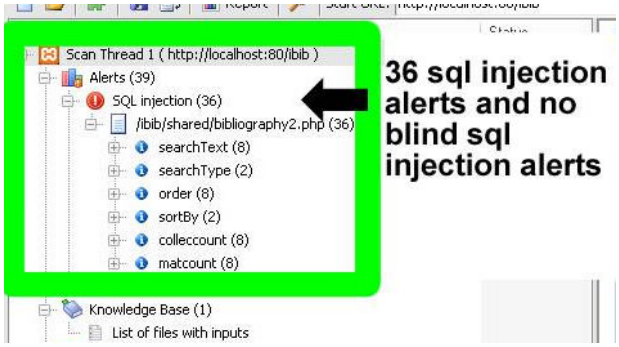

Figure 4 - Acunetix scan result after our solutions were implemented

The screenshot shows only 36 SQL injection vulnerabilities associated with the page *bibliography2.php*. The function of this particular page is to display the search results in a popular bibliography format. As of this moment, we are ignoring these vulnerabilities because this page does not do its function and as such, students in the course EECE 496 will modify this page in future terms. The students may have to add the security that we've implemented to the variables in *bibliography2.php*.

VI. Future Security Considerations

Future considerations for iBib security would be to introduce the use of parameterized statements or procedure calls. A parameterized statement prepares a SQL statement to bind variables to represent user input.

The advantage of this is that user input is not embedded in the SQL statement anymore; thus an attacker could not easily insert unintended queries at the end of the statement.

The other option is to use stored procedures. A stored procedure can be said as a function/ method that does specific tasks to the database developed in Transact-SQL language. It also uses binding variables to represent user input, but on top of that, the SQL statement is no longer visible in the code and instead is wrapped inside the stored procedure. For example:

```
$stmt $db-> prepare("SELECT * FROM
Table")
```

would become:

```
$stmt $db-> prepare("call method()")
```

Furthermore, security permissions are introduced in stored procedures by granting different access permission to various users. However, stored procedures must be coded carefully because SQL injection can be possible in stored procedures. [9]

VII. Comments

We initially investigated the use of procedure calls in iBib but found several problems to it. We were unable to create a procedure call property in phpMyAdmin, which is used to administrate iBib MySQL database. A simple query such as "`SELECT * FROM BIBLIO`" would return an empty result set which should not happen. Instead, we downloaded another tool called *MySQL Administrator* to manage the database and create the procedure call from there. We were able to create stored procedure and execute it properly through the manager. This procedure would also show up under phpMyAdmin; however, when we execute it in phpMyAdmin, it still retumed empty result set. We also tried running the stored procedure in iBib, but the result is still the same. We further investigated this issue and noticed that the MySQL API used to connect to the database may not support procedure calls. Therefore, in the future, we would suggest updating the code to use the MySQL API introduced in the Python 5.x version.

VIII. Conclusion

What we have demonstrated here is the proof of concept that SQL injection is indeed fixable. First, we used Acunetix to scan iBib to find all SQL vulnerabilities and found that there were 249 SQL injection related problems. Our solution is to implement code in both the front-end and back-end of iBibliography. In the front end on *search_form.php*, the form's field(s) would be checked every time the user types a letter and immediately before the user submits the form. If the field contains any meta-characters (or punctuation), a pop-up will alert the user to only input alphanumeric characters. If the user manages to submit the form with meta-characters, the code in the back-end on *biblio_search2.php* would replace all the meta-characters with null characters. After implementing these fixes, we scanned the web application again and all the SQL injection related problems have been solved. Therefore, we conclude that introducing character filters in the search form is a viable solution.

References

[1] "SQL Injections". Wikipedia, 2008. [Online] Available: http://en.wikipedia.org/wiki/SQL_injection

[2] W. G. Halfond, J. Viegas, and A. *Orso*, "A *Classification of SQL-Injection* Attacks and Countermeasures," 2006

[3] S. Sun, T.H.Wei, S.Liu and S.Lau, "Classification of SQL Injection Attacks,"2007. [Online]. Available: http://courses.ece.ubc.ca/412/term_project/reports/2007-fall/Classification_of_SQL_Injection_Attacks.pdf

[4] "MySQL". Wikipedia, 2008. [Online] Available: http://en.wikipedia.org/wiki/MySQL

[5] "SQL Injection". Microsoft Development Center. SQL Server Development Center. 25 November 2008. http://msdn.microsoft.com/en-us/library/ms161953.aspx

[6] "SQL Injections". Open Web Application Security Project, 2008 [Online]. Available: http://www.owasp.org/index.php/SQL_injection

[7] "Preventing SQL injection attacks in stored procedures," Software Engineering Conference, 18 - 22 April 2006 [Online]. Available: http://www.owasp.org/index.php/SQL_injection

[8] "Eregi Replace" Php.net, 2008 [Online]. Available: http://ca3.php.net/eregi_replace

[9] "Secure Programming for Linux and Unix HOWTO," DocMirror.net, 2008 [Online]. Available: http://www.docmirror.net/en/linux/howto/programming/Secure-Programs-HOWTO/handle-metacharacters.html