

SQL INJECTION VULNERABILITY DETECTION IN WEB APPLICATIONS

By

Jason York

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science in Computer Science

Middle Tennessee State University

May 2014

Thesis Committee:

Dr. Zhijiang Dong

Dr. Cen Li

Dr. Jungsoon Yoo

ACKNOWLEDGEMENTS

I am heartily thankful to my supervisor, Dr. Dong for all his continuous advices, encouragements, and guidance at all times. I want to dedicate this work to my parents for their continuous support and motivation. To all my family and friends for being there at all times when I needed them most.

ABSTRACT

Security is an essential requirement of most web applications, which typically access sensitive data such as personal information, and financial records. Leaking of such sensitive data could cause huge financial losses and hurt the reputation of the organization. However, studies have shown that security vulnerabilities are common in web applications due to the increased pressure on budget and timeline as well as the lack of security training. The goal of the project is to detect one specific kind of security vulnerabilities – SQL injection vulnerability in web applications by exploring source code. The developed tool is easy to use and provides enough flexibility to handle different database extensions.

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	viii
LIST OF SYMBOLS AND ABBREVIATIONS	xi
Chapter	
I. INTRODUCTION.....	1
1. Problem and Motivation	1
2. Approach	2
II. BACKGROUND	4
1. SQL Injection	4
2. Related Works on SQL Injection Detection.....	10
3. Related Works on Vulnerability Detection System.....	12
III. SYSTEM DESIGN	14
1. System Architecture	14
2. Configuration File.....	16
3. Parser	18
3.1. Nikic PHP Parser	19
3.2. Application Parsing Design	23
4. Analyzer.....	24
4.1. Dependency	24
4.2. Vulnerability Analysis	26
4.3. Vulnerability Analysis for Function	28

5. Output	29
IV. IMPLEMENTATION	30
1. Introduction	30
2. Parsing implementation	30
3. Analysis Implementation	32
4. Test Cases	34
4.1. MYSQL Test Cases	35
4.1.1. User Input to SQL Query Directly	35
4.1.2. User Inputs to SQL Query Indirectly	36
4.1.3. Accessing Database through User-Defined Functions	38
4.1.4. Escape method inside Function	39
4.1.5. Branches	41
4.2. MYSQLI Test Cases	42
4.2.1. User Input to SQL Query Directly	42
4.2.2. Escape Method	44
4.2.3. Access Database within Functions	45
4.2.4. Access Database within Function—Parameter Contributes Indirectly.....	47
4.2.5. Multiple Value Source	48
V. CASE STUDY	50
1. Introduction	50
2. Data Collection	52
2.1. Detail Analysis of Each Group	55

2.1.1. Group One	55
2.1.2. Group Three.....	57
2.1.3. Group Nine	60
3. Summary.....	61
VI. CONCLUSION AND FUTURE WORKS.....	63
1. Conclusion	63
2. Future Works	64
BIBLIOGRAPHY.....	66
APPENDICES	68
A. SOURCE CODE	69
B. TEST CASE TABLE STRUCTURE	101

LIST OF TABLES

Table 1 – Languages used.....	53
Table 2 – Page extensions.....	54
Table 3 – Result of Student Projects.....	55

LIST OF FIGURES

Figure 1 – Login page.....	5
Figure 2 – HTML code for login page.....	5
Figure 3 – Login PHP code.....	7
Figure 4 – HTML login form with malicious input.....	8
Figure 5 – Successful login with malicious input.....	9
Figure 6 – Ambiguous function name	11
Figure 7 – System Architecture	14
Figure 8 – Interface for developed tool.....	16
Figure 9 – Example of adding new database extension.....	17
Figure 10 – Example of adding safe functions	17
Figure 11 – Example of adding new user input function.....	18
Figure 12 – Sample PHP code to be converted.....	19
Figure 13 – Nikic’s PHP parser.....	19
Figure 14 – Usage of Nikic PHP parser.....	20
Figure 15 – PHP code to be parsed.....	20
Figure 16 – Generated abstract syntax tree.....	21
Figure 17 – Simple code for converted into array	23
Figure 18 – Sample PHP code	25
Figure 19 – PHP code for function dependency.....	25
Figure 20 – Variable without SQL injection prevention function	27
Figure 21 – Example with SQL injection prevention function.....	28
Figure 22 – Example of function analysis	29

Figure 23 – Example of function parameter tracing	33
Figure 24 – MYSQL Example of basic usages.....	35
Figure 25 – Application output of example code in Figure 24.....	36
Figure 26 – MYSQL Example of user input for query indirectly.....	37
Figure 27 – Application output of example code in Figure 26.....	37
Figure 28 – MYSQL Example of accessing database within function.....	38
Figure 29 – Application output of example code in Figure 28.....	39
Figure 30 – MYSQL Example of escape variables within function.....	40
Figure 31 – Application output of example code in Figure 30.....	40
Figure 32 – MYSQL Example of condition statement.....	41
Figure 33 – Application output of example code in Figure 32.....	42
Figure 34 – MYSQLI Example of basic usage.....	43
Figure 35 – Application output of example code in Figure 34.....	43
Figure 36 – MYSQLI Example of escape method.....	44
Figure 37 – Application output of example code in Figure 36.....	45
Figure 38 – MYSQLI Example of database access within function.....	46
Figure 39 – Application output of example code in Figure 38.....	46
Figure 40 – MYSQLI Example of indirect database access within function	47
Figure 41 – Application output of example code in Figure 40.....	48
Figure 42 – MYSQLI example of multiple value sources.....	49
Figure 43 – Application output of example code in Figure 42.....	49
Figure 44 – Group one login page vulnerable code.....	56
Figure 45 – Application output of group one login page.....	57

Figure 46 – Group three tagprint page vulnerable code	58
Figure 47 – Application output of group three tag print.....	58
Figure 48 – Incorrect used of prepare statement.....	59
Figure 49 – Correct way of using prepare statement.....	59
Figure 50 – Application output of group three insert tag page.....	60
Figure 51 – Application output of group nine user page	61
Figure 52 – Return value.....	65

LIST OF SYMBOLS AND ABBREVIATIONS

PHP – PHP: Hypertext Preprocessor (originally stood for Personal Home Page)

SQL – Structured Query Language

MYSQL – My Structured Query Language

PDO – PHP Data Objects

MYSQLI – MySQL Improved

HTML – Hyper Text Markup Language

CSS – Cascading Style Sheets

JS – JavaScript

AST – Abstract Syntax Tree

XSS – Cross Site Scripting

CHAPTER I

INTRODUCTION

1. Problem and Motivation

Developing a web application is not hard. Most people who have some programming experience could build a simple web application fairly easily. However, it is not easy to build web applications without security vulnerabilities. According to the “Hewlett-Packard 2011 top cyber security risks report,” all the investigated web applications contained at least one vulnerability [7]. One of the major reasons is that developers don’t pay enough attention to the security component because of variant reasons such as lack of experience, deadline pressure, and budget restriction. Therefore, many web applications contain dangerous security flaws, which attract evil attackers and leads to the leak of data that are used for e-commerce, e-shopping, online education, etc. These data are typically confidential and have sensitive information like credit card numbers, social security numbers, and medical records. Those data cannot and should not be viewed or used by unauthorized personnel. The illegal disclosing of those data could lead to lawsuits, financial losses, and reputation damage.

In 2011, a hacktivist group called Anonymous hacked into Sony's PlayStation Network. This incident caused seventy-seven million users’ personal information disclosed, including names, addresses, e-mail addresses, and login details for the PSN [15]. It also prevented users of PSN from playing online. It was one of the largest data security breaches in history, and it lasted twenty four days [15]. This incident is due to a variety of types of vulnerabilities in systems. SQL injection is one of them.

This research focuses on the detection of SQL injection vulnerability in web applications written in PHP. We choose SQL injection and PHP web applications for several reasons. First, many web applications have SQL injection vulnerability. SQL injection attack could combine with other attack techniques, which occurs very frequently, and could cause huge financial losses for organizations and companies. Actually, SQL injection was ranked in top 10 web application vulnerabilities in 2007 and 2010 by the Open Web Application Security Project (OWASP) [16]. In 2013, OWASP Top Ten Project rated SQL injection as the number one attack [16]. Second, PHP is a popular web development programming language, but web applications developed in PHP are the most vulnerable web applications according to the study conducted by Positive Technologies [14]. The study compared the security vulnerabilities of web sites on PHP, ASP.NET, and Java caused by inappropriate software implementation. It showed that 81% of sites in PHP contain critical security vulnerabilities, and 91% contained medium-risk vulnerabilities. Last, but not the least, the tool developed in the research can be used for educational purpose. Students can use this tool to check SQL injection vulnerability on their course projects.

2. Approach

This research develops a tool to detect SQL injection vulnerability by exploring PHP code of the websites. The user interface of the tool is a simple web page which allows the users to specify files or folders to be checked as well as the database extension used in the web site. There are many popular database extensions for PHP, such as MYSQL, MYSQLI, and PDO. Each database extension provides a set of functions or classes to access database. In addition to the support of these common PHP database

extensions, the tool allows the users to add new database extension. All they need to do is to add the extension name and functions that trigger SQL execution to the configuration file.

Once the files and the database extension are provided, the tool will start to detect SQL injection vulnerabilities based on the database extension by exploring the source code. When the process is completed and SQL injection vulnerability exists, the tool will output detailed information about the SQL injection vulnerability including the file name, the line number and the variable or function that causes the security vulnerability

To evaluate the effectiveness of the tool, a case study on student projects from database classes has been conducted. The result shows that the tool can detect SQL injection vulnerabilities accurately and efficiently.

The main advantage of the tool that distinguishes it from existing vulnerability-detection applications is that it uses the white-box testing approach by exploring source code. White-box testing means it will audit the code instead of just checking the functionality. It gives the developers a better understanding of where the code is vulnerable [3].

CHAPTER II

BACKGROUND

1. SQL Injection

An SQL injection attack occurs when an application does not validate the input from users and gives evil attackers the chance to influence the SQL query. The attacks usually happen when the web page, like login page, produces SQL statements based on user inputs to retrieve data from database servers located behind web applications [8]. The attackers could insert a malicious query into the web page instead of developer expected user name and password.

Consider the basic login page shown in Figure 1. The login page is used to verify the identity of legal users. A sample HTML code of the login page is given in Figure 2. As shown in the HTML code, the action of validating users is in the login.php file.

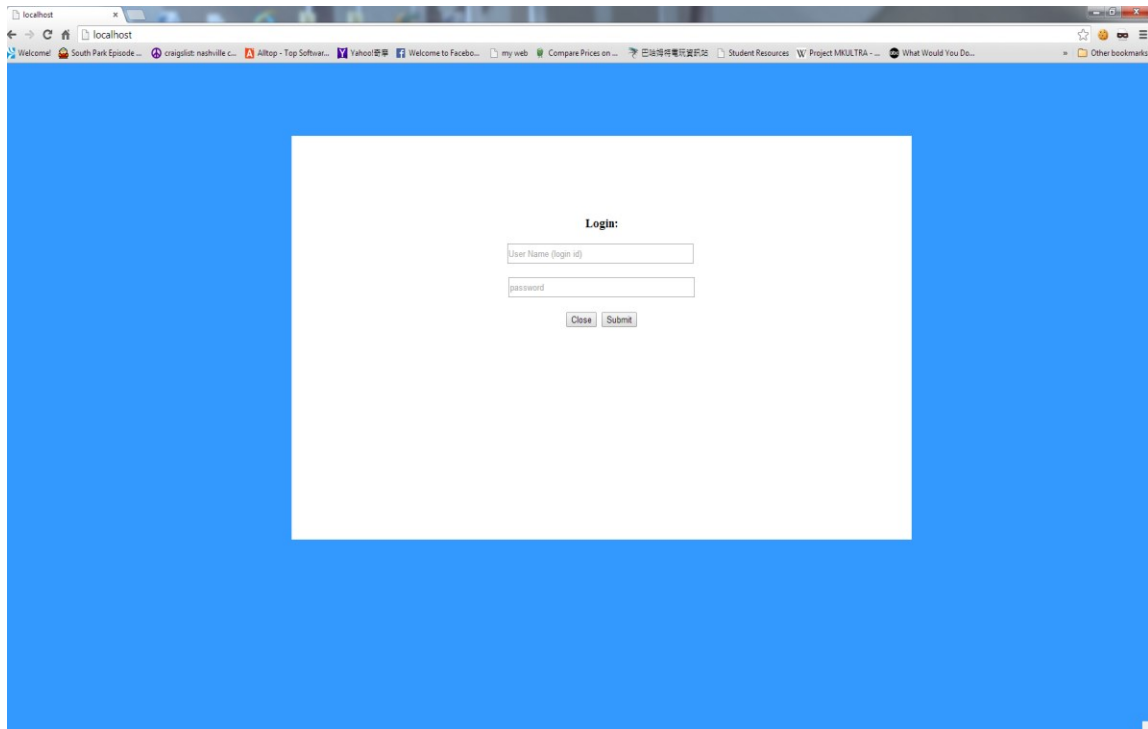


Figure 1. Login page

```
<form action="login.php" method="post" style="margin:0px;">
<!-- Modal -->
<div class="modal-header">
<h3 id="myModalLabel">Login:</h3>
</div>
<div class="modal-body">
<input name="user_ID" class="input-xlarge" type="text"
placeholder="User Name (login id)">
<br>
<input name="user_password" class="input-xlarge"
type="password" placeholder="password">
<div class="modal-footer">
<button class="btn-danger" data-dismiss="modal" aria-
hidden="true">Close</button>
<button class="btn-primary">Submit</button>
</div>
</div>
</form>
```

Figure 2. HTML code for login page

A sample login.php is given in Figure 3, When the submit button in the login page is clicked, the login.php script is executed in server side to construct and submit a SQL query to database servers to verify the existence of the user with the given username and password. If such user doesn't exist in the system, the access to the system will be rejected.

```
<?php
$tempID = "" . $_POST['user_ID'] . "";
if($row = mysql_fetch_assoc(mysql_query("SELECT Session_exp
FROM User Where Id =$tempID")))
if(strtotime($lastLogin) < strtotime('-90 days'))
{
$tempPW = "" . $_POST['user_password'] . "";
$query = "SELECT * FROM User Where Id = $tempID AND
Password = $tempPW ";
}
else
$query = "SELECT * FROM User Where Id = $tempID";
echo $tempID. " " . " " . $lastLogin;
$result = mysql_query($query) or die("Query failed : " .
mysql_error());
if($row = mysql_fetch_assoc($result))
{
$user = new user;
$user->Id = $row['Id'];
$user->Password = $row['Password'];
$user->Session_exp = $row['Session_exp'];
$user->FName = $row['FName'];
$user->LName = $row['LName'];
$_SESSION['user'] = $user;
}
if(isset($_SESSION[user]))
{
if(strtotime($lastLogin) < strtotime('-90 days'))
$storeSession = mysql_query("Update User SET
Session_exp=NOW() where Id= $tempID");
mysql_query("INSERT INTO Log VALUES (" . $tempID . ", '" .
$_SERVER['REMOTE_ADDR'] . "', '" . getenv('COMPUTERNAME') .
"', NOW())");
header( 'Location:'. 'homepage.php' );
}
else
header( 'Location:'.
$_SERVER['HTTP_REFERER'].'?login=fail' ) ;
?>
```

Figure 3. Sample login.php file

As long as regular names and passwords are entered, the above login page works fine. Unfortunately, attackers can access the system without entering correct username and password. For example, an attacker could enter 'OR '1'='1' for the password and nothing for the username as shown in Figure 4:

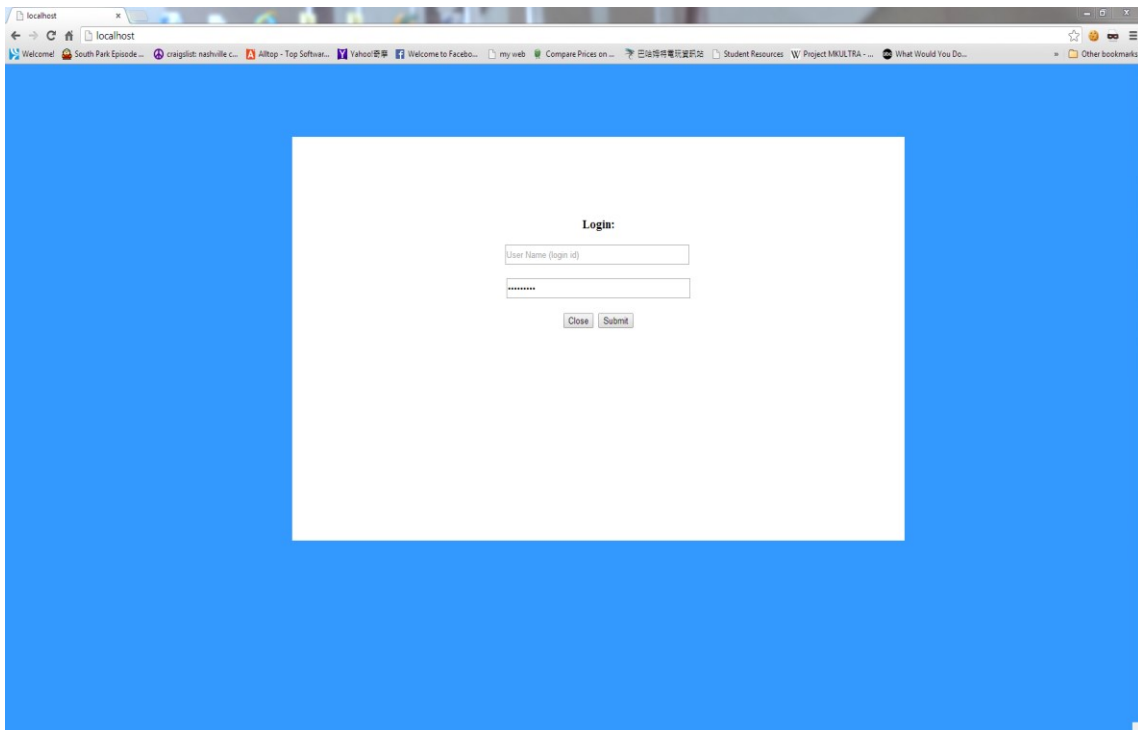


Figure 4. Malicious input in login page

The query generated in the sample login.php file from the above malicious input is:

```
SELECT * FROM User Where Id = '' AND Password = '' OR  
'1'='1'
```

Because of the OR statement in the SQL query, the application checks '1' does equal '1', thus the query will return TRUE, resulting in a successful login as shown in Figure 5.

Another way to skip password checking is to type ' OR 1=1 --' to the user name and leave password empty. The generated query by the code becomes the following:

```
SELECT * FROM User Where Id = '' OR 1=1 --' AND Password = ''
```

Since -- in SQL comments out the rest of the line, the where condition in the above query will ignore the password checking and therefore always be true. This particular attack is frequently used to bypass the system authentication.

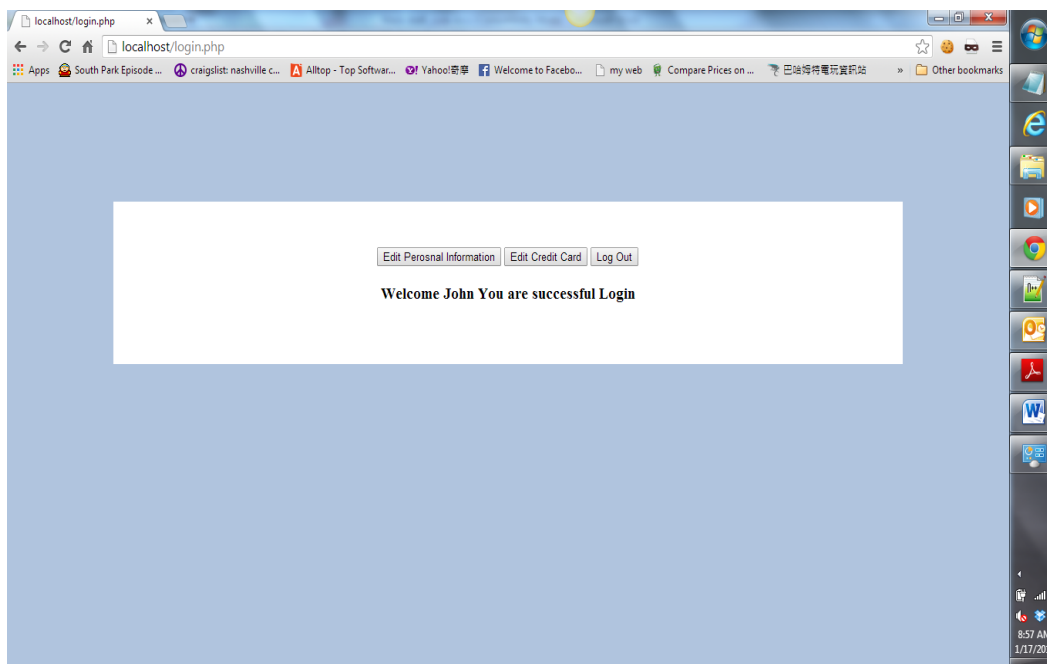


Figure 5. Successful login with malicious input

Thus, the above sample login page has a SQL injection vulnerability, which allows attackers to log in successfully without providing valid usernames and passwords. Such SQL injection vulnerability could lead to a leak of sensitive information and loss of confidentiality. However, it could be even worse. For example, an attacker could empty or modify a table by appending a DELETE, UPDATE, INSERT, or DROP SQL statement. This action would be disastrous to a web page's information integrity and availability. Consider the following SQL code snippet, which can be generated when attackers enter `' ; DROP ALL TABLES; --` for password and leave username field empty:

```
SELECT * FROM User Where Id = '' AND Password = '' ; DROP ALL TABLES; --'
```

If the above query is executed, although attackers don't get authentication to access the database, the injected DROP statement would be executed successfully and completely erase all the data and all the tables in the database. Unless a database backup is available, all the data would be lost. If the attackers use UPDATE or INSERT SQL statements to pollute the database, the users could retrieve deceitful data that could cause confusion.

2. Related Works on SQL Injection Detection

Etienne Janot and Pavol Zavorsky's work "Preventing SQL Injections in Online Applications" [4] describes how to prevent SQL injections on the applications. The article describes techniques, like active input data encoding, instruction-set randomization, and query pre-modeling to prevent SQL injection from happening.

However, the paper does not describe how to detect SQL Injection if the vulnerabilities exist within the code.

To detect all the vulnerabilities, one has to look inside the code and search for any place that may be vulnerable to SQL injection. In “SQL Injection Attacks and Defense,” Justin Clarke provides useful methods such as pattern matching and backward tracing [6]. However, pattern matching can lead to false positive results if used alone. For example, the pattern matching leads to a false positive for the following example because these two functions are similar and it cannot distinguish one from the other:

```
$result = MyCustomFunctionToExec_mysql_query($sqlStm);  
$result = mysql_query($sqlStm);
```

Figure 6. Ambiguous function name

Pattern matching alone is not suitable for large or complex applications. However, if other approaches are applied on top of pattern matching, the overhead cost will increase, and performance will decrease. To counter this problem, our tool uses PHP parser with pattern matching [6].

Justin Clarke also talks about using trace-back technique to detect SQL injection vulnerable code. This approach traces a variable back to its root. It will be used to eliminate unnecessary work for the detection function which improves performance [6]. Our tool also adopts this trace-back technique.

3. Related Works on Vulnerability Detection System

There are commercial web application scanners to detect security vulnerabilities, such as Acunetix WVS by Acunetix [1], AppScan by IBM [5], and Retina Web Security Scanner by BEYONDTRUST [2]. Those applications are non-freeware. Some of them cost a fortune to use, and are, therefore, not very good candidates for small companies or students.

There are also some open-source web application scanners such as Wapiti by Nicolas Surribas [9], Grabber by Romain Gaucher [12] and Wikto by SensePost [13]. Those are free of charge to use, but they have limited features and are not as powerful as the commercial scanner. Some of them may lack the ability to detect certain types of vulnerabilities, have lower performance rates, and consume more memory.

Those web application scanners focus on black box techniques, which take users' URL's and scan each web page using the tree structure of the webpages. They inject malicious code into the website and investigate the website's vulnerability to injections. However, they are not able to see the hidden files or server-side files.

One drawback of those applications is they do not tell the users where exactly the vulnerable code is, like which lines of code cause the security breaches. The applications only tell users how many threats the website may have and the threat level of each, ranked from high to low. The information is somewhat important if developers have some knowledge about security. However, they will not be able to understand what each means or will not be able to know where to start looking for security holes.

Most web vulnerability scanners on the market have one common approach: most of them use black-box testing. It will give the developers a clear view of how secure the application is, but not where the vulnerabilities are.

CHAPTER III

SYSTEM DESIGN

1. System Architecture

The architecture of the system, as shown below, contains four major components: Configuration file, PHP parser, Analyzer, and Output. Each of them performs different sets of tasks.

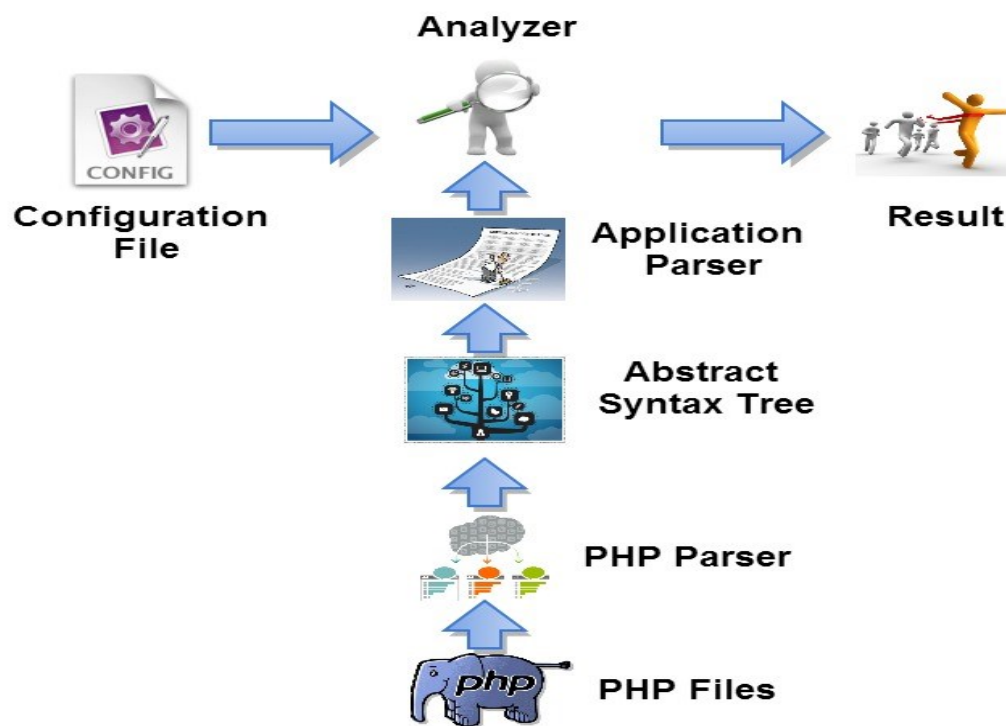


Figure 7. System Architecture

The configuration file provides information about PHP database extensions, which adopted by web applications to access databases. It tells the analyzer the types of database extensions, user input methods, prevention methods, and functions to be

evaluated. The information is essential for the analyzer and hard to retrieve from web applications.

The PHP parser builds an abstract syntax tree from PHP code. It takes the files from the user and converted them into abstract syntax trees, which specify the syntactic structure of PHP code. It has all the information about each variable and function.

The analyzer takes database extension information from the configuration file as well as the program structure from the parser. A dependency tree is built to help analyze the data. Trace back process is performed at the analysis step, to properly identify the ancestors of variables based on the dependency tree. The ancestors of variable *var* are a set of variables or values, each of which can affect the value of the variable *var*.

There will be two different output results: the overall result and function only result. Overall result contains information of all vulnerabilities based on the entire dependency tree. On the other hand, function only result contains information of vulnerable functions that access database but have SQL injection vulnerability. Those results will be populated during the analysis step if a SQL injection vulnerability is detected. The output includes file name, line number of the code and unsafe variables that generate SQL injection vulnerability.

The interface of the tool is shown in Figure 8. Once the source code of web applications and the configuration file is submitted, the PHP parser starts parsing and builds an abstract syntax tree. The analyzer will detect SQL injection vulnerabilities based on the abstract syntax tree and information specified in the configuration file. If vulnerable code exists, the analyzer will write out the debug information associated with the vulnerable code.

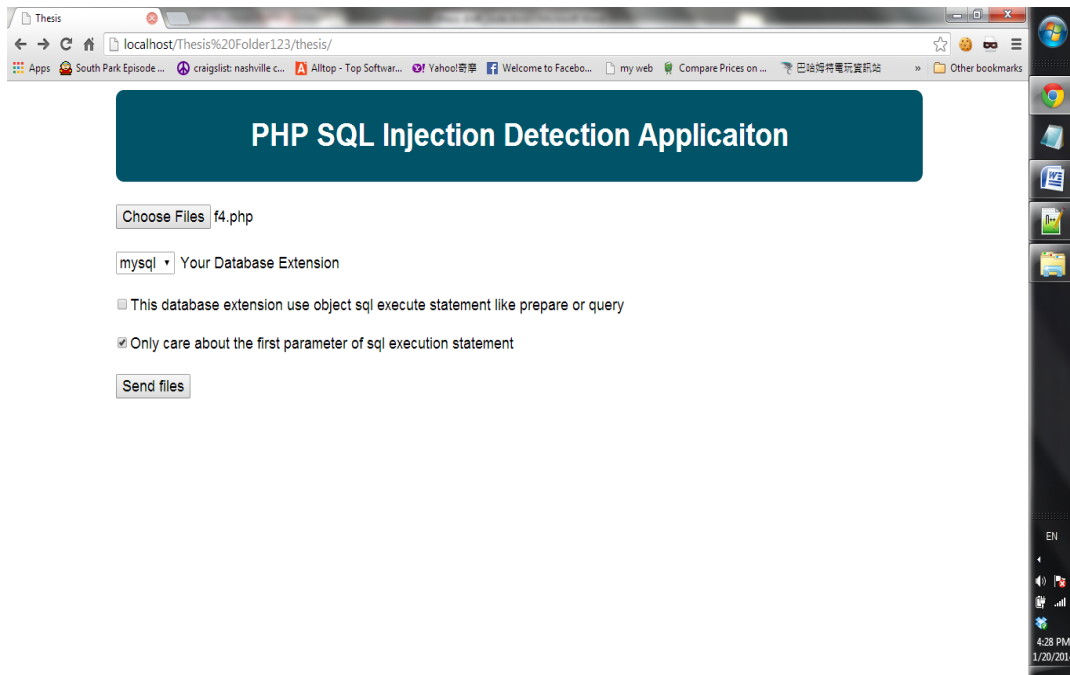


Figure 8. Interface for developed tool

2. Configuration File

There are different PHP database extensions that can be used to access database. The most common ones are MYSQL, MYSQLI, and PDO to access MySQL database, and MSSQL to access MS SQL Server. The configuration file allows users to tell the analyzer which database extension is used. The default options are MYSQL, MYSQLI, PDO, and MSSQL. Different strategies are adopted based on the provided database extension to detect SQL injection vulnerabilities.

The users are also allowed to add any new database extension. The configuration file requires users to insert the database name preceded by @ character, followed by all database access functions the users wish to evaluate. The following code is an example

for adding MYSQL database extension to the configuration file. This example requires the tool to detect SQL injection vulnerabilities in all occurrences of `MYSQL_QUERY` function calls in the code.

```
@mysql
MYSQL_QUERY
```

Figure 9. Example of adding new database extension

Users are allowed to add functions they believe will make the variables safe. The default value for it is `MYSQL_REAL_ESCAPE_STRING` which escapes the special character from the SQL statement. Users could add, edit, or delete any functions. To add more function to the safe function section, users need to insert the function between `#safe` and `#database`, as shown in the following example.

```
#safe
MYSQL_REAL_ESCAPE_STRING
NewSafeFunction
#database
```

Figure 10. Example of adding safe functions

Furthermore, users could specify the approach used in PHP code to retrieve user inputs. In the PHP, there are three main approaches: GET, POST, and REQUEST. Users could modify the configuration file to insert or delete any user input identifiers they wish.

Any additional input method should be inserted after `#php`. For example, if a new user input method is introduced in PHP called `RECEIVE`, the user could add the following code and subsequently, the new input will be considered in the analyst step.

```
#php  
RECEIVE
```

Figure 11. Example of adding new user input function

The configuration file also allows users to specify the type of PHP database extensions, either functional or object-oriented. For example, `MYSQL` database extension provides a set of functions to access database, while `PDO` database extension provides a set of objects to access database. For `MYSQLI` database extension, users can use either functions like `mysqli_query`, or object oriented features like query statement to access database.

Since not every parameter in the database access functions will be used to construct SQL query, like the second parameter in `mysql_query` function, it is more efficient to check the function parameters only that affect the construction of SQL queries. Therefore, the configuration file allows users to specify the parameters that will not produce SQL injection vulnerability.

3. Parser

The parser component consists of two parts: niki PHP parser [10] and application parser. The niki PHP parser is an open source PHP parser, which parses the PHP code

and generates an abstract syntax tree. However, the generated abstract syntax tree contains lots of information that is not needed by our tool. Therefore, the application parser takes the abstract syntax tree as input and converts it into some concise form. The following is a snippet of code that is converted into more analyzable form.

```
<?php
$var3=$_GET['Fname'];
mysql_query("SELECT * FROM Person where
FirstName=".$var3,$link);
?>
```

Figure 12. Sample PHP code to be converted

The above code is eventually converted into the following by the application parser:

```
Array ([var3] => Array ([0] => _GET) []) => Array ([fun] =>
mysql_query [line] => 3 [0] => var3 [1] => link)
```

3.1 Nikic's PHP Parser

The parsing step utilizes Nikic's open source PHP parser. The parser converts PHP code into an abstract syntax tree. It also has the functionality to convert a syntax tree back to PHP code which allows the developer to do code preprocessing. It is simple to install. The developer only needs to include the folder in the code as the following code [10]:

```
require '/PHP-Parser/lib/bootstrap.php';
```

Figure 13. Nikic's PHP parser

In order to parse source code, the user first has to create a `PHPParser_Parser` object by taking a `PHPParser_Lexer` instance as the parameter. The PHP code is passed as a string to parse method of the `PHPParser_Parser` object. If a syntax error occurs, the application will throw a `PHPParser_Error` if no catch is used. Figure 14 shows the basic usage of Nikic PHP parser.

```
<?php
$code = '<?php // some code';
$parser = new PHPParser_Parser(new PHPParser_Lexer);
try {
    $stmts = $parser->parse($code);
} catch (PHPParser_Error $e) {
    echo 'Parse Error: ', $e->getMessage();
}
```

Figure 14. Usage of Nikic PHP parser

For example, Figure 15 shows the PHP code to be parsed, and the generated abstract syntax tree is displayed in Figure 16.

```
$code = "<?php echo 'Hi ', getTarget();"
```

Figure 15. PHP code to be parsed

```

array(
  0: Stmt_Echo(
    exprs: array(
      0: Scalar_String(
        value: Hi
      )
      1: Expr_FuncCall(
        name: Name(
          parts: array(
            0: getTarget
          )
        )
        args: array(
        )
      )
    )
  )
)

```

Figure 16. Generated abstract syntax tree

Since there is only one statement in the code, the parser will generate only one array, with only one node. The node is an instance of `PHPParser_Node_Stmt_Echo`.

PHP is a large language with a variety of node types; therefore, the PHP parser groups the tree nodes into the following four categories [10]:

- I. `PHPParser_Node_Stmts` are statement nodes, i.e. language constructs that do not return a value and cannot occur in an expression. For example, a class definition is a statement. It does not return a value, and something like `func(class A { });` cannot be written.

II. `PHPParser_Node_Exprs` are expression nodes, i.e. language constructs that return a value and thus can occur in other expressions. Examples of expressions are `$var` (`PHPParser_Node_Expr_Variable`) and `func()` (`PHPParser_Node_Expr_FuncCall`).

III. `PHPParser_NodeScalars` are nodes representing scalar values, like 'string' (`PHPParser_Node_Scalar_String`), 0 (`PHPParser_Node_Scalar_Lnumber`) or magic constants like `__FILE__` (`PHPParser_Node_Scalar_FileConst`). All `PHPParser_NodeScalars` extend `PHPParser_Node_Expr`, as scalars are expressions, too.

IV. There are some nodes not in either of these groups, for example names (`PHPParser_Node_Name`) and call arguments (`PHPParser_Node_Arg`).

Every node has a number of sub nodes. The user can access sub nodes by writing `$node->subNodeName`. The above example has only one sub node "exprs."

Therefore, to retrieve the statement, the user can write `$stmts[0]->exprs`. If the user wants to retrieve the name of the function call, he or she would write `$stmts[0]->exprs[1]->name`.

Furthermore, the PHP parser gives the user extra information such as node types, node attributes, start line, etc. The information can be retrieved with PHP functions. The type is the class name with the `PHPParser_Node_` prefix. Some examples are `PHPParser_Node Stmt_If`, `PHPParser_Node_Expr_Variable`, and `PHPParser_Node Stmt_Function`. The `PHPParser_Node Stmt_If` is a node of if statement, `PHPParser_Node_Expr_Variabl` is a a node of a expression variable, and `HPParser_Node Stmt_Function` is a function node. By default, the parser adds the start

line number, end line number, and comments attributes. They can be retrieved through provided functions such as `etline()`, `ibute()`, `getAttribute()` and `getAttributes()`. Also, it is possible to associate custom metadata with a node using the `setAttribute()` method. Comments can be retrieved with `getAttribute('comment')`. The start line can be accessed using `getLine()` or `etline()` instead of `getAttribute('startLine')`.

3.2 Application Parser

The application needs to know each variable and function to determine if the code is vulnerable to SQL injection attacks, but those data need to be separated and identified. Therefore, the application has two main arrays, one for functions and one for variables. Everything that is not a function will be put into the variable array. Anything that is inside a function will be put into the function array. The code in Figure 17 will be converted into a variable array:

```
<?php
$var=$var2;
?>
```

Figure 17. Simple code for converted into array.

The full abstract syntax tree generated from PHP parser will look like the following:

```
PHPParser_Node_Expr_Assign Object ( [subNodes:protected] =>
Array ( [var] => PHPParser_Node_Expr_Variable Object (
[subNodes:protected] => Array ( [name] => var )
[attributes:protected] => Array ( [startLine] => 2
[endLine] => 2 ) ) [expr] => PHPParser_Node_Expr_Variable
Object ( [subNodes:protected] => Array ( [name] => var2 )
[attributes:protected] => Array ( [startLine] => 2
```

```
[endLine] => 2 ) ) ) [attributes:protected] => Array (
[startLine] => 2 [endLine] => 2 ) )
```

The array contains many unnecessary information. Therefore, the application parser only collects the data it needs and categorizes them into the respective array. The variable array for the above example will look like

```
Array ([var] => Array ([0] => var2))
```

The variable name is the key of the array. The array contains another array which includes all the assigned value.

4. Analyzer

Once all the necessary data are collected, the analysis step will be performed. The first step of analyzing the data is to determine if the variables or the functions have relation to each other. Therefore, a dependency tree is constructed. Only the functions or variables that contribute to SQL query construction and execution are considered. After the dependency tree is built, the application will verify SQL injection vulnerability for every statement that accesses database.

4.1 Dependency

The dependency function traces back each variable or function call to their original variables. To accomplish this, the function uses the key and value structured arrays to determine if the value is the original value. If a value is a key in the array, then the function keeps tracing back until the value is not a key. The end value is the original value. The following PHP code is an example of trace back:

```
<?php
$firstName = $_GET['Fname'];
$middleName = $firstName;
$lastName = $middleName;
?>
```

Figure 18. Sample PHP code

The variable `lastName` is assigned to `middleName`, and the `middleName` is assigned to `firstName`. In the trace-back process, the application looks into the variable `lastName` and checks which array contains the key with the `lastName` assigned value. The process continues until either no assigned value is found or user input is reached. Therefore the application will determine that the variable `lastName` is affected by `$_GET['Fname']`.

To trace back a function call, the application need to use the function array and may need to use the variable array to complete the dependency function. The PHP code in Figure 19 is used to demonstrate how the function dependency is created.

```
<?php
$test = $_GET['Fname'];
myFunctionCall($test);
function myFunctionCall ($var){
echo $var;
```

```
}
?>
```

Figure 19. PHP code for function dependency

Variable array for the above code looks like the following array:

```
Array ([test] => Array ([0] => _GET) []) => Array ([fun] =>
myFunctionCall [line] => 3 [0] => test))
```

Function array for the above code looks like the following array:

```
Array ([myFunctionCall] => Array ([var] => Array ( )))
```

In the above PHP code, the function `myFunctionCall` is called. In order to evaluate the function call, one must determine if the function is vulnerable. First, the dependency function will look for the function call `myFunctionCall` in the variable array and then it searches for it in the function array. If the function name matches the key in function array, then the dependency will be created base on the matched key and value.

4.2 Vulnerability Analysis

During the vulnerability's analysis step, the application will first look up the arrays to see if any SQL execution or user-defined function in the configuration file are used to access database. If such a function call is found, the application will look at the dependency tree and trace back the variable to verify if any user input is involved directly or indirectly to construct the SQL query. If user input is involved, then the application checks if any prevention function is applied to the user input. If no prevention functions are found, the application will alert the user that this function call is vulnerable to SQL

injection. The PHP code in Figure 20 demonstrates how the application determines which variables are unsafe.

```
<?php
$test2 = $_GET['Fname'];
$var2=$test2;
mysql_query("SELECT * FROM Person where
FirstName".$var2,$link);
?>
```

Figure 20. Variable without SQL injection prevention function

The application first looks at the SQL execution statement `mysql_query` and then traces back the variable `$var2` that is used to construct the query in the function call. Therefore, by looking up the dependence tree, the application traces back to `$test2` from `$var2`. Because `$test2` contains the user input and no prevention function is applied to `$test2` or `$var2`, the application will alert the user that the function call `mysql_query` is vulnerable to SQL injection attack.

Figure 21 shows a similar example. The function call `mysql_query` is safe because `mysql_real_escape_string` is applied to `$test3` so that it is impossible for attackers to pollute the SQL query. To analyze the code in Figure 21, the application will perform the same process illustrated in the earlier example, but

instead of warning the user, the application will not throw any messages because there is no SQL injection vulnerability.

```
<?php
$test3 = mysql_real_escape_string($_GET['Fname']);
$var3 = $test3;
mysql_query("SELECT * FROM Person where
FirstName".$var3,$link);
?>
```

Figure 21. Example with SQL injection prevention function

4.3 Vulnerability Analysis for Function

In addition to detect the SQL injection vulnerability in functions provided by the database extensions, we also need to detect vulnerabilities in user defined functions that access databases. The reason to provide this feature is because in large web applications, a group of user-defined functions may be provided so that the rest of web applications could use these functions to access databases. Instead of checking all PHP files, only the group of user defined functions need to be verified. If there is no vulnerability in these functions, we can say the web application is safe with regard to SQL injection attack.

We use the PHP code in Figure 22 to explain the idea. In this code, the SQL execution function is called to access database within the function

“myFunctionCall,” and the function parameter \$var is used to construct the SQL query. When myFunctionCall is invoked, the variable \$test is passed as argument. Since \$test is escaped by a SQL injection prevention function, the PHP code is not SQL injection vulnerable. However, the function itself is vulnerable to attack since the function myFunctionCall could be invoked (in other places) with unescaped user input. Thus, the application will give a warning to the user that the function myFunctionCall is vulnerable to attack in the function result section.

```
<?php
$test = mysql_real_escape_string($_GET['Fname']);
myFunctionCall($test);
function myFunctionCall ($var) {
$query = "SELECT * FROM users WHERE user=".$var;
mysql_query($query);
}
?>
```

Figure 22. Example of function analysis

5. Output

The application displays two groups of results: code vulnerability result and function vulnerability result. Each of them provides different information of the given data. The code vulnerability result displays all detected SQL injection vulnerabilities; while the function vulnerability result show all unsafe functions, which may or may not produce any SQL injection vulnerability in the current code.

To help developers locate vulnerable code or functions, the application provides detailed information including file name, total number of detected vulnerabilities, vulnerable code or unsafe functions and their line numbers as well as detailed error message.

CHAPTER IV

IMPLEMENTATION

1. Introduction

The developed application is a simple and clean web page. The users use the page to submit PHP files as well as database extension information. Based on user provided database extension, the application reads all necessary information about the database extension from the configuration file and starts SQL injection vulnerability.

The detection of SQL injection vulnerability is divided into two phases. The first phase is parsing, which retrieves useful program information from the PHP code and store them in respective arrays. The second phase is data analysis to detect vulnerable code by looking up stored data.

2. Parsing Implementation

The major class that was adapted from Nikic's PHP Parser for this application is PHPParser_Parser. The method "parse" in the class is used frequently to generate an abstract syntax tree for a piece of PHP code. And then, useful information is retrieved from the abstract syntax tree and stored in variable and function arrays, respectively. The

focus objects of the arrays are name and value. The name could be a variable name or a function name. Value could sometimes be a function name.

The application defines a class named `myPar` to handle the parsing process. The class contains several major methods such as `restart`, `breakStmts`, `arrayHandle`, `fun`, and `general`, as well as several trivial functions to set and gather data from the configuration file and from functions.

This application loops through and evaluates each file individually; thus, new arrays are needed for each file. The data generated from the previous file should be removed when start processing the next file. Therefore, a `restart` function is implemented to reset all the arrays and variables. It is the first function called whenever a new file is being processed.

`ArrayHandle` function is used to manage the variable and function arrays. The function assigns incremental index numbers for statements that are not assigned by a variable. It also removes duplicate keys in the same array. For example, if a key is assigned to a different value, it will keep the old value and add the new value into the array, instead of creating another new array with the same key.

`BreakStmts` function acts as a distributor in the class. It sorts the statements and determines whether they belong to the function or the variable array. Since Nikic's PHP Parser defines and categorizes `if`, `elseif`, and `else` statements as different from any other conditional statement, they need to be sorted differently. Hence, there are five different categories that need to be sorted: `if`, `elseif`, `else`, other conditional statements, and the remaining statements. If a PHP statement is an `if`, `elseif`, `else`, or conditional

statement, a recursion will occur until the statement is not a function statement. If it is a function statement, it will be sorted to the `fun` function; otherwise, it will call the `general` function.

The `general` function has a recursion method which repeats itself until the name is reached. This function also captures the line of each `Fun` function, handles all the variables and statements that are inside the function, and puts them into the function array.

3. Analysis Implementation

The analysis process is handled by the `myAn` class in this application. Major functions in this class include `checkFun`, `getTarget`, `traceback`, `checkParam`, and `focusTarget`. There are several trivial functions to retrieve and set data from user input and the configuration file.

In the analysis process, all that needs to be validated is the statements that access database. Therefore, the first step in the analysis is to filter out any statements that do not access database, which is the major job of the `checkFun` function. In other words, the `checkFun` function explores the generated abstract syntax tree to find all database access function call and store them into an array.

The `getTarget` function uses the array returned from the `checkFun` function. It breaks down the potential statement into variables and sends it to the `focusTarget` function for further evaluation. It also checks to see if the statement has any parameters. If it does, the parameters are sent to the `focusTarget` function for later use.

An important piece of the analysis process is being able to know what the variables actually are. Variables may not be constant. They may change at different times. In Figure 18, the variable “lastName” is assigned to the variable firstName. The application has no idea what the variable “middleName” is if the application does not trace back what the variable middleName is. Therefore, the `traceBack` function serves the purpose of finding the root value of the variables.

The `checkParam` function is invoked in the `focusTarget` function. It serves the purpose of tracing back the parameters. Since the parameters’ variables are treated differently from the normal variables, the `traceBack` function will not work. Considering the code in Figure 23, the function `abc` is invoked with two arguments `$test2` and `$test`. Inside the function, an SQL execution statement is called, and the parameter `$var6` is used to construct the query. This application needs to determine the value of the parameter `$var6` is actually variable `$test3` defined outside the function. The `checkParam` function serves this purpose to trace back parameter variables inside the function to outside variables.

```
<?php
//function calling sql statment with reference variables
$test2 = ($_GET['Fname']);
$test3 = mysql_real_escape_string($_GET['Fname']);
abc($test2,$test3);
function abc($var5,$var6) {
mysql_query("SELECT * FROM Person where
FirstName=".$var6,$link);
}
?>
```

Figure 23. Example of function parameter tracing

The last step of the analysis phase is implemented in the `focusTarget` function. It uses all the information that was collected by other functions to determine if the PHP code are vulnerable to SQL injection or not. The function first checks if users want to evaluate the whole PHP code or just a group of user-defined database access functions.

To evaluate the whole PHP code, both the variable and function arrays will be checked. If a variable is not a user input, then the variable is safe. If it is a user input, then this application will check if the SQL injection prevention is applied to it. If the prevention method is used, then the variable will be safe, else this application will evaluate if the variable is used to construct a SQL query. If yes, it then determines if the query is used in the SQL execution function, if yes, then the code is not safe, else it is safe.

To evaluate the safety of user-defined functions, only the function array is used. Since in this step, only local variables defined with the functions are checked, it is simpler than the previous case. This application only evaluates any SQL execution statement that is called inside the function. If any SQL execution statement is found, then this application traces back the variable inside the function scope. If no SQL injection prevention method is applied to the variable that are used to construct SQL query, then we say the function itself is unsafe.

4. Test Cases

Ten test cases are designed, five for MYSQL and five for MYSQLI, are used to ensure major features of this tool are implemented correct. MYSQL and MYSQLI database extensions are chosen because they are the two main database extensions and other database structures are similar to them. MYSQL will cover all the database extensions that use structures like `somedatabase_query`. MYSQLI will cover all the database extension that used SQL execution method like `someObject->prepare($sql)` or `someObject->query($sql)`

4.1 MYSQL Test Cases

4.1.1 *User Inputs to SQL Query Directly*

The first test case is to detect the situation where user inputs are used directly to construct SQL queries in MYSQL execution statements. The sample code for the test case is given in Figure 24. There are three database execution statements. One uses an unprotected variable, one uses a protected variable, and the last one uses a value straight from the user input.

```
<?php
$test2 = ($_GET['Fname']);
$test3 = mysql_real_escape_string($_GET['Fname']);
mysql_query("SELECT * FROM Person where FirstName".$test2);
mysql_query("SELECT * FROM Person where FirstName".$test3);
mysql_query("SELECT * FROM Person where
FirstName".$_GET['Fname']);
?>
```

Figure 24. MYSQL Example of basic usages

Since the first and the third execution statements' variables are not validated, the application returns the variables with warning.

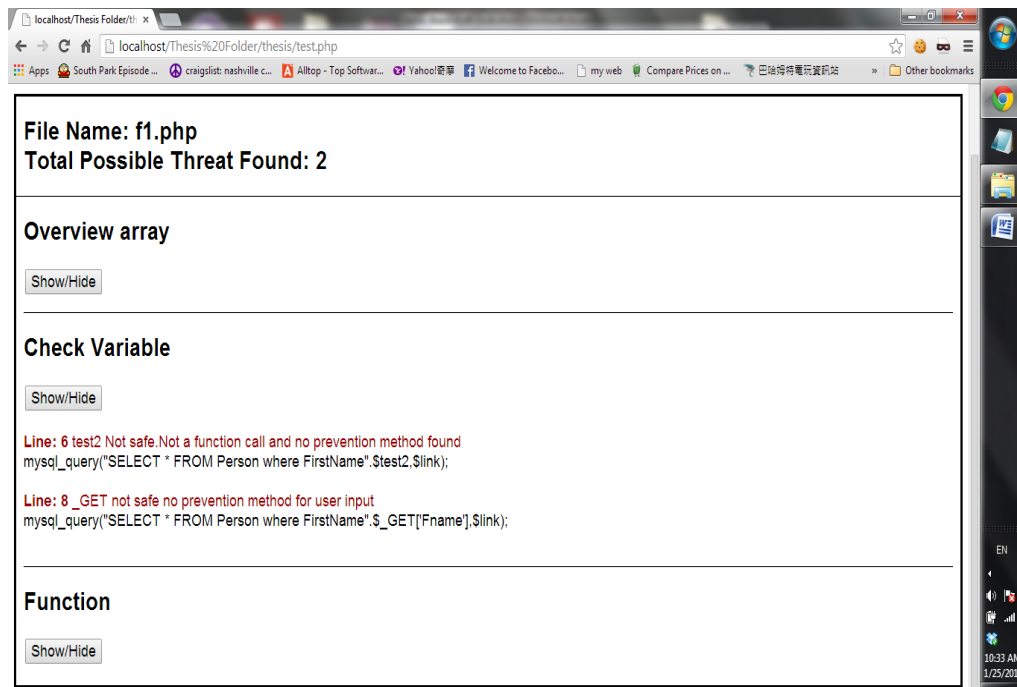


Figure 25. Application output of example code in Figure 24

4.1.2 User Inputs to SQL Query Indirectly

The second test case is to detect the situation where user inputs are used indirectly to construct SQL queries in MYSQL execution statements. It means the variables/values used to construct SQL query depends on user inputs indirectly. The sample code for the test case is given in Figure 26. The variable `$var2` refers to the variable `$test2` which is an unsafe variable. The variable `$var3` refers to the variable `$test3` which is a safe

variable. There are two SQL execution statements. The first one uses the unsafe variable, and the second uses the safe variable.

```
<?php
$test2 = ($_GET['Fname']);
$test3 = mysql_real_escape_string($_GET['Fname']);
$var2 = $test2;
$var3 = $test3;
mysql_query("SELECT * FROM Person where FirstName".$var2);
mysql_query("SELECT * FROM Person where FirstName".$var3);
?>
```

Figure 26. MYSQL Example of user input for query indirectly

As shown in Figure 27, the application gives the one with the unsafe variable a warning.

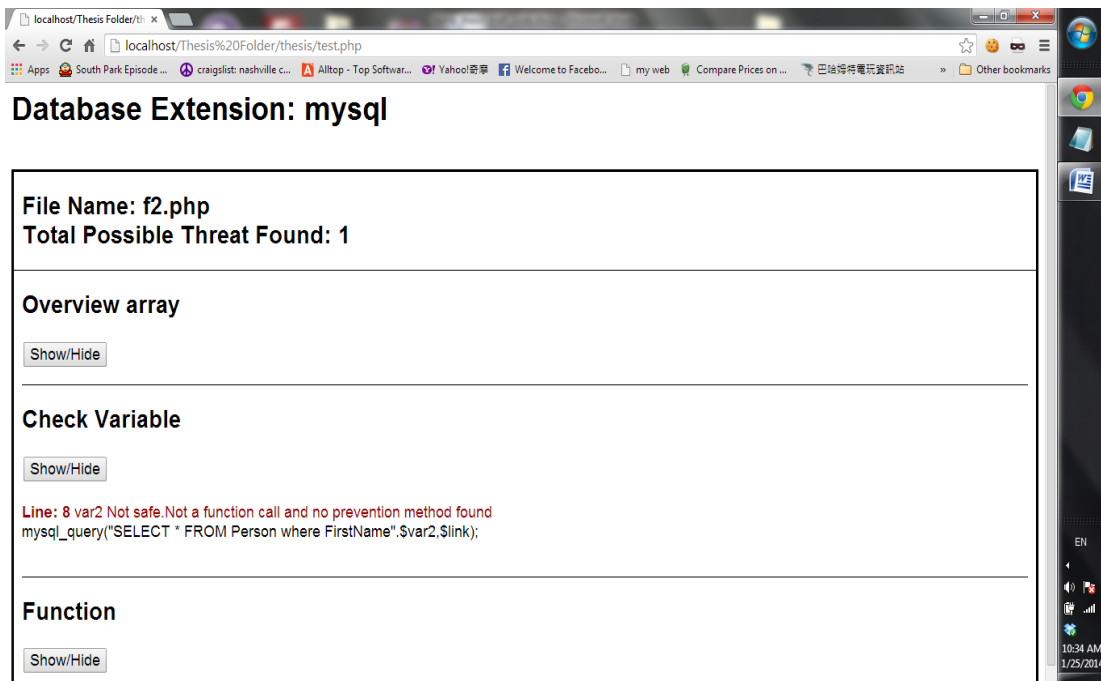


Figure 27. Application output of example code in Figure 26.

4.1.3 Accessing Database Through User-Defined Functions

The third test case is to detect the situation where user-defined functions are used to construct and submit SQL queries. The SQL statement as shown below is called through the function `abc`. The variables are passed through outside variables. The first statement uses the unsafe variable `$var5` which is the variable `$test2`. The second statement uses a safe variable `$var6` which is the variable `$test3`.

```
<?php
$test2 = ($_GET['Fname']);
$test3 = mysql_real_escape_string($_GET['Fname']);
abc($test2,$test3);
```

```
function abc($var5,$var6) {
mysql_query("SELECT * FROM Person where FirstName=".$var5);
mysql_query("SELECT * FROM Person where FirstName=".$var6);
}
?>
```

Figure 28. MYSQL Example of accessing database within function

As shown in Figure 29, the result returns one warning for the overall variable and two for the function only. The warning shows because the variable `$test2` is unsafe since no prevention method is applied. There are two warnings in the function because the variables inside the function are not validating. If the function is used somewhere else, the code may be vulnerable.

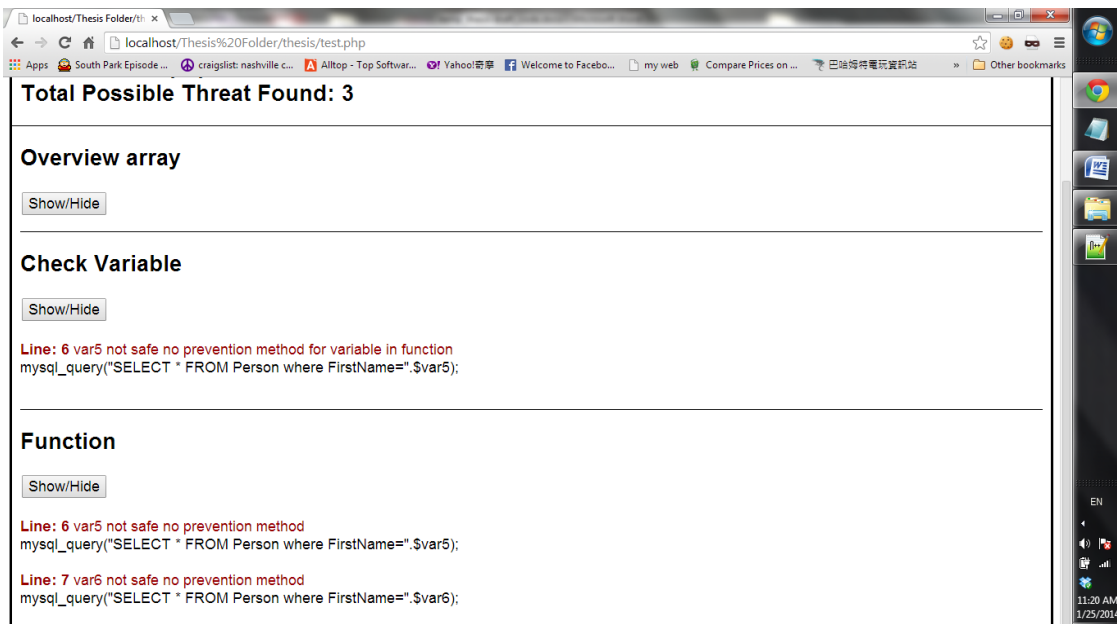


Figure 29. Application output of example code in Figure 28

4.1.4 *Escape method inside Function*

The fourth example changes the variable's value to another value inside the function, as shown in Figure 30. The variable `$var5`'s original value is the variable `$test3`, but it is assigned to the user input value "Fname." The unsafe variable `$var5` is escaped first and then assigned to the variable `$var6` which is now a safe value. The output of the code is given in Figure 31.

```
<?php
$test2 = ($_GET['Fname']);
$test3 = mysql_real_escape_string($_GET['Fname']);
abc($test2,$test3);
function abc($var5,$var6) {
$var5=$_GET['Fname'];
mysql_query("SELECT * FROM Person where FirstName=".$var5);
$var6=mysql_real_escape_string($var5);
mysql_query("SELECT * FROM Person where FirstName=".$var6);
}
?>
```

Figure 30. MYSQL Example of escape variables within function

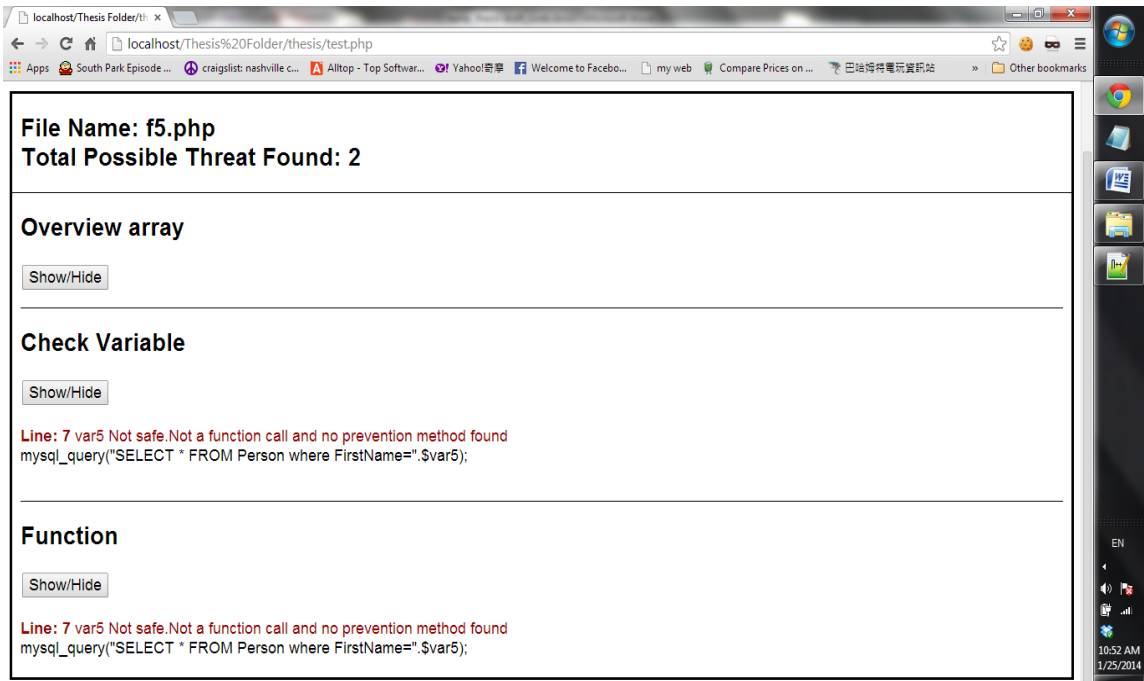


Figure 31. Application output of example code in Figure 30

4.1.5 Branches

The last example for MYSQL uses conditional statements, if and else. Since the application will never know which condition is correct, this application considers all the possible paths that could happen. Therefore, this application validates function `abc` in the if statement and `def` in the else statement. The if condition is taken from the fourth example, and else condition is taken from the third example. The result is the combination of both examples, except the line values changed slightly.

<?php

```

$test2 = ($_GET['Fname']);
$test3 = mysql_real_escape_string($_GET['Fname']);
if($test2==1)
abc($test2,$test3);
else
def($test2,$test3);
function abc($var5,$var6){
$var5=$_GET['Fname'];
mysql_query("SELECT * FROM Person where FirstName=".$var5);
$var6=mysql_real_escape_string($var5);
mysql_query("SELECT * FROM Person where FirstName=".$var6);
}
function def($var5,$var6){
mysql_query("SELECT * FROM Person where FirstName=".$var5);
mysql_query("SELECT * FROM Person where FirstName=".$var6);
}
?>

```

Figure 32. MYSQL Example of condition statement

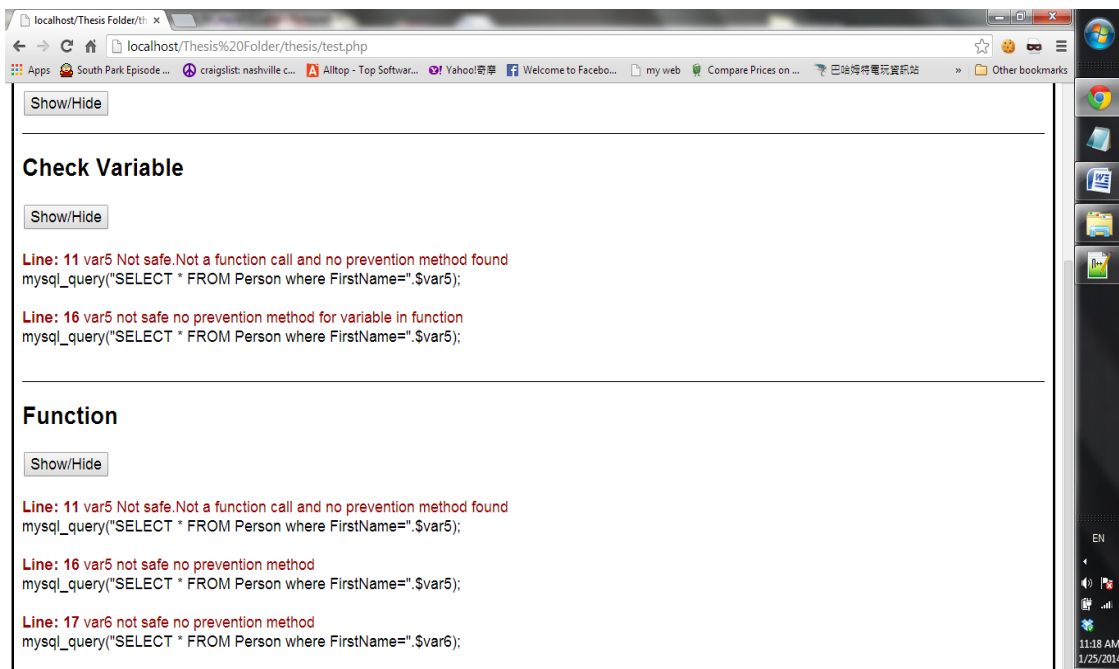


Figure 33. Application output of example code in Figure 32

4.2 MYSQLI Test Cases

4.2.1 *User Inputs to SQL Query Directly*

The first example is the basic usages of MYSQLI. The MYSQLI database extension provides two interfaces to access database: `prepare` and `query` methods of class `mysqli`, or a global function `mysqli_query`. All the function calls in the code of Figure 34 use the variable `$var1` to construct SQL queries. This variable is not escaped by any prevention method, so it is an unsafe value. Hence, all the function calls that use the variable `$var1` are vulnerable to SQL injection attack, as shown in Figure 35.

```
<?php
$var1=$_POST['uid'];
$abc = "INSERT INTO fo_table (Tag_num, FO_num, Eng_notes,
Type) VALUES ('$var1')";
$mysqli->query($abc);
$sth = $dbh->query("SELECT name, colour, calories FROM
fruit WHERE calories < $var1");
$test= $mysqli->prepare($abc);
mysqli_query($abc);
?>
```

Figure 34. MYSQLI Example of basic usage

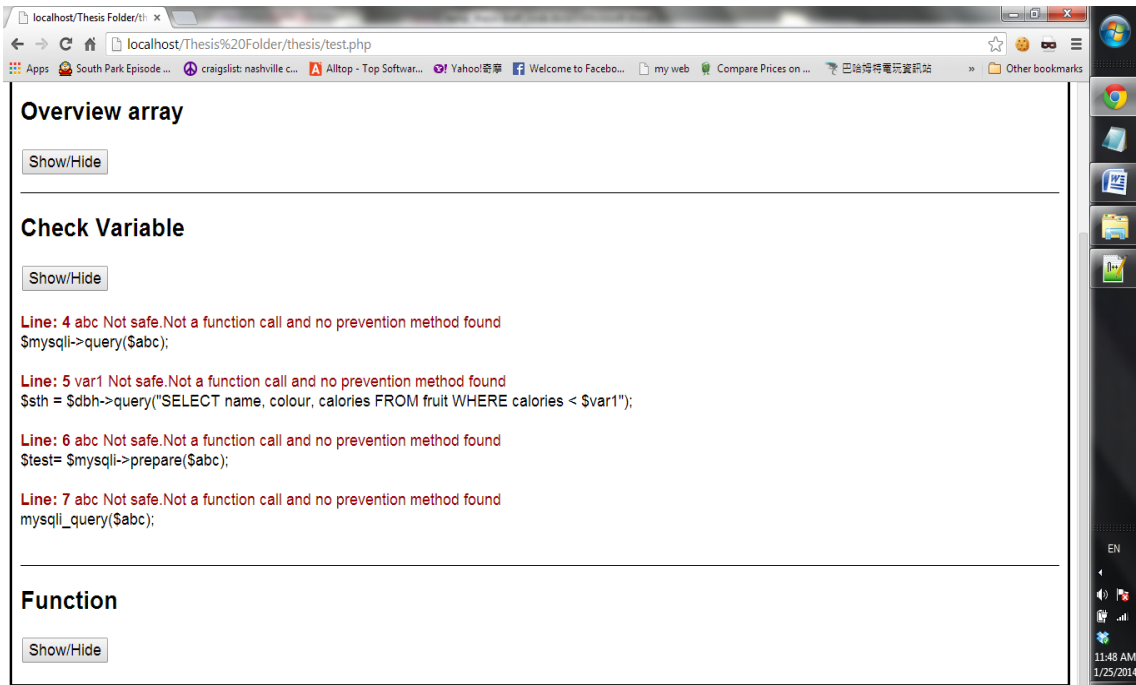


Figure 35. Application output of example code in Figure 34

4.2.2 *Escape Method*

In the second example given in Figure 36, the code is similar to the previous example. The only difference is that a prevention method is applied to the variable `$var1` to escape special characters in the variable so that SQL injection is prevented. Therefore, all function calls are safe in this example as shown in Figure 37.

```
<?php
$var1=mysql_real_escape_string($_POST['uid']);
$abc = "INSERT INTO fo_table (Tag_num, FO_num, Eng_notes,
Type) VALUES ('$var1')";
```

```

$mysqli->query($abc);
$sth = $dbh->query("SELECT name, colour, calories FROM
fruit WHERE calories < $var1");
$test= $mysqli->prepare($abc);
mysqli_query($abc);
?>

```

Figure 36. MYSQLI Example of escape method

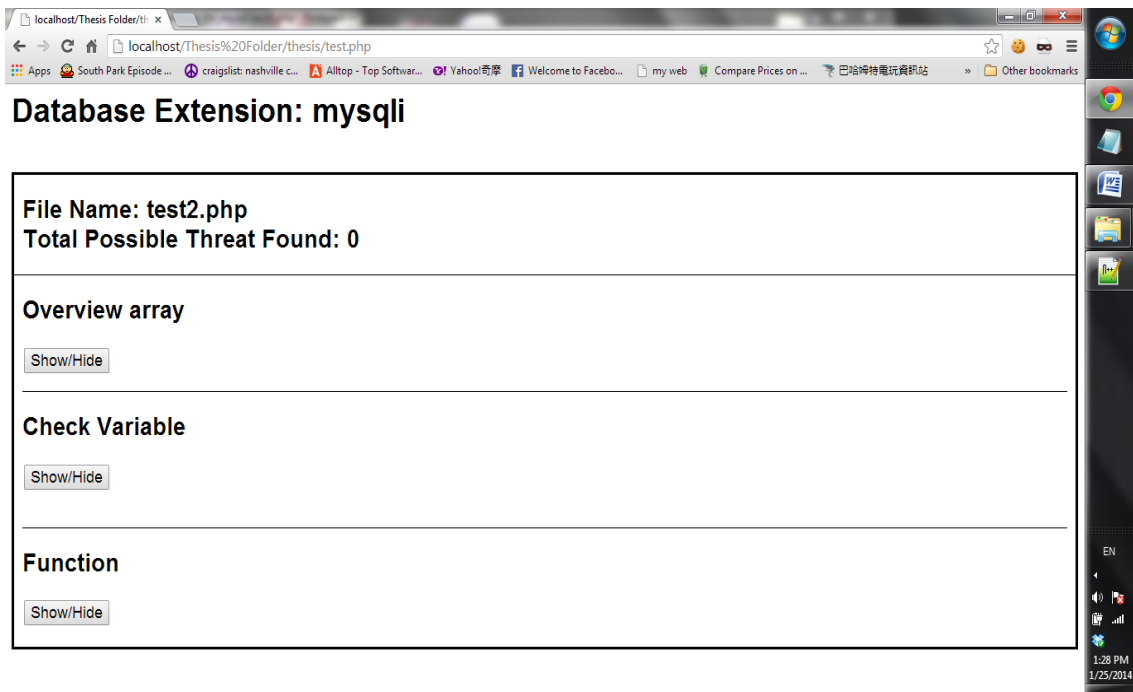


Figure 37. Application output of example code in Figure 36

4.2.3 Access Database Within Functions

In the third example, database access function is invoked within a user-defined function. One function parameter is used to construct SQL query. Therefore, if the function is invoked and the function argument comes from user input, it causes SQL injection vulnerability. As shown in Figure 38, both parameters of the function abc are

used to construct queries. When the function is invoked in the example, the variable `$var5`'s root value is variable `$test2` which is assigned to an unsafe value. The variable `$var6`'s root value is a safe value, the variable `$test3`. Therefore, the SQL execution statement with `$var5` is vulnerable to an SQL injection attack because the variable `$var5` is unsafe. Furthermore, if one considers the function alone, then both SQL statements will not be safe, because the variables are not validating inside the function. Hence, if the function is used somewhere else in the code, it is possibly vulnerable to SQL injection attack. The application output of this example is given in Figure 39.

```
<?php
$test2 = ($_GET['Fname']);
$test3 = mysql_real_escape_string($_GET['Fname']);
abc($test2,$test3);
function abc($var5,$var6){
$sth = $dbh->query("SELECT name, colour, calories FROM
fruit WHERE calories < $var5");
$sth2 = $dbh->query("SELECT name, colour, calories FROM
fruit WHERE calories < $var6");
}
?>
```

Figure 38. MYSQLI Example of database access within function

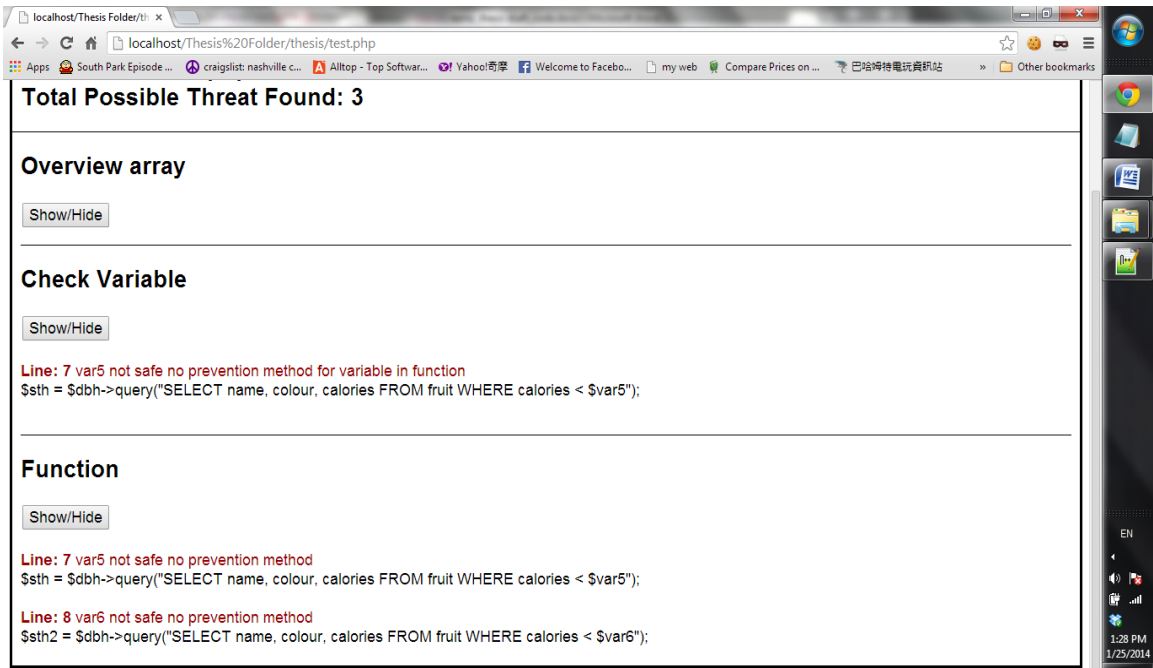


Figure 39. Application output of example code in Figure 38

4.2.4 Access Database Within Function—Parameter Contributes Indirectly

The fourth example is similar to the previous case except that function parameters are used indirectly to construct SQL queries. As shown in Figure 40, database is accessed within the function `abc`. Within the function, new local variables `$var7` and `$var8` are introduced to construct SQL queries. The values of these two new local variables are dependent on function parameters, respectively. The application should be able to detect SQL injection vulnerability: the query constructed from variable `$var7`, since it uses the value of `$var5`, which is actually the value of the unsafe variable `$test2`. The application output is given in Figure 41.

```
<?php
$test2 = ( $_GET[ 'Fname' ] );
```

```

$test3 = mysql_real_escape_string($_GET['Fname']);
abc($test2,$test3);
function abc($var5, $var6){
$var7=$var5;
$sth = $dbh->query("SELECT name, colour, calories FROM
fruit WHERE calories < $var7");
$var8=$var6;
$sth2 = $dbh->query("SELECT name, colour, calories FROM
fruit WHERE calories < $var8");
}
?>

```

Figure 40. MYSQLI Example of indirect database access within function

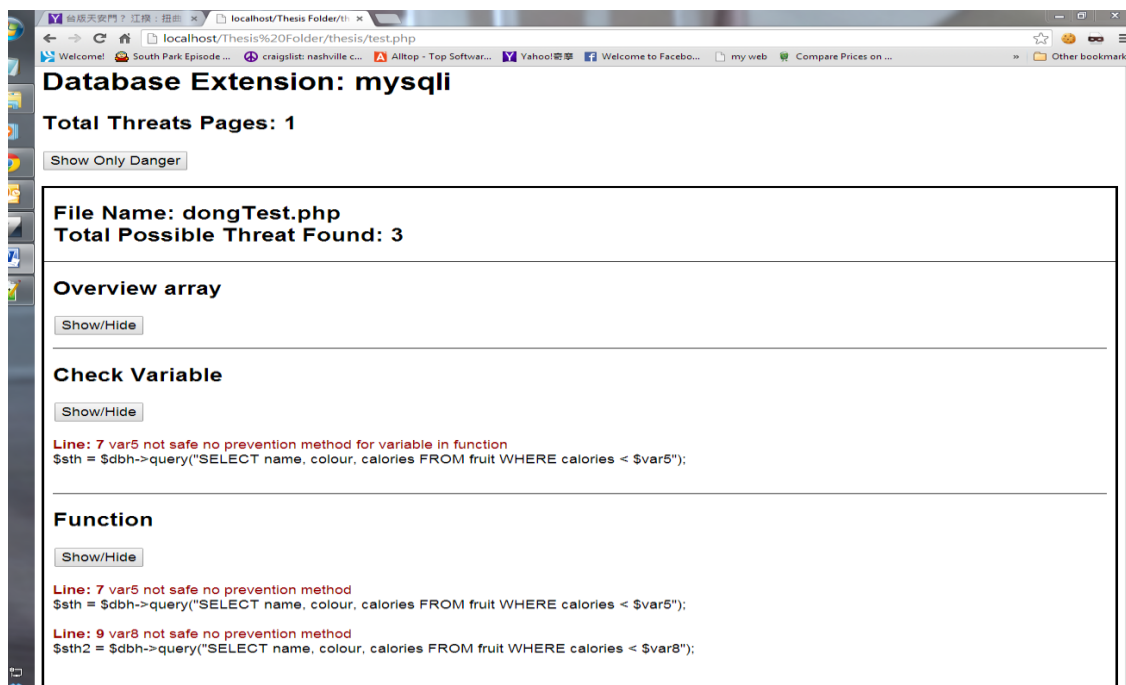


Figure 41. Application output of example code in Figure 40

4.2.5 Multiple Value Source

The last example demonstrates the situation that the variable used to construct query depends on two or more value sources. For example, in Figure 42, the variable

`$var1` is used to construct SQL query, and its value is a concatenation of two variables `$test2` and `$test3`. Since one value resource, i.e. `$test2`, is unsafe, the application should return a warning for the statement but only for the unsafe variable. The output is given in Figure 43.

```
<?php
$test2 = ($_GET['Fname']);
$test3 = mysql_real_escape_string($_GET['Fname']);
$var1=$test2." ".$test3;
$sth2 = $dbh->query("SELECT name, colour, calories FROM
fruit WHERE calories < $var1");
?>
```

Figure 42. MYSQLI example of multiple value sources

localhost/Thesis Folder/thi x
localhost/Thesis%20Folder/thesis/test.php
Welcome! South Park Episode ... craigslist: nashville c... Alltop - Top Softwar... Yahoo! 奇摩 Welcome to Facebo... my web Compare Prices on ... Other bookmarks

Database Extension: mysql

Total Threats Pages: 0

Show Only Danger

File Name: dongTest.php
Total Possible Threat Found: 0

Overview array
Show/Hide

Check Variable
Show/Hide
Line: 5 var1 not safe no prevention method for variable in function

Function
Show/Hide

Figure 43. Application output of example code in Figure 42

CHAPTER V

CASE STUDY

1. Introduction

To evaluate the effectiveness of the tool, a case study has been conducted. The PHP web applications were collected from CSCI 4560/5560: Database Management System class taught in Middle Tennessee State University at Fall 2013.

In the class, a real-world database project was provided by local office of Schneider Electric, a France-based multinational corporation. The goal of the project is to rewrite an existing Access application using PHP and MySQL. Students in the class are divided into groups of three students with a total of 10 groups. Each group is required to develop their own applications and give the presentation at the end of the semester. The company will choose the one they feel more comfortable. The rest of the section gives a brief introduction of the project. Detailed specification of the project can be found at Appendix B.

The major functionality of the project is to maintain the database of products (called TAG in the company), and manage users. The users of the application will be divided into groups. Each user group has different access privileges on the product database. Four predefined groups are provided: tag members, OE, user, and administrator. Only users of the tag members group can insert, revise, and search TAGs. Those of the user group can only view the TAGs one by one (excluding Price information which is only accessible by tag members and OE users). Only administrators can view and add groups in addition to adding and removing users to or from groups. A user may be in multiple groups. All other unassigned viewers will see only a blank page or one

showing “Contact TAG Group for pricing.” This restriction requires non-approved users to contact the TAG group to verify proper application of TAGs.

Besides the viewing and editing of TAGs, the project has other functionalities like security, print, and search. The security feature records successful logins on a table. The recorded information includes time, date, machine name, IP address, username, and other necessary information. Only administrative users can view the login log. After a user logs in successfully, the system remembers the user for the next ninety days so that no password is required. The print feature is on all viewable pages and allows end users to print a clean page that excludes unnecessary filler (colors, links, buttons, etc.) but includes company logo, time, and date of printed page, and username as “Printed by”. The search feature is available for tag member group users and allows them to search TAGs using one or more of the following searching criteria:

- Tag Number
- Rev#
- Date
- Tag description contains a substring
- Sub-category
- HVL product type
- HVL/CC product type
- Metal clad product type
- MVMCC product type
- Tag note contains a substring
- Install cost

- Price note contains a substring
- User who created/updated the revision

An option is provided to specify the target of the search with the latest revision of TAGs or obsolete revision of TAGs. Whenever a change is made to the database, a window will pop up to confirm the change. The default values for Price expire, Labor/Engineering are three months and one hour, respectively.

2. Data Collection

MySQL database is required to store data, but there is no restriction on the database extension that can be used to access database. As shown in Table 1, three different database extensions were used among student projects: MYSQL, MYSQLI, and PDO. It is clear that MYSQL and MYSQLI are more popular than PDO since there are four groups using MYSQL and MYSQLI respectively, but there is only one group using PDO.

Table 1. Languages used

| Group | MYSQL | MYSQLI | PDO |
|----------|-------|--------|-----|
| Group 1 | 1 | 0 | 0 |
| Group 2 | 1 | 0 | 0 |
| Group 3 | 0 | 1 | 0 |
| Group 4 | 0 | 1 | 0 |
| Group 5 | 0 | 1 | 0 |
| Group 6 | 0 | 1 | 0 |
| Group 7 | 1 | 0 | 0 |
| Group 8 | 0 | 1 | 0 |
| Group 9 | 1 | 0 | 1 |
| Group 10 | 1 | 0 | 0 |
| Total | 5 | 5 | 1 |

Table 2 lists the number of different types of pages each group created. Since our application can only check PHP files, all non-PHP files such as HTML, CSS, or JS files are omitted because those files are not able or not likely to interact with databases. Several groups use PHP files as HTML files, so some results will not return anything, since the parser will not parse HTML code.

Table 2. Page extensions

| Group | PHP | HTML | JS | CSS |
|----------|-----|------|----|-----|
| Group 1 | 29 | 0 | 1 | 4 |
| Group 2 | 21 | 0 | 0 | 3 |
| Group 3 | 26 | 0 | 0 | 8 |
| Group 4 | 24 | 0 | 0 | 1 |
| Group 5 | 70 | 5 | 0 | 0 |
| Group 6 | 18 | 0 | 2 | 8 |
| Group 7 | 33 | 0 | 2 | 2 |
| Group 8 | 23 | 13 | 0 | 0 |
| Group 9 | 38 | 0 | 0 | 2 |
| Group 10 | 73 | 0 | 8 | 4 |

The result of each group is showing in table 3. Group five is omitted because the group's code has syntax errors. The first four groups used MYSQL, the second four groups used MYSQLI and the last group used combination of PDO and MYSQL.

Table 3. Result of Student Projects

| Group # | 1 | 2 | 7 | 10 | 3 | 4 | 6 | 8 | 9 | 9
MYSQL |
|-----------------------|-----|-----|-----|-----|-----|-----|-----|-----|----|------------|
| Vulnerable Page | 16 | 12 | 8 | 32 | 6 | 5 | 7 | 5 | 0 | 3 |
| Vulnerabilities Found | 19 | 24 | 9 | 50 | 59 | 13 | 12 | 8 | 0 | 10 |
| Vulnerable Page Ratio | 55% | 57% | 24% | 44% | 23% | 21% | 39% | 22% | 0% | 8% |

Three groups were selected to take a closer look: group one, group three, and group nine. Group one was selected because it is vulnerable to the simple SQL injection attack on the login page. It will be valuable to check if this application detects the vulnerability. Group three was selected because it uses MYSQLI database extension, and is working properly. Group nine was selected because it is the only project using PDO database extension.

2.1 Detail Analysis of Each Group

The most vulnerable pages among the groups are the login page, search page, modify page, and insert page. The reason for these vulnerabilities is that those pages are required to use SQL statements. SELECT for login pages and search pages, UPDATE for modify pages, and INSERT for insert pages.

2.1.1 Group One

The main problem for group one is that most variables are used without validation. For example, the application found two vulnerable variables in group one's login page. In line twenty-nine, group one uses the variable query in the SQL execution

function. In line forty-five, the variable `tempID` is used directly in the SQL execution function. The variable `query` is a string of SQL statements concatenate with variable `tempID`, a user input value. In the code, group one did not validate variable `tempID` nor variable `query`, so the application returns warnings for these two variables.

```
<?php
...
//Here is the code to get user input
$tempID = "" . $_POST['user_ID'] . "";
...
$query = "SELECT * FROM User Where Id = $tempID";
...
//Here is the code to execute database access function
$result = mysql_query($query) or die("Query failed : " .
mysql_error());
...
?>
```

Figure 44. Group One login page vulnerable code

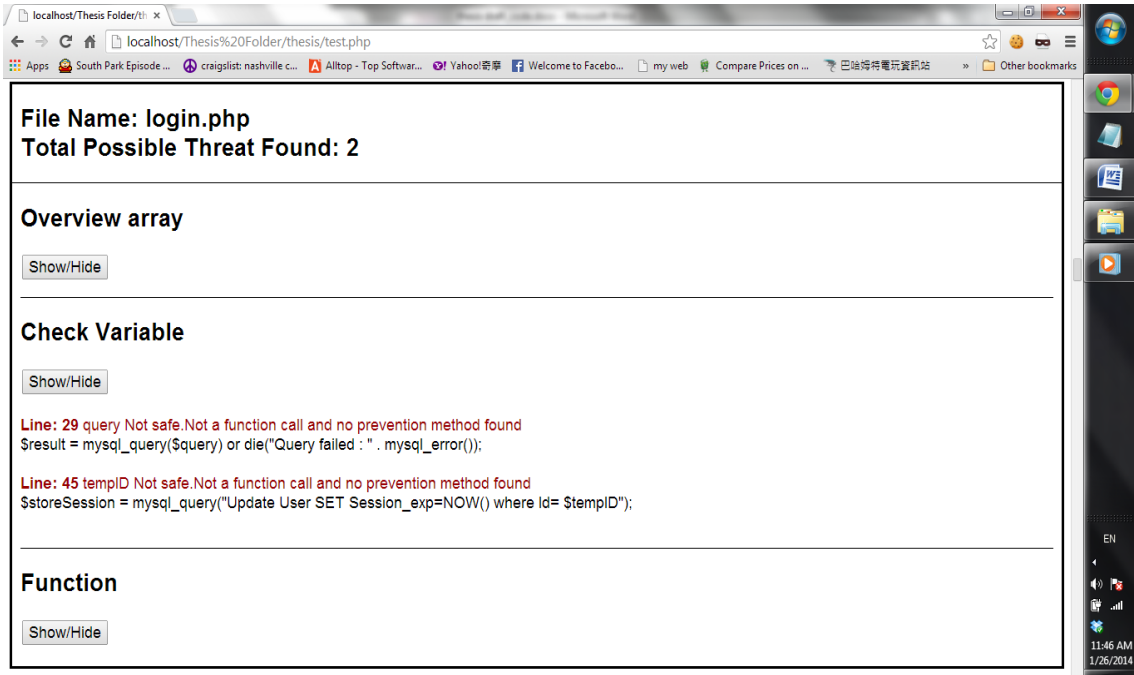


Figure 45. Application output of group one login page

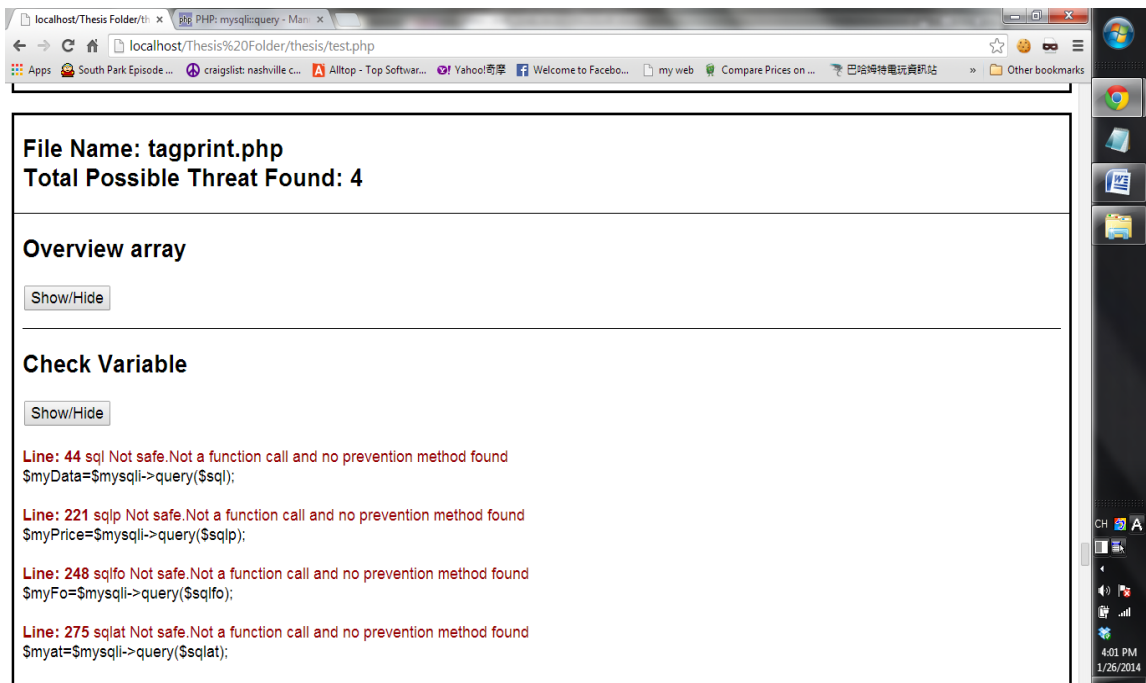
2.1.2 Group Three

Group three used MYSQLI, which is a newer and more advanced version of the MYSQL database extension. It is safer than MYSQL, if the users utilize it properly. The groups used prepare and query statements to execute SQL statements. The application would be safer if the group used prepare statement across the web pages and used SQL escape methods for the variables or SQL statements.

In the tagprint page, the group used the query function instead of the prepare function. All the variables used in the query function were not validated. Therefore, the application returned warnings to all the SQL execution functions in the page.

```
<?php
...
//Here is the code to get user input
$rev = ($_GET['rev'] == "") ? getRevMax($Num) :
test_input($_GET['rev']);
...
//Here is the code to execute database access function
$sql="SELECT * FROM TAG WHERE Tag_number = $Num and
Rev_number = $rev";
$myData=$mysqli->query($sql);
...
?>
```

Figure 46. Group three tagprint page vulnerable code



The screenshot shows a web browser window displaying the output of a security analysis tool. The browser's address bar shows the URL `localhost/Thesis%20Folder/thesis/test.php`. The tool's output is as follows:

File Name: tagprint.php
Total Possible Threat Found: 4

Overview array

Check Variable

Line: 44 sql Not safe. Not a function call and no prevention method found
`$myData=$mysqli->query($sql);`

Line: 221 sqlp Not safe. Not a function call and no prevention method found
`$myPrice=$mysqli->query($sqlp);`

Line: 248 sqlfo Not safe. Not a function call and no prevention method found
`$myFo=$mysqli->query($sqlfo);`

Line: 275 sqlat Not safe. Not a function call and no prevention method found
`$myat=$mysqli->query($sqlat);`

Figure 47. Application output of group three tag print

In some pages, the group used the prepare statement incorrectly. For example, in line forty-seven of “inserttag” page, the following prepare statement is used:

```
$stmt = $mysqli->prepare("INSERT INTO TAG (Tag_number,  
Rev_number, Current) VALUES ('$_POST[tag_number]',  
'0','1')");
```

Figure 48. Incorrect used of prepare statement.

This prepare statement is dangerous and vulnerable because the SQL statement has a user input without validation. Users should instead use the placeholder for the value, like the following:

```
$stmt = $mysqli->prepare("INSERT INTO TAG (Tag_number,  
Rev_number, Current) VALUES (?, '0','1')");  
$stmt->bind_param("s", $_POST[tag_number]);
```

Figure 49. Correct way of using prepares statement.

The prepare function will not be safe if the developers use the user input directly in the statement. Group three did not validate user input nor use the place holder in the function. Therefore, the application returns the lines with warning.

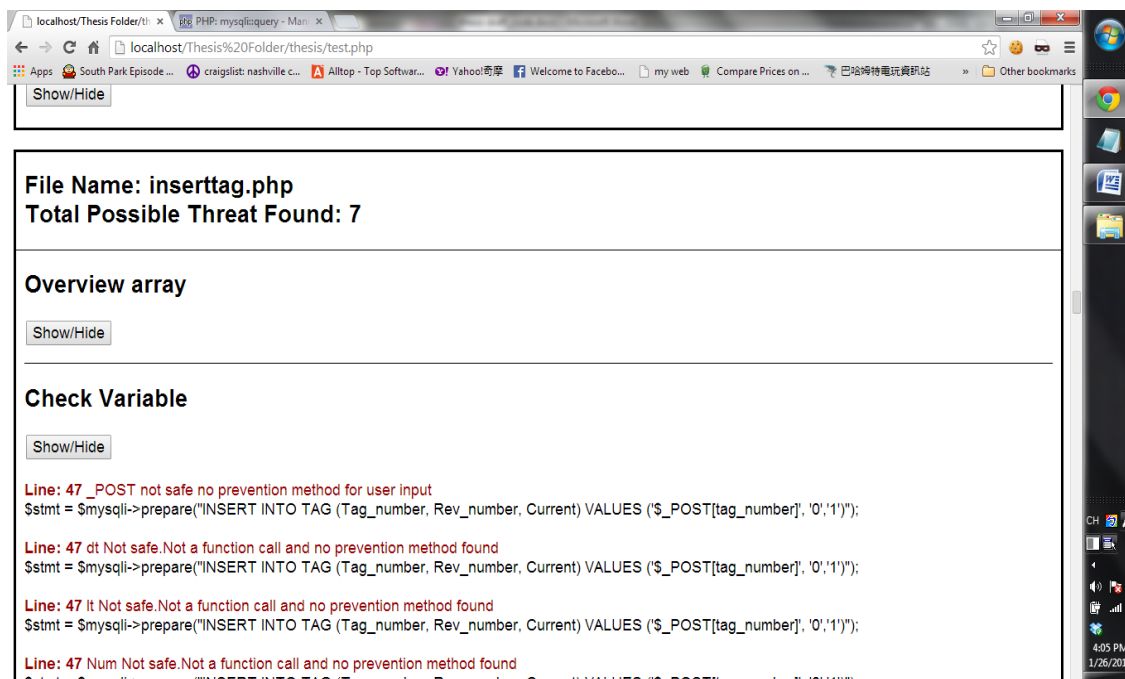


Figure 50. Application output of group three insert tag page

2.1.3 Group Nine

Group nine is the only group using PDO extension. PDO is similar to MYSQLI. Both are object-oriented and safer than MYSQL. In some pages, the group also used MYSQL to access database. The application is not able to detect two extensions at the same time, so the results are split into PDO and MYSQL.

The pages with PDO return zero vulnerability. All the pages with PDO use prepare statements with placeholders. Therefore, all the SQL statements are safe. However, the pages with MYSQL have some vulnerability. In most cases, the reason is that the group did not validate the variables before using them in the SQL execution statements.

The group also used pages that contain only functions. These functions can be called by other PHP pages, and these functions did not validate the parameters before inserting the parameters variable in the MYSQL SQL execution statements. Therefore, the application returns several warnings to those functions.

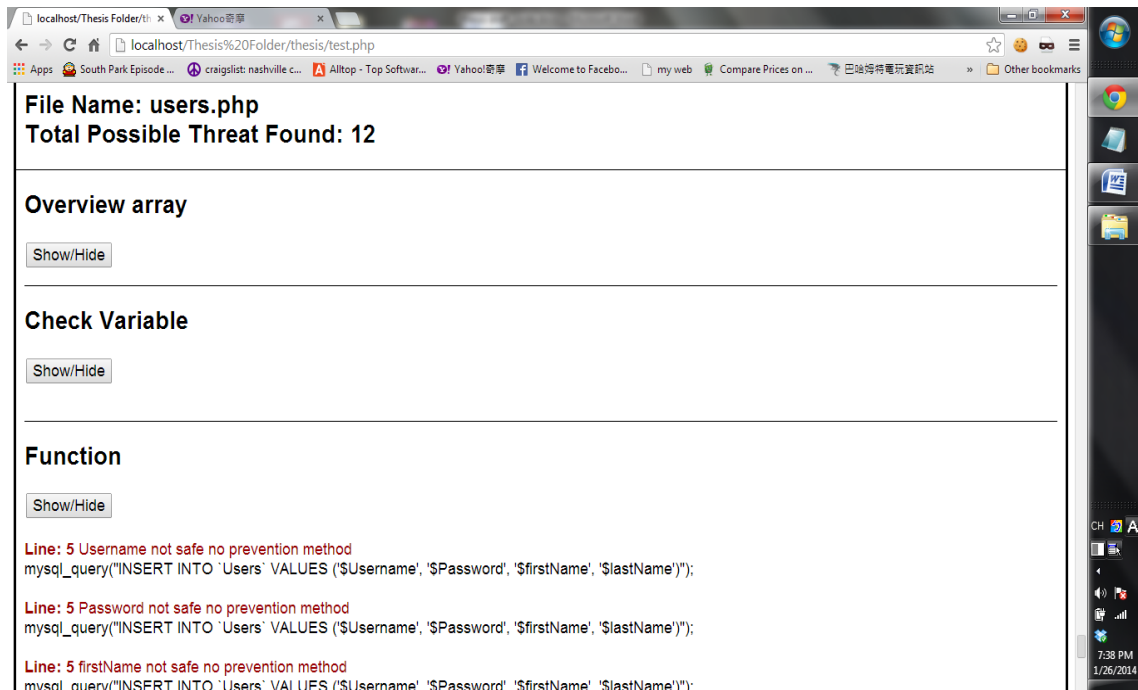


Figure 51. Application output of group nine user page

3. Summary

The common mistakes students made are that user input values are not validated before executing SQL statements. This mistake is happened to almost every group in this case study. This mistake can be easy prevent by using `mysql_real_escape_string()`.

Another mistake student made is that the prepare statement of MYSQLI and PDO is not used properly. In group three example, instead of using placeholder and parameter binding approach to construct SQL query, user inputs is used directly to create the query string in `prepare` statement. This creates SQL injection vulnerabilities in the code.

This case study shows that most students are not aware of security issues when developing software. As security becomes an important feature in today's software, the training on security is required for students. The application can help students quickly and precisely detect SQL injection vulnerabilities in their programs.

CHAPTER VI

CONCLUSION AND FUTURE WORKS

1. Conclusion

It takes time and knowledge to implement security requirement correctly. Some developers would neglect the security components due to variant reasons such as lack of experience, deadline pressure, and budget restriction. To help developers detect SQL injection vulnerability in their PHP applications, we developed a tool for this purpose. This tool explores the PHP code to detect all SQL injection vulnerable code. The tool supports all different types of PHP database extensions. The developed tool meets all the expectations. It is able to find potential SQL injection threats in PHP code. All the run times are under a second, which is much faster than most of the vulnerability-detecting system online. The application is reliable since no false positive is found in the results.

Through our case study on student projects, we found out that students typically don't pay attention to security requirements. In the case study, three of the nine projects had potential to be victims of SQL injection. The main reason these students' projects are vulnerable is that the students did not validate the data or use the function correctly. Also, many students are still using the MYSQL extension, which is not recommended by the PHP user group. The PHP official site states, "This extension is deprecated as of PHP 5.5.0, and will be removed in the future. Instead, the MYSQLI or PDO_MYSQL extension should be used." [11] MYSQL does not provide the prepare statement like MYSQLI and PDO, making MYSQL more vulnerable to injection-type attacks like SQL injection, JavaScript injection, and XSS.

Schools and teachers have a lot of influence on students' coding styles. Students should be encouraged to write safer code. Additionally, teachers should teach the importance of picking the right API's and extensions. In our case study, students should be encouraged to use MYSQLI and PDO instead of MYSQL. Students also need to know how to use the prepare statement properly. In the group three example, the group used the prepare statement, but they used it improperly causing the code to be vulnerable to attacks. In addition, teachers should encourage students to implement basic security features in projects by offering extra credits.

2. Future Works

Some additional features could be added to application. Since JavaScript injection attacks are similar to SQL injection attacks, it would be beneficial to implement the detection mechanism to the application. The application could add another, or modify the existing, configuration file to implement this feature in the application.

A problem with this application is that the application is not able to detect the variables or functions if the developers use include statements in the PHP code. The application is not able to trace back the variables or function in the include files. This decreases the reliability of the application. However, to implement this feature, a larger scale of detecting and analyzing method is required. The run time will increase exponentially if the developers have a lot of include files. To complement this, the application implemented the function only result, which could be an alternative of this feature.

Another problem is that the application is not able to validate the return values.

An example code is showing below:

```
<?php
$var5=$_GET['Fname'];
mysql_query("SELECT * FROM Person where
FirstName=".def($var5));
function def($var7){
return $var7;
}
?>
```

Figure 52. Return value

In this case, this application should return a warning, but in current version it will not warn the user. Another function will be needed to parse the return value into our current array. Once the return value is in the array, the application will need to trace back and validate the possible values of the return value.

BIBLIOGRAPHY

- [1] Acunetix (2014) *Acunetix WVS* Available at: <http://www.acunetix.com/> (Accessed: 3rd march 2014)
- [2] BEYONDTRUST (2014). *Retina Web Security Scanner*. Available at: <http://www.beyondtrust.com/> (Accessed: 3rd march 2014)
- [3] Ehmer Khan, Mohd (2011) *Different Approaches to White Box Testing Technique for Finding Errors*, International Journal of Software Engineering and Its Applications, 5(3), pp. 1-6.
- [4] Etienne Janot, Pavol Zavarisky (2008) *Preventing SQL Injections in Online Applications: Study, Recommendations and Java Solution Prototype Based on the SQL DOM*. Available at: <https://www.owasp.org/images/5/57/OWASP-AppSecEU08-Janot.pdf> (Accessed: 3rd march 2013)
- [5] IBM (2014). *Security AppScan*. Available at: <http://www-03.ibm.com/software/products/en/appscan/> (Accessed: 3rd march 2014)
- [6] Justin Clarke (2012) *SQL injection attacks and defense, 2nd*. Waltham, MA: Elsevier.
- [7] Linda Davis, et al (2011) *2011 top cyber security risks report*: Hewlett-Packard Development Company. Available at: <http://www.hpenterprisesecurity.com/collateral/report/2011FullYearCyberSecurityRisksReport.pdf> (Accessed: 3rd march 2013)
- [8] Mirza Mohammed, Akram Baig (2012) *Security Vulnerabilities in PHP Application* San Diego State University. Available at: http://sdsudspace.calstate.edu/bitstream/handle/10211.10/2027/Baig_Mizra.pdf?sequence=1 (3/18/13).
- [9] Nicolas Surribas (2013). *Wapiti*. Available at: <http://wapiti.sourceforge.net/> (Accessed: 3rd march 2014)
- [10] Nikic (12/01/2013). *Documentation for version 0.9.x* GitHub.com/nikic. Available at: <https://github.com/nikic/PHP-Parser/tree/0.9/doc>. (Accessed: 3rd march 2013)
- [11] The PHP Group (2013) *Choosing an API*, Available at: <http://www.php.net/manual/en/mysqlinfo.api.choosing.php> (Accessed: 1st march 2013).

- [12] Romain Gaucher (2013). Grabber. Available at:
<http://rgaucher.info/beta/grabber/> (Accessed: 3rd march 2014)
- [13] SensePost (2008). *Wikto*. Available at:
<http://research.sensepost.com/tools/web/wikto> (Accessed: 3rd march 2014)
- [14] Sergey Gordeychik, et al *Web application vulnerability statistics for 2010-2011* (2012) Positive Technologies. Available at:
<http://www.ptsecurity.com/download/statistics.pdf> (Accessed: 3rd march 2013)
- [15] *Sony faces legal action over attack on PlayStation network*. (4/29/2011). BBC News. Available at: <http://www.bbc.co.uk/news/technology-13192359>.
(Accessed: 3rd march 2013)
- [16] *Top 10 2007* OWASP Foundation (2007). Available at:
https://www.owasp.org/index.php/Top_10_2007 (Accessed: 3rd march 2014)

APPENDICES

APPENDIX A

SOURCE CODE

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="style.css">
<script src="http://code.jquery.com/jquery-
1.9.1.js"></script>

<script type="text/javascript" src="myScript.js"></script>
</head>
<body>
<?php
error_reporting(E_ERROR);
require '/PHP-Parser/lib/bootstrap.php';
$parser          = new PHPParser_Parser(new PHPParser_Lexer);
$traverser       = new PHPParser_NodeTraverser;
$prettyPrinter   = new PHPParser_PrettyPrinter_Default;
$fileSize        = count($_FILES['userfile']['name']);
$testFiles       = array();
$allFiles=checkUpload($_FILES['userfile']['tmp_name'],
$_FILES['userfile']['name']);
$zipFiles=forZipFile($allFiles[0]);

//added 2/3/2014 handle zip files
foreach ($zipFiles[0] as $zipFile){
    $filename = substr(strrchr($zipFile, "/"), 1);

    if (!array_key_exists($filename,$testFiles)){

        $testFiles[$filename]=file_get_contents($zipFile);
        $myFile[substr($filename,0,-4)]=file($zipFile);
    }
    else{
        $testFiles[$filename."-
copy".rand()]=file_get_contents($zipFile);
        $myFile[substr($filename."-copy".rand(),0,-
4)]=file($zipFile);
    }
}

foreach ($allFiles[1] as $key=>$tempFile){
```

```

        if (!array_key_exists($filename,$testFiles)){
            $testFiles[$key]=file_get_contents($tempFile);
            $myFile[substr($key,0,-4)]=file($tempFile);
        }
        else{
            $testFiles[$key."-
copy".rand()]=file_get_contents($tempFile);
            $myFile[substr($key,0,-4)]=file($tempFile);
        }
    }

$count=0;
$functionName = "";
$userDatabase=$_POST["userDatabase"]; //user select
database
//read the config file
$config = file('config.txt', FILE_IGNORE_NEW_LINES |
FILE_SKIP_EMPTY_LINES);
$dataBaseArray= array();
//database sql execute method
$sqlExeMethod = array();
$key=array_search('@'.$userDatabase ,$config);
$size= sizeof($config);

for ($i=$key+1;$i<$size;$i++){
    if (!strstr($config[$i],"@")){//new database extension
start
        $sqlExeMethod[]=strtoupper($config[$i]);
    }
    else
        break;
}
//end of getting execute method  $sqlExeMethod array
//php user input method
$key=array_search('#safe',$config);
$userInput = array();
for ($i=1;$i<$key;$i++){
    $userInput[]=$config[$i];
}

$StopSafeKey=array_search('#database',$config);

for ($i=$key+1;$i<$StopSafeKey;$i++){
    $safeConfig[]=$config[$i];
}

```

```

}
//end of getting user input method
//object sql exe stmt like prepare or query 1 means yes
empty means no
$objectSQL=$_POST["objectSQL"]; //user select database
//end
//should we remove the 2nd parameter of sql stmt or not 1
mean yes empty means no
$onlyOne=$_POST["onlyOne"]; //user select database
//end
echo "<h1>Database Extension: $userDatabase </h1>";
echo "<h2>Total Threats Pages: <span
id='totalScore'>0</span></h2>";
echo "<button onclick=toggleThreatA('danger')>Show Only
Danger</button> </br></br>";

foreach ($testFiles as $testFile){
    $me = new myPar();
    $newStatement =newClass();
    $newStatement->restart();
    $myAn=NewAn();
    $myAn->destoryVul();
    if (stripos(array_keys($testFiles)[$count], 'php') !==
false){
        echo "<div
id='result'.array_keys($testFiles)[$count].'"
class='mainResult'>";
        echo "<div class='resultPage' id='overviewHeader' >";
        echo "<h2>File Name: ".array_keys($testFiles)[$count];
        $tempFileStringName=array_keys($testFiles)[$count];
        if (strpos($tempFileStringName, 'copy')==false){
            $tempFileName=
substr(array_keys($testFiles)[$count], 0, -4);
        }
        else{
            $tempFileName= str_replace(".php-",
", "", array_keys($testFiles)[$count]);
        }
        echo "</br>Total Possible Threat Found: <span
id='vulCount'.'. $tempFileName.'">0</span></h2>";
        echo "</div>";
        echo "<hr>";
        //gather data ----phase 1
        try {
            // parse

```

```

$stmts = $parser->parse($testFile);
// traverse
$stmts = $traverser->traverse($stmts);
$counStmt=0;
foreach ($stmts as $temp){
    $newStatement->SetobjectSQL($objectSQL);
    $newStatement->SetexSqlStmt($sqlExeMethod);

    $newStatement->breakStmts($temp);
}
}

catch (PHPParser_Error $e) {
    echo 'Parse Error: ', $e->getMessage();
}
if (strstr($DBname, '_')){
    $pos = strpos($DBname, "_");
    $DBname= substr($DBname,0,$pos);
}
//function params
echo "<div class='resultPage' >";
echo "<div class='heading'><h2>Overview array </h2>
<button
onclick=toggle('".$tempFileName."')>Show/Hide</button></div
>";
    echo "<div id=".$tempFileName."
style='display:none'>";
    echo "<h3 >this is variable Array: </h3>";
    print_r(array_filter($ifData));
    echo "</br>";
    //echo "This is ".$DBname." Databasae ".$</br>";
    echo "<h3>this is function Array: </h3>";
    $functionArray=$newStatement->getFunctionData();
    print_r($functionArray);
    echo "</br>";
    echo "</div></div>";
    //analysts data phase 2
    $myAn->SetobjectSQL($objectSQL);
    $myAn->SetexUserInput($userInput);
    $myAn->SetexSqlStmt($sqlExeMethod);
    $myAn->SetOnlyOne($onlyOne);
    $myAn->setFile($myFile);
    $myAn->SetSafeFunFromConfig($safeConfig);
    foreach ($newStatement->getFunctionData() as
$objKey=>$tester) {

```

```

        foreach($tester as $tester2){
            $result=$myAn->checkFun($tester2,$objKey);
//if the array has MYSQL_QUERY added it to the result array
            if(!empty($result)){
                $myAn->addCheckFun($objKey,$ifData);
//only test the function that is used
            }
        }
    }
    echo "<div class='resultPage' >";
    echo " <hr><div class='heading'><h2>Check Variable
</h2> <button
onclick=toggle('".$tempFileName."VS')>Show/Hide</button></d
iv></br>";
    echo "<div id='".$tempFileName."VS' >";
    $useFunction = $myAn->getDangerFun();
    //check inside the function in $dangerFun array
    foreach($functionArray as $key=>$checkThisArray){

        $myAn-
>getTarget2($checkThisArray,$key,null,$ifData);
    }
    $myAn->getTarget($ifData); // normal variable array

    echo "</div>";
    echo "</div>";
    echo "<div class='resultPage' >";
    echo " <hr><div class='heading'><h2>Function </h2>
<button
onclick=toggle('".$tempFileName."FS')>Show/Hide</button></d
iv></br>";
    echo "<div id='".$tempFileName."FS' >";
    foreach($functionArray as $key=>$checkThisArray){
        $myAn->getTarget($checkThisArray,$key,true);
    }
    echo "</div>";
    echo "</div>";
    $tempCount=$myAn->getVul();
    $tempFileStringName= array_keys($testFiles)[$count];

    $tempFileStringName=array_keys($testFiles)[$count];
    if (strpos($tempFileStringName,'copy')==false){
        $tempFileName=
substr(array_keys($testFiles)[$count],0,-4);
    }

```

```

        else{
            $tempFileName= str_replace(".php-
", "", array_keys($testFiles) [$count]);
        }
        echo "<script
type=text/javascript>javascript:jsfunction('".$tempFileName."', ".$temp
Count.", '".array_keys($testFiles) [$count]."');</script>";
        echo "</br></div> ";
        }
        $count++;
    }

function newClass(){
    $myNewClass = new myPar();
    return $myNewClass;
}

function newAn(){
    $myNewAn = new myAn();
    return $myNewAn;
}

#seperate the zip file and normal php files allfile array
index 0 is for zip file index 1 is normal file array
function checkUpload($uploads, $fileType){
    $counter =0;
    $normalCounter=0;
    $zipFile = array();
    $normalFile = array();
    foreach ($fileType as $upload){
        if (substr($upload, -3)=="php"){

            $normalFile[$fileType[$normalCounter]]=$uploads[$count
er];

                $normalCounter++;
            }
            elseif (substr($upload, -3)=="zip"){
                $zipFile[]=$uploads[$counter];
            }
            $counter++;
        }
    }
    $allFile[0]=$zipFile;
    $allFile[1]=$normalFile;
    return ($allFile);
}

```

```

function forZipFile($myZipFiles){
    $dir = array();
    foreach ($myZipFiles as $myZipFile){
        if(!empty($myZipFile))
            $dir[]=unzip($myZipFile, false, true, true);
    }
    return $dir;
}
function unzip($src_file, $dest_dir=false,
$create_zip_name_dir=true, $overwrite=true)
{
    $zipFiles=array();
    if ($zip = zip_open($src_file))
    {
        if ($zip)
        {
            $splitter = ($create_zip_name_dir === true) ? "." :
"/";
            if ($dest_dir === false) $dest_dir =
substr($src_file, 0, strrpos($src_file, $splitter))."/";
            // Create the directories to the destination dir if
they don't already exist
            create_dirs($dest_dir);
            // For every file in the zip-packet
            while ($zip_entry = zip_read($zip))
            {
                // Now we're going to create the directories in the
destination directories
                // If the file is not in the root dir
                $pos_last_slash =
strrpos(zip_entry_name($zip_entry), "/");
                if ($pos_last_slash !== false)
                {
                    // Create the directory where the zip-entry
should be saved (with a "/" at the end)

                    create_dirs($dest_dir.substr(zip_entry_name($zip_entry), 0,
$pos_last_slash+1));
                }

                // Open the entry
                if (zip_entry_open($zip,$zip_entry,"r"))
                {
                    // The name of the file to save on the disk

```

```

        $file_name =
$dest_dir.zip_entry_name($zip_entry);

        // Check if the files should be overwritten or
not
        if ($overwrite === true || $overwrite === false
&& !is_file($file_name))
        {
            // Get the content of the zip entry
            $fstream = zip_entry_read($zip_entry,
zip_entry_filesize($zip_entry));
            file_put_contents($file_name, $fstream );
            // Set the rights
            chmod($file_name, 0777);
            //echo "save: ".$file_name."<br />";
            $zipFiles[]=$file_name;
        }
        // Close the entry
        zip_entry_close($zip_entry);
    }
    // Close the zip-file
    zip_close($zip);
}
}
else
{
    return false;
}
return $zipFiles;
}

```

```

//This function creates recursive directories
//if it doesn't already exist @param String
//The path that should be created @return void

```

```

function create_dirs($path)
{
    if (!is_dir($path))
    {
        $directory_path = "";
        $directories = explode("/", $path);
        array_pop($directories);

        foreach($directories as $directory)

```



```

    {
        $directory_path .= $directory."/";
        if (!is_dir($directory_path))
        {
            mkdir($directory_path);
            chmod($directory_path, 0777);
        }
    }
}
}

```

```

class myPar{
private $check=1 ; //0 is true 1 is false
public $array=array(); // data for the hashtable
private $funData=array(); // data include all the info
private $subData=array();
private $keyStorage= array(); //check if key is already
exist
private $ifData = array(); //store if stmt variable and
values;
private $ifValue = array(); // store the value of if
statment
private $functionArray = array(); //store functional name
and its info
private $noNameCount;
private $funParams=array(); //store the function param
private $objectSQL; //determine if it is a db using prepare
or query stmts

public function restart(){
global $array;
global $funData;
global $subData;
global $keyStorage;
global $ifData;
global $ifValue;
global $functionArray;
global $noNameCount;
global $funParams;
$array=array();
$funData=array();
$subData=array();
$keyStorage= array();
$ifData = array();

```

```

$ifValue = array();
$functionArray = array();
$funParams=array();
}
public function SetexSqlStmt($exSqlStmt){
    global $exSqlExeMethod;
    $exSqlExeMethod = $exSqlStmt;
}
public function GetexSqlStmt(){
    global $exSqlExeMethod;
    return $exSqlExeMethod;
}
public function SetobjectSQL($UserInput){
    global $objectSQL;
    $objectSQL= $UserInput;
}
public function GetobjectSQL(){
    global $objectSQL;
    return $objectSQL;
}
public function general($object){

    global $check;
    global $myCount;
    global $DBname;
    global $funData;
    global $ifData;
    global $noNameCount;

    foreach($object as $key=>$stmts){

        if (get_class($stmts)!=""){
            if (substr(get_class($stmts),15)=="Expr_Variable"){
                $check = 0; // check if it is variable (we need
                check variables only)
            }
            if (substr(get_class($stmts),15)=="Expr_MethodCall"){
                $check = 3; // check if it is object like $sth =
                $dbh->prepare
            }
        }
        if (get_class($stmts)=="")
        &&substr(get_class($object),15)=="Expr_MethodCall"){
            $check = 4; // check if it is object like $dbh-
            >prepare
        }
    }
}

```

```

}

if (strstr($key,"name") || strstr($key,"value")
||strstr($key,"class"))
{
    $checkValue = $stmts->value;
    if(strstr($checkValue,"?")||strstr($checkValue,":")){
        $funData['safe']=1;
    }
    if (gettype($stmts)==="object"){
        if (!empty($stmts->parts)){
            foreach ($stmts->parts as $value){
                if(gettype($value)==="string"){

                    if
(get_class($stmts)==="PHPParser_Node_Name"){//only if it is
a function
                        //added 12/11/2013
mysql_query($query) or die("Query failed : " .
mysql_error()
                        //ignore the MYSQL_ERROR function

                    if(strtoupper($value)!="MYSQL_ERROR")
                        $funData['fun']=$value;
                        $funData['line']=$stmts-
>getAttribute('startLine');
                }
            }
            else if (gettype($value)==="object"){

                if(get_class($value)==="PHPParser_Node_Expr_ArrayDimFet
ch"){
                    $funData[]= $value->var-
>name;
                }
            }
            else{
                $funData[]=$value->name;
            }
        }
    }
    else{
        $this->general($stmts);
    }
}

```

```

    }
    else{

        if ($check ==0){
            $check=1;
            $funData[]=$stmts;
        }
        else if ($check==4){ //added 1/22/2014 for $dbh-
>prepare
            $check=1;
            //if(strtolower($stmts)=="query" ||
strtolower($stmts)=="prepare")
            if(in_array(strtoupper($stmts),$this-
>GetexSqlStmt())){
                $funData['fun']=strtolower($stmts);
                $funData['line']=$object-
>getAttribute('startLine');
                $funData[]=null; //because analyst skip
the first value need have a placeholder.
            }
        }

    }
}

else if(strpos($key,"expr") && $check==3 && $this-
>GetObjectSQL()==1){ // check if it is object like $sth =
$dbh->prepare 11/11/2013
    if(in_array(strtoupper($stmts),$this-
>GetexSqlStmt())){
        $funData['fun'] = strtolower($stmts->name);
        $funData['line']=$stmts-
>getAttribute('startLine');
    }
    $this->general($stmts);
}

else{
    $this->general($stmts);
}
}
}

public function breakStmts($object){

```

```

        if (substr(get_class($object),15)=="Stmt_If" &&
!empty($object->elseifs)){
            foreach($object->elseifs as $elseifs){
                $this->breakStmts($elseifs);
            }
        }

        if (substr(get_class($object),15)=="Stmt_If" &&
!empty($object->else)){
            $this->breakStmts($object->else);
        }

        if (substr(get_class($object),15)=="Stmt_Function"){

            $this->fun($object,$object->name);
        }
        //it mean it is a if/try/while type of block statement
        elseif(!empty($object->stmts) && gettype($object->
stmts)=="array"){
            foreach($object->stmts as $stmts){
                //it mean inside the block it has more block
                if(!empty($stmts->stmts) && gettype($stmts->
stmts)=="array"){
                    $this->breakStmts($stmts);
                }
                else{
                    $this->general($stmts);
                    $this->arrayHandle();
                }
            }
        }
        else{
            $this->general($object);
            $this->arrayHandle();
        }
    }

public function arrayHandle(){

global $funData;
global $keyStorage;
global $ifData;
global $ifValue;
global $noNameCount;
$preArray=$this->GetFunData();

```

```

$keepArray=$preArray; //keep array in case the function
call without set variable
$spliceArray=array_splice($preArray, 1); //first index is
key rest is value

if(array_key_exists('safe',$spliceArray)){ //added
1/26/2014
    array_push($ifData[$preArray[0]],9999);
}

//if key is already exist
if(in_array($preArray[0],$keyStorage)){

    foreach ($spliceArray as $arrayData){
        array_push($ifData[$preArray[0]] ,$arrayData);
    }
}

//function call without variable
else if ($preArray[0]==null){

    $ifData[$noNameCount]=$keepArray;
    $noNameCount++;
}

else{
    $keyStorage[]=$preArray[0];
    $finalKeys = array_unique($keyStorage); //remove
duplicate keys

    if ($spliceArray != ""){
        $ifData[$preArray[0]] = $spliceArray;
    }
}

$this->destroy_Fundata();
}
//this handle function stmt
public function fun($object,$functionName){

global $check;
global $myCount;
global $DBname;
global $funData;

```

```

global $functionArray;
global $noNameCount;
global $funParams;
$functionAlert=1; //0 is true 1 is false

foreach($object as $key=>$stmts){

if (get_class($stmts)!=""){
    //echo substr(get_class($stmts),15)."</br>";
    if (substr(get_class($stmts),15)=="Expr_Variable"){
        $check = 0; // check if it is variable (we need
check variables only)
    }
}
//sub statement need to improved
if($key=="stmts"){

    if (gettype($stmts)=="array"){
        if($myCount==null){
            $myCount=0;
        }
        $subTempArray= array();
        foreach($stmts as $subStmts){
            $this->general($subStmts);
            $preArray=$this->GetFunData();

            $savArray=$preArray;
            $spliceArray=array_splice($preArray, 1);
//first index is key rest is value
//if there no assign variable
            if ($preArray[0]==""){
                $subTempArray[$noNameCount]=$savArray;
                $noNameCount++;
            }
            else
                $subTempArray[$preArray[0]]=$spliceArray;

            $this->destroy_Fundata();
            //echo "</br>";
            //echo "Sub Stmt: ".$myCount."</br>";
            $myCount++;
        }
        $this->
>functionArray($functionName,$subTempArray);
    }
}

```

```

}

else if ($key=="params"){ // add params value to array

    $tempParmArray=array();
    foreach($stmts as $stmt){
        $tempParmArray[]=$stmt->name;
    }
    $funParams[$functionName]=$tempParmArray;
}
}
}

//all the data
public function hashTable($key, $object){
    global $array;
    if($key!="")
        $array[$key]=$object;
}

public function functionArray($key, $object){
    global $functionArray;
    if($key!="")
        $functionArray[$key]=$object;
}

public function getFunctionData(){
    global $functionArray;
    return $functionArray;
}

//return all the data
public function getData(){
    global $array;
    return $array;
}

//data in one statment
public function GetFunData(){
    global $funData;
    return $funData;
}

//new array for each statement
function destroy_Fundata() {

```



```

        global $funData;
        $funData=array();

    }

    public function GetFunParams(){
        global $funParams;
        return $funParams;
    }

    public function destory_ifdata(){
        global $ifdata;
        $ifdata = array();
    }
}

class myAn{
private $check ; // the variable to be check
private $user; //check if the variable is user input
private $myTestArray =array();
private $userInputCheck= array();
private $dangerFun=array(); //function that need to be
checked
private $avoidFun=array();
private $exUserInput = array(); //exeternal userInput (from
the ini file)
private $exSqlExeMethod = array(); //external sql exe stmt
(from the ini file)
private $safeFun = array(); //safe function from the config
private $objectSQL; //determine if it is a db using prepare
or query stmts
private $onlyOne; //if we only care about the first
parameter
private $vulCount; //number of vulnerability
private $testFile;
public function SetexUserInput($exUser){
    global $exUserInput;
    $exUserInput=$exUser;
}

public function GetexUserInput(){
    global $exUserInput;
    return $exUserInput;
}
}

```

```

public function SetexSqlStmt($exSqlStmt){
    global $exSqlExeMethod;
    $exSqlExeMethod = $exSqlStmt;
}
public function GetexSqlStmt(){
    global $exSqlExeMethod;
    return $exSqlExeMethod;
}

public function SetSafeFunFromConfig($safeFunConfig){
    global $safeFun;
    $safeFun = $safeFunConfig;
}
public function GetSafeFun(){
    global $safeFun;
    return $safeFun;
}

public function SetobjectSQL($UserInput){
    global $objectSQL;
    $objectSQL= $UserInput;
}

public function GetobjectSQL(){
    global $objectSQL;
    return $objectSQL;
}

public function SetOnlyOne($UserInput){
    global $onlyOne;
    $onlyOne= $UserInput;
}

public function GetOnlyOne(){
    global $onlyOne;
    return $onlyOne;
}

public function checkFun($objects,$objKey){
    //$checkArray=array("MYSQL_QUERY"); //if the function
using sql exe stmts
    $checkArray=$this->GetexSqlStmt();
    $result =
array_intersect(array_map('strtoupper',$objects),$checkArra
y);

```

```

$newStatement =newClass();
$tempParam=$newStatement->GetFunParams();

foreach($objects as $key=>$object){
    if(is_int($key)){
        if(in_array($object,$tempParam[$objKey])){

            return $result;
        }
    }
}

}

public function addCheckFun($fun,$objects){ //the dangerFun
that is used is added to checkarray

    global $dangerFun;

    foreach($objects as $object){
        if($fun==$object['fun']){
            $dangerFun[]=strtoupper($fun);
            $dangerFun=array_unique($dangerFun);
        }
    }
}

public function getDangerFun(){
    global $dangerFun;
    return $dangerFun;
}

public function getTarget($objects,$myKey,$Fon){ //check if
any variables use my_sql_query $myKey is the function name

    global $dangerFun;
    $objects=array_filter($objects);
    $tempTest=$this->GetexSqlStmt();

    foreach($tempTest as $temp){
        $dangerFun[] = $temp;
    }

    $dangerFun=array_unique($dangerFun);
    //the function that we want to check

```

```

    $checkArray=$dangerFun;
    foreach($objects as $key=>$object){
        foreach($object as $var){
            $newStatement =newClass();
            $testTemp=$newStatement->GetFunParams();
            if(in_array(strtoupper($var),$checkArray) ){
//mysql found ?
                if($key == null){
                    $this->
>focusTarget(null,$objects,$Fon);
                }
                else{
                    if($oldKey!=$key) // added
1/5/2014 prevent repeat analyst
                    $this->
>focusTarget($key,$objects,$Fon);
                    $oldKey = $key;
                }
            }
        }
    }
}

public function getTarget2($objects,$myKey,$Fon,$ifdata){
//check if any variables use my_sql_query $myKey is the
function name

    global $dangerFun;
    $tempTest=$this->GetexSqlStmt();
    foreach($tempTest as $temp){
        $dangerFun[] = $temp;
    }
    $dangerFun=array_unique($dangerFun);
    //the function that we want to check
    $checkArray=$dangerFun;
    foreach($objects as $key=>$object){
        foreach($object as $var){
            $newStatement =newClass();
            $testTemp=$newStatement->GetFunParams();

            if(in_array(strtoupper($var),$checkArray) ){
//mysql found ?

                $sturn=$this->
>checkParm($myKey,$objects,$ifdata);

```

```

        if ($sturn==1 and !empty($myKey)){ //if
param is use in sql exe statement

                if($key == null){
                    $this-
>focusTarget(null,$objects,$Fon,$myKey,$ifdata);
                }
                else{
                    $this-
>focusTarget($key,$objects,$Fon,$myKey,$ifdata);
                }
            }
        }
    }

public function checkParm($myKey,$objects,$ifdata){

    global $dangerFun;
    global $myTestArray;
    $newStatement =newClass();
    $tempParm=$newStatement->GetFunParams();

    foreach ($objects as $object){
        if
(in_array(strtoupper($object['fun']),$dangerFun)){

            foreach ($object as $key=>$Iobject){

                if (is_int($key)){ // only check
variable
                    if(
in_array($Iobject,$tempParm[$myKey])){ // check if
parameter is use in the danger function
                        return 1;
                    }
                    //added 1/23/2014
                    else{

                        if(!empty($objects[$Iobject])){
                            $this-
>traceback($objects,$Iobject);

```



```

//removing the first variable of prepare statment
11/11/2013
foreach( $objects[$key] as $objKey=>$set){
    if($objKey=="fun" ){
        if(gettype($objKey)=="string"){ //some
reason index 0 keep popping up disregard the comparesion of
the value

        if(in_array(strtoupper($objects[$key]['fun']), $userInp
utSql) && !empty($objectSql)){
            if(strtolower($set)=="query"
||strtolower($set)=="prepare")
                unset($checkVar[0]);
            }
        }
    }
}

$tempOnlyOne =$this->GetOnlyOne();
if (!empty($popIt) && !empty($tempOnlyOne)){ //only
check the first parameter of mysql_stmt 11/9/2013
    array_pop ($checkVar);
}

$treeArray=array(); // with key
foreach($checkVar as $var){ // variable found that
need to be check
    $this->traceback($objects,$var);
    $treeArray[$var]=$this->getMyTestArray();
    $this->destoryTestArray();
}

$tempFun=($objects[$key]['fun']);
if(!empty($treeArray) && !empty($tempFun)){
    $objLine= $objects[$key]['line'];
}
foreach ($objects[$key] as $value){ //added 1/26/2014
    if($value==9999){
        $SafeObject = 1;
    }
}
if ($objects[$key]['safe']==1){//added 1/26/2014
    $SafeObject = 1;
}
}

```

```

    $lastStepCheck=$this-
>checkFunByLine($objects[$key]['line'],$objects); //check
if it is a function call 11/09/2013

    if ($SafeObject!=1){

        foreach($treeArray as $indKey =>$treeArrayInd){

            $result=$this-
>checkArrayUserInput($treeArrayInd); //if an array has user
input (_GET,_POST,_REQUEST)
            if(!empty($result)){

                $result2=$this-
>checkArrayDef($treeArrayInd); // check if users use sql
prevention
                if(!empty($result2)){ //if use sql
prevention
                    }
                    else{ // at this point the variabe has
no sql prevention method and has user input _GET etc
                        $result3= $this-
>paramCheck($indKey,$tempFun,$objects); //check if the
variable is really use in the function
                        //if it is empty it means it is
not use in function
                        if( !empty($result3 )){

                            $para2 = $this-
>paramCheck2 ($indKey,$tempFun,$objects);

                            if(!empty($para2)){
                                //echo "<span
class='Nothreat'>Line: $objLine $indKey safe with
prevention method</span></br>";
                            }
                        }
                        else //not find in the function
statement
                            {

                                if(in_array(strtoupper($lastStepCheck),$userInputSql))
                                {

```



```

        if(!empty($indKey)){
            echo "<span
class='threat'><FONT COLOR=#990000><b>Line:
".$objLine."</b> $indKey Not safe.Not a function call and
no prevention method found </FONT></span> </br>";
            $this->addVul();
            $this-
>textFromFileByLine($tempFileName,$objLine);
        }
    }
}
else{
    if($Fon){ // for function only
        $checkFunOnly = $this-
>checkArrayDef($treeArrayInd);
        if (!empty($checkFunOnly))
            echo""; //echo "<span
class='NothreatF'>Line: $objLine $indKey safe with
prevention method</span></br>";
        else{
            if(!empty($indKey)){
                echo "<span
class='threatF'><FONT COLOR=#990000><b>Line:
".$objLine."</b> $indKey not safe no prevention
method</FONT></span></br>";
                $this->addVul();
                $this-
>textFromFileByLine($tempFileName,$objLine);
            }
        }
    }
    else{ //if the an array doesn't have
user input and not a function only

        if(in_array(strtoupper($indKey),$userInputArray )){
            echo "<span
class='threatF'><FONT COLOR=#990000><b>Line:
".$objLine."</b> $indKey not safe no prevention method for
user input</FONT></span></br>";

```



```

public function checkFunByLine($line,$objects){

    foreach($objects as $object){
        if($object['line']==$line){
            return $object['fun'];
        }
    }
}

public function paramCheck($key,$fun,$objects){ // $fun is
the function name $key is the variable inside the function
that needs to be check

    $newStatement =newClass();
    $normalStmt=$newStatement->GetFunParams();
    $funStmt=$newStatement->getFunctionData();
    $dangerFun=$this->getDangerFun();

    foreach ($objects as $object){
        if ($object['fun']==$fun){
            $index= ( array_search( $key,$object));
//index of the variable
            $stran=$normalStmt[$fun][$index];
            foreach($funStmt[$fun] as $objects2){

                if(in_array(strtoupper($objects2['fun']),$dangerFun)){
                    $find=array_search(
$stran,$objects2);

                    if (is_int($find)){
                        return "find";
                    }
                }
            }
        }
    }
}

//12-4-2013 added to check if the variable is using any
prevention method
public function paramCheck2($key,$fun,$objects){ // $fun is
the function name $key is the variable inside the function
that needs to be check

    $newStatement =newClass();

```

```

    $normalStmt=$newStatement->GetFunParams();
    $funStmt=$newStatement->getFunctionData();
    $dangerFun=$this->getDangerFun();
    foreach ($objects as $object){
        if ($object['fun']==$fun){
            $index= ( array_search( $key,$object));
//index of the variable
            $stran=$normalStmt[$fun][$index];
            foreach($funStmt[$fun] as $objects2){

                if(in_array(strtoupper($objects2['fun']),$dangerFun)){

                    $find=array_search(
$trian,$objects2);
                    if (is_int($find)){
                        $result=$this-
>checkArrayDef($funStmt[$fun][$stran]);
                        return ($result);
                    }
                }
            }
        }
    }
}

public function traceback($objects,$var){

    global $myTestArray;
    $checkVar =array();
    $maxCount =sizeof($objects[$var])+1;
    $count=0;
    foreach( $objects[$var] as $objKey=>$set){

        $count=$count+1;
        if($count<$maxCount){
            $checkVar[]=$set;
        }
        unset($objects[$var]); //1/5/2014 remove infinte
loop
    }
    foreach( $checkVar as $myVar){
        $myTestArray[]=$myVar;
        if($myVar!=$var){
            $this->traceback($objects,$myVar);
        }
    }
}

```

```

    }
}
public function destoryTestArray(){
    global $myTestArray;
    $myTestArray=array();
}

public function getMyTestArray(){
    global $myTestArray;
    return $myTestArray;
}

public function addVul(){
    global $vulCount;
    $vulCount++;
}

public function getVul(){
    global $vulCount;
    return $vulCount;
}

public function destoryVul(){
    global $vulCount;
    $vulCount=0;
}

public function checkArrayUserInput($array) {
    // $userInputArray=array("_GET","_POST","_REQUEST"); //
    array for all the user input method
    $userInputArray=$this->GetexUserInput();
    $result =
    array_intersect(array_map('strtoupper',$array),$userInputAr
    ray);
    return $result;
}

public function checkArrayDef($array) {
    $userDefArray=$this->GetSafeFun(); //array for all the
    sql prevention method
    $result =
    array_intersect(array_map('strtoupper',$array),$userDefArra
    y);
    return $result;
}

```

```

public function destoryAn(){
    global $check ; // the variable to be check
    global $user; //check if the variable is user input
    global $myTestArray;
    global $userInputCheck;
    global $dangerFun; //function that need to be checked
    global $avoidFun;
    global $exUserInput ; //exeternal userInput (from the
ini file)
    global $exSqlExeMethod ; //external sql exe stmt (from
the ini file)
    global $objectSQL; //determine if it is a db using
prepare or query stmts
    global $onlyOne; //if we only care about the first
parameter
    global $vulCount; //number of vulnerability

    $check=null;
    $user=null;
    $myTestArray =array();
    $userInputCheck= array();
    $dangerFun=array(); //function that need to be checked
    $avoidFun=array();
    $exUserInput = array(); //exeternal userInput (from
the ini file)
    $exSqlExeMethod = array(); //external sql exe stmt
(from the ini file)
    $objectSQL=null;
    $onlyOne=null;
    $vulCount=null;
}

public function setFile($FileArray){
global $testFile;
$testFile=$FileArray;
}

public function getFile(){
global $testFile;
return $testFile;
}

//added 1/24/2013 print out the entire line
public function textFromFileByLine($fileName,$line){

```



```
$myFile=$this->getFile();  
$line=$myFile[$fileName][$line-1];  
echo " ".$line." </br></br>";  
}  
  
}  
?>  
  
</body>  
</html>
```

APPENDIX B

TEST CASE TABLE STRUCTURE

Attributes	Description
Tag Number	A unique number, which should be auto incremented when a new TAG is inserted into the database.
Rev#	Whenever the information of a TAG is modified, the Rev# is increased by one automatically
Date	Date the original or updated revision was created. When a TAG is inserted or revised, the value of Date is always the current date.
Sub-Category	Has one of the predefined values such as: AC Panel, Arc Resistant, and Auto Xfer. These predefined values should be stored in a separate table.
Complexity	Contains one of the following: A, B, C, D, E, F, and G, which should be stored in a separate table.

Lead Time	Value is in week days.
Tag Description	General description associated to TAG
Tag Notes	General Notes Associated to TAG
Price Note	General Notes associated to Tag Pricing
TAG Member	User selected from User table during edit/creation process
Price Expires	The value is Date (created/edited) plus months inputted during created/edited process.
Product Option	<p>TAG price may be applied to one or more of the following 4 product types:</p> <ul style="list-style-type: none"> • HVL • HVL/CC • Metal Clad • MVMCC <p>For each product type, it has price in different country currency, which is calculated using the following formula:</p> <p>Country List dollar equal Install Cost times</p>

	<p>Product Multiplier times Country Multiplier</p> <p>Product Type and its multiplier should be stored in a separate table. Country and its multiplier should be stored in a separate table.</p>
Attachments	<p>Attachments can be any pdf, doc, and xls, bmp document that has been associated and uploaded during the creation /editing process.</p>
Applied FO table	<p>Each TAG may have an applied FO table, which contains information about Quote, Factory order or both.</p>
<p>The following four attributes are only visible to Tag Members and OE group. All TAGs have the same Labor price per hour and Engineering price per hour, which should be stored in a separate table.</p>	
Material	<p>Cost in dollar based on Material cost.</p>
Labor	<p>Cost in dollar based on hours inputted x Labor price per hour.</p>
Engineering	<p>Cost in dollar based on hours inputted x Engineering price per hour.</p>
Install Cost	<p>Total of Material plus Labor plus Engineering.</p>

Attributes	Description
Tag Number	A unique number, which should be auto incremented when a new TAG is inserted into the database.
Rev#	Whenever the information of a TAG is modified, the Rev# is increased by one automatically
Date	Date the original or updated revision was created. When a TAG is inserted or revised, the value of Date is always the current date.
Sub-Category	Has one of the predefined values such as: AC Panel, Arc Resistant, and Auto Xfer. These predefined values should be stored in a separate table.
Complexity	Contains one of the following: A, B, C, D, E, F, and G, which should be stored in a separate table.
Lead Time	Value is in week days.
Tag Description	General description associated to TAG
Tag Notes	General Notes Associated to TAG
Price Note	General Notes associated to Tag Pricing

TAG Member	User selected from User table during edit/creation process
Price Expires	The value is Date (created/edited) plus months inputted during created/edited process.
Product Option	<p>TAG price may be applied to one or more of the following 4 product types:</p> <ul style="list-style-type: none"> • HVL • HVL/CC • Metal Clad • MVMCC <p>For each product type, it has price in different country currency, which is calculated using the following formula:</p> <p>Country List dollar equal Install Cost times Product Multiplier times Country Multiplier</p> <p>Product Type and its multiplier should be stored in a separate table. Country and its multiplier should be stored in a separate table.</p>

Attachments	Attachments can be any pdf, doc, and xls, bmp document that has been associated and uploaded during the creation /editing process.
Applied FO table	Each TAG may have an applied FO table, which contains information about Quote, Factory order or both.
The following four attributes are only visible to Tag Members and OE group. All TAGs have the same Labor price per hour and Engineering price per hour, which should be stored in a separate table.	
Material	Cost in dollar based on Material cost.
Labor	Cost in dollar based on hours inputted x Labor price per hour.
Engineering	Cost in dollar based on hours inputted x Engineering price per hour.
Install Cost	Total of Material plus Labor plus Engineering.

