Int J Syst Assur Eng Manag (Oct-Dec 2012) 3(4):343–351
DOI 10.1007/s13198-012-0125-6

ORIGINAL ARTICLE

Integrated approach to prevent SQL injection attack and reflected cross site scripting attack

Pankaj Sharma · Rahul Johari · S. S. Sarma

Received: 8 June 2012 / Revised: 28 August 2012 / Published online: 26 September 2012

Abstract The Internet and web applications are playing very important role in our today's modern day life. Several activities of our daily life like browsing, online shopping and booking of travel tickets are becoming easier by the use of web applications. As the volumes of the web applications are increasing the security of web applications becomes a major concern. Most of the web applications use the database as a back end to store critical information such as user credentials, financial and payment information, company statistics etc. These websites are continuously targeted by highly motivated malicious users to acquire monetary gain. Multiple client side and server side vulnerabilities like SQL injection and cross site scripting are discovered and exploited by malicious users. SQL injection attacks and cross site scripting vulnerabilities are top ranked in the open web application security project top ten vulnerabilities list. A number of security approaches are proposed and used like secure coding practices, encryption, static and dynamic analysis of code to secure the web applications but statistics shows that these vulnerabilities are still transpiring at the top. In this paper, we present an integrated model to prevent SQL injection attacks and reflected cross site scripting attack in PHP based implementation. This model is more effective to prevent SQL injection attack and reflected cross site scripting attack in production web environment. Our mechanism is divided into two modes, a safe mode and a production mode environment. In the safe mode we construct a security query model for SQL injection and sanitizer model for reflected cross site scripting attack for each identified SQL queries for SQL injection attacks and input entry points for reflected cross site scripting attacks. In the production environment, input entries which create dynamic SQL queries are validated against security query model generated in safe mode and normal input text entered by the user is validated by sanitizer model instrumented in the code at safe mode. The results and analysis shows that the proposed approach is simple and effective to prevent common SQL injection vulnerabilities and reflected cross site scripting vulnerabilities.

Keywords SQL injection attack · Reflected cross site scripting attack · OWASP · Static analysis · Dynamic analysis · Web security · Application vulnerabilities · Illegitimate access · Authentication bypass · Input sanitization · Pattern matching · Risk management · Assurance · Insurance

Abbreviations

XSS	Cross site scripting
SQL	Structured query language
SQLIA	SQL injection attacks
OWASP	Open web application security project
MHAPSIA	Model based hybrid approach to prevent SQL injection attacks
DFA	Deterministic finite automata
NFA	Nondeterministic finite automata
AMNeSIA	Analysis and monitoring for neutralizing SQL injection attacks
JDBC	Java database connectivity
HTML	Hyper text markup language

P. Sharma (✉) · S. S. Sarma
Department of Electronics and IT, CERT-In,
Goverment of India, New Delhi, India
e-mail: Pankaj.vats@gmail.com

S. S. Sarma
e-mail: ss.sarma@gmail.com

R. Johari
USIT, GGSIP University, Sector 16-C Dwarka, Delhi, India
e-mail: rahuljohari@hotmail.com

1 Introduction

In today's life Web applications provide various facilities like online shopping-banking, e-reservation, e-governance etc. Different types of functionalities are built into the web applications according to user requirements without considering the security requirements such as secure coding practices leading to serious web vulnerabilities. The most exploited vulnerabilities in the web applications are SQL injection and reflected cross site scripting vulnerability.

Software assurance requires implementation of security practices throughout the Software Development Life Cycle (SDLC). Further, current best practices include Statics Application Security Testing and Dynamic Application Security Testing. These methods, however test the source code and functionality of the web application for known vulnerabilities bugs in the software code.

The methodology proposed in this paper helps in securing the software at the code level, prior to deployment in the practical applications. The current scope of the proposed method envisages certain security methods to prevent most popular web application attacks i.e., SQL injection and one type of cross site scripting attack namely the reflected cross site scripting.

The methodology and solution proposed in the our paper is an enhancement of a method proposed in the paper on model based hybrid approach to prevent SQL injection attack (MHAPSIA) (Kunal et al. 2011). We modified the existing algorithms used in MHAPSIA by reducing certain drawbacks of existing approach for better prevention of SQL injection attacks. Further, new modules are integrated to prevent reflected cross site scripting attacks.

1.1 SQL injection attacks

SQL injection attacks (SQLIAs) (Common Weakness Enumeration 2012)—structured query language (SQL) is an interpreted language used in database driven web applications which construct SQL statements that incorporate user-supplied data or text. If this is done in an unsafe manner, then the web application may be vulnerable to SQL Injection Attack i.e. if the user supplied data is not properly validated then users can modify or craft a malicious SQL statement and can execute arbitrary code on the target machine or modify the contents of the database.

1.1.1 Problem formalization

Any web application can be formalized with respect to SQL injection attack as follows:

- It accepts the input from a user or system.
- It concatenates input with hardcoded SQL statement and builds complete query structure.
- The generated query gets executed and concatenates result with HTML code.

In the context of above formalization SQL injection attack is targeted on a program at the database layer which is connected to a web application. This SQL injection attack (Rahul and Sharma 2012) exploits weakness or vulnerability in the target program to properly verify the input supplied to it through a web form. In a typical SQL injection attack the attacker posts specially crafted structured query language (SQL) statements which are executed in the database server and produce malicious outcomes.

1.2 Cross-site scripting (XSS)

Cross-site scripting (XSS) (Common Weakness Enumeration 2012) is another common web application attack technique that involves echoing attacker-supplied code into a client's browser via web pages viewed by the target users. Here the attacker's host or inject attack code written in different static or dynamic contents such as HTML, Java, JavaScript, ActiveX, Flash or any other browser supported technology.

When an attacker gets a user's browser to execute his/her script, the script will run within the security context (or zone) of the hosting web site. With this kind of privilege, the application code has the ability to read, change and transmit any critical data accessible by the browser. Successful XSS attack allows attackers to hijack a user's account via cookie, redirecting the user to another website from the website visited and thereby facilitating other types of attacks such as Phishing or drive-by-download attacks. XSS attack poses significant risk in cases where the browser interacts closely with file system on the users' computers for loading content.

1.3 Reflected XSS

Reflected XSS (Rahul and Sharma 2012) occurs when the server does not properly sanitize the output server to a visiting web browser/client. In the typical attack scenario, the attacker targets visitor of a specific website example 'abc.com' containing reflected XSS vulnerability and tricks targeted user to click on a maliciously crafted URL. The user intends to visit the 'abc.com' and does so but in the process the client side script/code contained in the malicious URL supplied by the attacker gets executed thereby enabling the attacker to gain access to sensitive user credentials, cookies etc.

2 Related work

MHAPSIA (model based hybrid approach to prevent SQL injection attacks in PHP (Kunal et al. 2011)): This approach focuses on static analysis and run time validation i. e. it runs the application in two different modes safe and real. In the safe environment, it creates a query model for all legitimate SQL statements using a deterministic finite automata (DFA). The nodes of the nondeterministic finite automata (NFA) are SQL keywords and operators with special symbols. In the real environment, SQL statements are intercepted with the instrumented code and then validated with the query model generated in the safe environment. A limitation of this approach is that the particular malicious SQL injection queries such as **'or *1* = *1* #** are not blocked by the query model as the query model for such injections is same as that of a legitimate query model.

AMNeSIA (JSP approach (Orso 2005)): Similarly, model based hybrid approach proposed in "Analysis and Monitoring for NEutralizing SQL Injection Attacks" (AMNeSIA) prevents SQL injection attacks by forming a query model which is again an NFA based on the same construction pattern as that of the above approach. An inadequacy of Amnesia tool is that, it can only be used in Java Server Pages (JSP) web applications. This tool makes use of the Java String Analyzer (JSA) library to construct a query model which is not available in any other languages. Furthermore, this tool depends on the accuracy of the string analyzer. Further experiments using this approach were conducted by Junjin (2009).

Static analysis technique (Gould et al. 2004): Further contribution to the related work is proposed by Gould, Su and Devanbu in the Java Database Connectivity (JDBC) checker that statically validates the correctness of dynamically generated queries. This technique is able to detect one of the major root causes for SQLIA, which is improper input checking. A shortcoming of this tool is that it doesn't focus on more general forms of SQLIAs.

Key analysis technique (Keromytis 2004): Another effective query randomization approach is SQLrand, but the security of this approach is purely dependent on its key used for randomization.

Automated generation of prepared statements to remove SQL injection vulnerabilities (Stephen et al. 2009): Thomas presents an algorithm in which prepared statement in SQL queries are replaced by secure prepare statements for removing SQL vulnerabilities.

Different other methods such as removal of SQL query passed by user in SQL query attributes values (Jeom-Goo 2011), dynamic candidate evaluations approach (Prithvi 2010), obfuscation-based analysis of SQL injection attacks (Raju and Cortesi 2010).

Mitigations were also proposed for prevention of XSS attacks such as protecting cookies from cross site script attacks using dynamic cookies rewriting technique (Rattipong and Bunyatnoparat 2011), execution-flow based method for detecting cross-site scripting attacks (Qianjie et al. 2010).

Automatic creation of SQL injection and cross-site scripting (XSS) attacks (Ardilla) (Kiezun et al. 2009): Adam Kie zun has suggested a technique to find out both SQL Injection and XSS vulnerabilities based on automated tool called Ardilla. This method uses static code analysis to find vulnerabilities. This technique examines source code of the application, creates concrete inputs that expose vulnerabilities. It is based on input generation, taint propagation, and input mutation to find variants of an execution that exploit vulnerabilities.

Our approach provides a protection against SQL injection using hybrid approach logic i.e. query model generation but also with a check on input field for a pattern such as "or", i.e. if any such pattern is found in the input entry point, then the program exits from the application and an error message is given. Further, the approach has been extended to prevent another type of attack called reflected cross site scripting.

3 Classifying vulnerabilities

3.1 SQL injection attacks

SQL injection (Common Weakness Enumeration 2012) is an attack technique that mainly occurs due to the insecure coding practices. This attack modifies the SQL statement in such a way so that the legitimate inputs or the authentication of a legitimate user is bypassed and the database executes the malicious code supplied by the attacker. The basic cause of the vulnerability is the un-sanitized user input.

To categorize SQL Injection let us consider a web page for User Authentication as shown in Fig. 1. Here, an attacker tries to insert a universal true condition with an OR in the username field which modifies the existing SQL query used for authenticating the user credentials by commenting rest of the logic for password authentication.

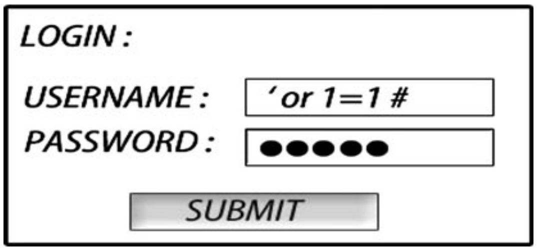


Fig. 1 SQL injection through login form for user authentication

The SQL injection discussed above is classified as injection based on tautology with a comment, as the statement with OR is a tautology followed by a comment. Let us demonstrate SQL injection in pre crafted SQL query.

Select * from table_name where username='$uname' and pass='$pwd'

Now on entering the credentials shown above, the query is formulated as:

Select * from table_name where username=''or 1 = 1 # and pass='anything'

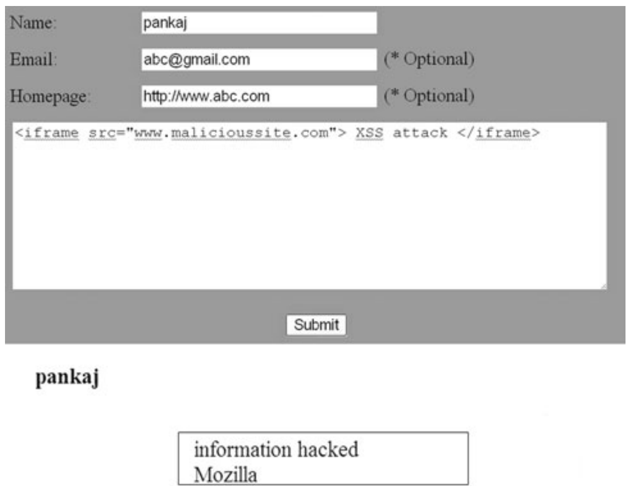
Now, due to improper sanitization of the user input, the query on the server side gets executed with user name as either a blank or a universal true condition followed by a # due to which the rest of the SQL statement gets commented and password accepts any string. This execution leads to successful login and hence bypassing the user authentication phase.

3.2 Reflected cross site scripting attack

Reflected cross site scripting is a web application vulnerability which could be exploited by the attacker by inserting malicious scripts on the victim's browser or client when he accesses a webpage. The malicious code hosted by an attacker can be written in any of the technology supported by the browsers such as HTML, Java, JavaScript, ActiveX, flash etc. The reflected cross site scripting vulnerability exists due to improper sanitization of all the entry points where an attacker is capable of writing any malicious scripts.

Lets us consider a web application shown below to show exploitation of reflected cross site scripting vulnerability.

The attack shown above is classified as reflected XSS attack. In this attack user inserts a malicious script containing iframe which is reflected back by the web server, so whenever a user visits the web page containing the iframe he/she is redirected to "www.malicious.com" which is an attackers' website (Fig. 2).



Name:	pankaj	
Email:	abc@gmail.com	(* Optional)
Homepage:	http://www.abc.com	(* Optional)

```
<iframe src="www.malicioussite.com"> XSS attack </iframe>
```

pankaj

information hacked
Mozilla

Fig. 2 Iframe injection using reflected cross site scripting vulnerability

4 Model based hybrid approach to prevent SQL injection attack and its limitations

MHAPSIA is a PHP application to prevent SQL injection attacks for PHP web applications. This tool works in two different modes described as follows:

Safe mode: In this mode, the application is first scanned to locate all SQL statements and names them as hotspot locations and also assign 'line number' and 'id' to each hotspot identified. Thereafter all the files are instrumented where in a monitor function is added preceding every SQL statement. Finally, the application is executed with all legitimate inputs to create the query model using DFA which remains in runtime arrays.

Real mode: In this mode, all the files are processed through two phases i.e. 'scanning phase' and 'instrumenting phase'. Thereafter the application is executed in production mode wherein the query model for all the dynamically generated queries is compared with static query model generated in safe mode. If the dynamically generated query model complies with the static query model then the authentication is granted otherwise the input is classified as an attack, the information stored in the error log file and a suitable error message is thrown.

4.1 Limitations of existing approaches

- The major drawback of MHAPSIA is that it doesn't prevent certain SQL injection attacks. The query model on which the success of the tool depends is same for legitimate input as well as for one of the SQL injection inputs in some cases. Consider the below two SQL statements in proof of concept :

Legitimate inputs: Select * from user_details where user_name = '$uname' and user_pass = '$pass'

SQL injection input: Select * from user_details where user_name = 'or 1 = 1 # and user_pass = 'any string'

The two query models shown in Fig. 3a, b, Shows that they both have 16 states and both of the query model ends in the same final state which act as a proof of concept of bypassing the authentication for this particular SQL injection attack.

- MHAPSIA is not designed for preventing reflected cross site scripting vulnerabilities.
- Query models are stored in run time arrays only there is no permanent source of storage due to which whenever the application is run, safe mode query models need to be constructed each time.

5 Proposed solution

The solution proposed is a modification to the existing MHAPSIA model for PHP web applications. In our solution we modified the existing model by incorporating the logic to thwart number of kinds of SQL injection attack and integrated a separate module to avert reflected cross site scripting attacks. In the following sections we present a list of algorithms that are related to each other in preventing the SQL injection and reflected cross site scripting (XSS) attacks.

5.1 Algorithms

Algorithm 1: Verify the source code for XSS and SQL injection attacks

Inputs : Path Of Application Files
Outputs: Locations Where Vulnerability May Exist

get_count_files (spath): return the number of files in the path specified
get_ all_tokens (file): return the number of tokens present in the specified file
array_merge (array1, array2): merge two arrays specified in the parameter
get_count (array): return the number of elements in the specified array.

```
Set count= get_count_files (spath)
Set i=0,j=0,k=0,$vul_sig=array()
While (i<=count) do
token_list <- get_ all_tokens (file[i])
verified_spot<-{}
v_count<-0
If(SQL==checked)
        Set $vul_sig=array_merge ($vukl_sig,$globals['SQL')
End if
Else if(XSS==checked)
        Set $vul_sig=array_merge ($vul_sig, $globals ['XSS'])
End if
Else if(rfi==checked)
        Set $vul_sig=array_merge ($vul_sig, $globals['RFI'])
End if
For j=0 to get_count (vul_sig) do
        For i=0 to get_count (token_list) do
                If (token_list(i)==vul_sig(j)
                        Print filename:  . $spath.$file [i]
                        While (token_list[i]! =';')
                                Print token_list[i]
                                $i<-$i + 1
                        End while
                End if
        End For
End for
End while
```

Algorithm 2: Identifying hotspots and instrument file for XSS

Input: Path Of Application Files
Output: Instrumented files

get_count_file ($spath) : return total number of files for the specified path
htmlentities (file[$i]): returns all the html entities present in the specified file
get_all_tokens (file1 [$i]): return the total numbers of tokens present in the file specified
preg_match (array, pattern): returns true if pattern specified is found in the array specified
get_name_id (token_list[i]): user defined function to get the extract the name or id field from the input type tag.
get_ispot_id (identified_spot[k]): user defined function to get the id of ispot .
get_ispot_location (identified_id) : user defined function to get the ispot location using its id
instru_file(identified_id,iname_id,identified_spot): function to instrument all the files and add a function sanitiser() to instrumented files.

```
$count=get_count_file($spath)
While($i<=count) do
File1[i]=htmlentities(file[$i])
Token_list=get_all_tokens(file1[$i])
Identified_spot={}
Ispot_count<-0
For i=0 to get_count(token_list)
        If preg_match(token_list[i],'/\< input /i')
                $iname_id=get_name_id(token_list[i]);
                Identified_spot<-location_pointer
                Identified_id<-i_spot_count
                Ispot_count<- i_spot_count +1
        End if
End for
While(i_spot_count!=0)
        Identified_id<-get_ispot_id(identified_spot[k])
        Location<- get_ispot_locvation(identified_id)
        Instrument_files[]=instru_file(identified_id,iname_id,identified_spot)
End while
Return  instrument_files[];
End while
```

Algorithm 3: Code for sanitizing input for XSS: Sanitizer(iname_id)

Input: Data Entered In Entry Points
Output: If Valid Then Allow Access And If Invalid Then Block Execution

```
1:  $var=$Get(iname_id)
2:  $_Globals('html_ vul_tag')
3:  For  i=0 to get_count(html_ vul_tag)
4:          If preg_match( $var, html_ vul_tag(i))==true
5:          Error " SCRIPT ARE NOT ALLOWED"
6:          Break
7:          End if
8:  End for
```

Algorithm 4: Modification of validator.php file

Inputs : Dynamic query and safe mode query model
Outputs: Result of comparison

```
1:  If(accept(dynamic->dfa, static->dfa)) then
2:          if (preg_match($iname_id,'/\'OR/i')
3:                  Print ERROR
4:          Break
5:          Else
6:                  Execute query
7:      End if
8:  Else
9:          Print ERROR
10: End if
```

6 Implementation

The proposed approach is implemented in various PHP applications. In this section, we explain the working of the proposed model and mechanism to mitigate SQL injection and reflected cross site scripting attacks.

The figure shows the complete architecture of the proposed model (Fig. 4).

Further elucidation of the above architecture is as follows:

The proposed tool works in two different modes, safe and production mode. Initially, it starts with verifying the application for SQL and XSS vulnerability for which it takes the complete application path and scans all files of web application for SQL queries and input entry points with respect to an array vul_sig () array. Then the tool enters into the instrumentation phase of the safe mode which is further divided into two modules, 'ispot identification' and 'instrumentation'. In ispot identification, each location is verified in the verification module an ispot id is and ispot line number is assigned. Further, the instrumentation module is divided into two modules, which are:

6.1 Instrumentation for SQL injection

The output of this module is an instrumented file which contains a 'validator program' instrumented before each SQL query located by the 'ispot'. Then these instrumented files are executed in the safe mode in which Security Query model is generated for all legitimate inputs.

6.2 Instrumentation for XSS

The output of this module is an instrumented file which contains a sanitizer program instrumented after each input entry points. This function sanitizes all malicious scripts entered by any attacker or malicious user. In the safe mode, the executions of the instrumented files are not mandatory.

The tool is then entered in production mode wherein, the actual inputs are validated against the Secure Query Model generated in safe mode for preventing SQL injection and all inputs from all input entry points like text boxes and text areas etc. are validated by sanitizer program which was instrumented in the instrumentation phase of the safe mode.

6.3 Basic modules of the integrated model

- Verification module

This module scans all files of web application to locate all SQL query signatures for SQL injection attack and input type html entities for reflected cross site scripting attacks. The signatures are defined in the array named vul_sig(). The module then returns all the locations of the signatures that consist of line number along with the statements consists of array elements.

- Ispot identification module

This module identifies all the ispots which are basically the verified locations of signatures found during verification module. Furthermore, each ispot is assigned an ispot _id and line number. The output of this module is list of ispots which are stored in a metafile.

- Instrumentation module

This module is divided into two sub modules.

Instrumentation module for SQL injection: Instrument() function is used to append Instrumented_function() program before each SQL query statement.

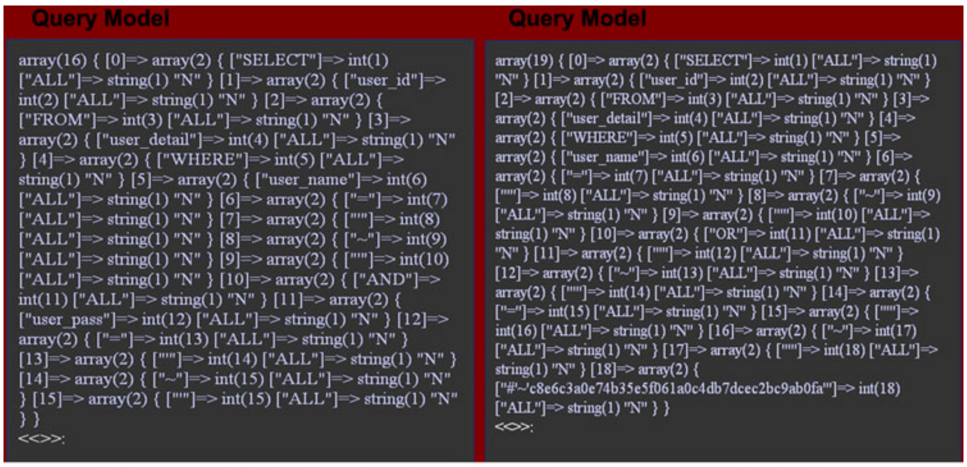
Instrumentation module for XSS: Instrument() function is used to append sanitizer() program after each Input entry point

Output: instrumented files.

- Query model construction module:

In this module the whole instrumented web application is run in safe mode. In this mode we take legitimate inputs from the user. These inputs are then passed to the function called construct_DFA() which calls sql_parser() to construct query model for legitimate inputs. This query model is stored in an array and used in real mode to validate dynamically generated query model.

Fig. 3 Query models for SQL injection input



Query Model

```
array(16) { [0]=> array(2) { ["SELECT"]=> int(1)
["ALL"]=> string(1) "N" } [1]=> array(2) { ["user_id"]=>
int(2) ["ALL"]=> string(1) "N" } [2]=> array(2) {
["FROM"]=> int(3) ["ALL"]=> string(1) "N" } [3]=>
array(2) { ["user_detail"]=> int(4) ["ALL"]=> string(1) "N"
} [4]=> array(2) { ["WHERE"]=> int(5) ["ALL"]=>
string(1) "N" } [5]=> array(2) { ["user_name"]=> int(6)
["ALL"]=> string(1) "N" } [6]=> array(2) { ["="]=> int(7)
["ALL"]=> string(1) "N" } [7]=> array(2) { [""""]=> int(8)
["ALL"]=> string(1) "N" } [8]=> array(2) { ["~"]=> int(9)
["ALL"]=> string(1) "N" } [9]=> array(2) { [""""]=> int(10)
["ALL"]=> string(1) "N" } [10]=> array(2) { ["AND"]=>
int(11) ["ALL"]=> string(1) "N" } [11]=> array(2) {
["user_pass"]=> int(12) ["ALL"]=> string(1) "N" } [12]=>
array(2) { ["="]=> int(13) ["ALL"]=> string(1) "N" }
[13]=> array(2) { [""""]=> int(14) ["ALL"]=> string(1) "N" }
[14]=> array(2) { ["~"]=> int(15) ["ALL"]=> string(1) "N"
} [15]=> array(2) { [""""]=> int(15) ["ALL"]=> string(1) "N"
} }
<<>>:
```

(a) Query model for legitimate input

Query Model

```
array(19) { [0]=> array(2) { ["SELECT"]=> int(1) ["ALL"]=> string(1)
"N" } [1]=> array(2) { ["user_id"]=> int(2) ["ALL"]=> string(1) "N" }
[2]=> array(2) { ["FROM"]=> int(3) ["ALL"]=> string(1) "N" } [3]=>
array(2) { ["user_detail"]=> int(4) ["ALL"]=> string(1) "N" } [4]=>
array(2) { ["WHERE"]=> int(5) ["ALL"]=> string(1) "N" } [5]=>
array(2) { ["user_name"]=> int(6) ["ALL"]=> string(1) "N" } [6]=>
array(2) { ["="]=> int(7) ["ALL"]=> string(1) "N" } [7]=> array(2) {
[""""]=> int(8) ["ALL"]=> string(1) "N" } [8]=> array(2) { ["~"]=> int(9)
["ALL"]=> string(1) "N" } [9]=> array(2) { [""""]=> int(10) ["ALL"]=>
string(1) "N" } [10]=> array(2) { ["OR"]=> int(11) ["ALL"]=> string(1)
"N" } [11]=> array(2) { [""""]=> int(12) ["ALL"]=> string(1) "N" }
[12]=> array(2) { ["~"]=> int(13) ["ALL"]=> string(1) "N" } [13]=>
array(2) { [""""]=> int(14) ["ALL"]=> string(1) "N" } [14]=> array(2) {
["="]=> int(15) ["ALL"]=> string(1) "N" } [15]=> array(2) { [""""]=>
int(16) ["ALL"]=> string(1) "N" } [16]=> array(2) { ["~"]=> int(17)
["ALL"]=> string(1) "N" } [17]=> array(2) { [""""]=> int(18) ["ALL"]=>
string(1) "N" } [18]=> array(2) {
["#'~'c8e6c3a0e74b35e5f061a0c4db7dcec2bc9ab0fa'"]=> int(18)
["ALL"]=> string(1) "N" } }
<<>>:
```

(b) Query model for 'or 1=1 #

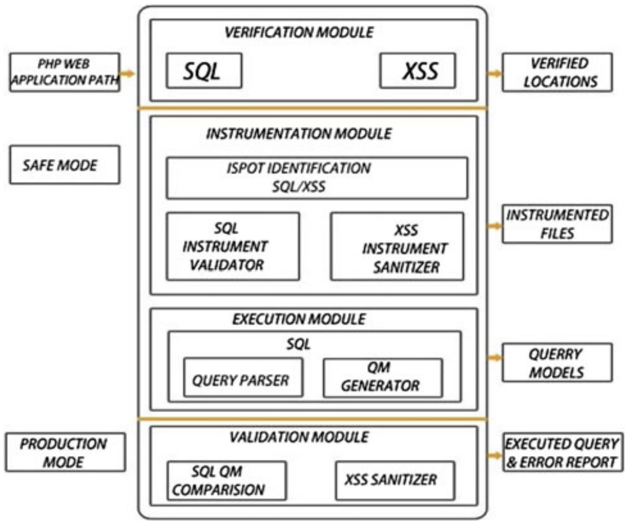


Fig. 4 Basic modules of the integrated tool

- Validation module

This module is called in production mode wherein, the actual inputs are validated against the static query model generated in safe mode using a validation() function.

For SQL Injection, query_validate() function is executed to validate the dynamically generated query model generated by accept() function to compare against the query model generated in the safe mode. Depending on the results of validation, access to the database is allowed/restricted.

For XSS Attack, all input entries point like text boxes and text areas etc. are validated by sanitizer() function to sanitize the real time inputs.

7 Result and analysis

To validate our approach, we have taken a proof-of concept perspective about making an application secure from SQL-injections and reflected cross site scripting attacks. Our objective is to test the proposed integrated model for various kinds of SQL injections and reflected XSS attacks for the applications with different complexities.

The Tables 1 and 2 shows various open web applications that are tested against this integrated tool to prevent SQL injection and reflected cross site scripting attacks. Table 1 shows the detailed analysis for XSS attacks and Table 2 shows the detailed analysis for SQL injection attack.

7.1 Analysis

We have analysed from the results that the proposed integrated model covers many kinds of SQL injection and XSS attacks. The effectiveness of the proposed approach is determined by the ratio of the number of attacks prevented to the total number of attacks performed. Results show that the proposed approach is 100 % effective to prevent SQL injection and reflected cross side scripting attacks. However, this analysis is performed for known attacks only. Hence the robustness of the solution need to be tested for newer types of SQL injection and reflected XSS attacks. Further, strength of solution need to be tested against more complex and exhaustive applications such as websites containing richer services and features.

8 Risk management and insurance

The current trend indicates that cyber insurance is becoming an important component of overall Risk Management and Software Assurance. Cyber insurance covers certain consequences of identified risks such as security breaches, programming errors and business interruptions (ENISA 2012). Cyber insurance also encourages adoption of best practices by the organizations (Clinton L) seeking the insurance by offering lower premiums to them. An organization can demonstrate that after deploying this

Table 1 Detailed execution analysis results for proposed model for XSS attack

Application	Lines of code (K)	Ispot instrumented for XSS	Detection rate (%)	Instrumentation overhead (%)	False positive	Query execution overhead (%)
Online bookstore	4.3	26	100	7	13	1.23
Matrimonial	3.7	23	100	5	11	1.10
Student portal	8.2	30	100	6	18	2.03
Travel portal	9.9	40	100	10	20	2.04

Table 2 Detailed execution analysis results for proposed model for SQL attack

Application	Lines of code (K)	Ispot instrumented for SQL injection attack	Detection rate (%)	Instrumentation overhead (%)	False positive	Query execution overhead (%)
Online bookstore	4.3	48	100	14	13	1.98
Matrimonial	3.7	38	100	10	11	1.56
Student portal	8.2	42	100	13	18	1.99
Travel portal	9.9	34	100	8	20	2.09

model, the exposure to exploitation of code level vulnerabilities will be greatly reduced and risk of compromise of the web application secured by this approach is relatively less, particularly through SQL injection attacks and cross site scripting attacks. As such, the security enhancements and methodologies such as proposed in the paper will help organizations in articulating better stake for lower premiums for cyber insurance policies.

9 Conclusion and future work

In this paper we proposed a model to prevent SQL injection and reflected cross site scripting attacks in pHP web applications. This integration modifies the existing model based hybrid approach to prevent SQL injection attacks which are not prevented by existing hybrid approach and extended to prevent reflected cross site scripting attacks.

The technique utilizes the logic of constructing query model along with a checking of different patterns (such as ‘OR pattern) in user input fields and if any such patterns are encountered then the authorization is not granted and appropriate error messages are generated.

For reflected cross site scripting attack, an algorithm has been proposed that puts a check on all the entry points to sanitize the input and if any of the patterns is matched with the blacklist of vulnerable patterns then input is blocked and appropriate error messages are generated.

The strength of this solution and approach need to be further improved by considering more complex web applications both in terms of size and features involving complex databases and associated inputs.

Also, keeping in view of the effectiveness of this model in preventing certain types of Reflected XSS attacks, the model could be further extended to prevent other types of XSS attacks such as ‘Stored XSS.

Further, the solution could also be extended further to prevent other web attacks such as File Inclusion (Remote and Local), as well as various other vulnerabilities classified by OWASP (2010). Also, attempts could be made to prevent these vulnerabilities and attacks in various other scripting languages such as ASP.NET and PERL.

Acknowledgments Authors wish to express sincere gratitude to the administration of Department of Electronics & Information Technology, Government of India and GGSIP University for providing the academic environment to pursue research activities. In particular we would like to thank Dr. Gulshan Rai, DG, CERT-In, Mr. B.J Srinath, Scientist ‘G’ and Mr. A. S. Chawla, Scientist ‘F’, CERT-In for guidance and inputs.

References

Clinton L (undated) Cyber-insurance metrics and impact on cybersecurity. http://www.whitehouse.gov/files/documents/cyber/ISA-Cyber-InsuranceMetricandImpactonCyber-Security.pdf. Accessed Aug 2012

Common Weakness Enumeration (2012) CWE-89: improper neutralization of special elements used in an SQL command (‘SQL injection’). http://cwe.mitre.org/data/definitions/89.html. Accessed 13 Jan 2012

Common Weakness Enumeration (2012) CWE-79: improper neutralization of input during web page generation (‘cross-site scripting’). http://cwe.mitre.org/data/definitions/79.html. Accessed 13 Jan 2012

ENISA (2012) Incentives and barriers of the cyber insurance market in Europe June 2012. European Network and Information Security Agency (ENISA), Heraklion. http://www.enisa.europa.eu. Accessed 5 July 2012

Gould C, Su Z, Devanbu P (2044) Java database connectivity (JDBC) checker: a static analysis tool for SQL/JDBC applications. In: Proceedings of the 26th international conference on software engineering (ICSE 2004) formal demos, p 697–698. ICSE, Minneapolis

Halfond W, Orso A (2005) AMNESIA: analysis and monitoring for neutralizing SQL injection attacks. In: Proceedings on 20th IEEE and ACM international conference automated software engineering, p 174–183. ACM, New York

Jeom-Goo K (2011) Injection attack detection using the removal of SQL query attribute values. In: Information science and applications (ICISA), international conference, p 1–7. Department of Computer Science, Namseoul University, Cheonan, 26–29 April 2011

Junjin M (2009) An approach for SQL injection vulnerability detection. In: Proceedings of the 6th international conference on information technology: new generations, p 1411–1414. IEEE, Las Vegas

Kiezun A, Guo PJ, Jayaraman K, Ernst MD (2009) Automatic creation of SQL injection and cross-site scripting attacks. In: ARDILLA, proceedings of the 31st international conference on software engineering, p 199–209. ICSE, Vancouver

Kunal S, Mohandas R, Pais AR (2011) Model based hybrid approach to prevent SQL injection attacks in PHP. InfoSecHiComNet'11 proceedings of the first international conference on security aspects of information technology, p 3–15. Springer, Berlin

Open Web Application Security Project (OWASP) (2012) Top 10 2010-main. https://www.owasp.org/index.php/Top_10_2010-Main. Accessed 13 Jan 2012

Prithvi B, Madhusudan P, Venkatakrishnan VN (2010) CANDID: dynamic candidate evaluations for automatic prevention of SQL injection attacks. ACM Trans Inf Syst Secur 13(2):1–39

Qianjie Z, Chen H, Sun J (2010) An execution-flow based method for detecting cross-site scripting attacks. In: Proceedings of software engineering and data mining, p 160–165. SEDM, Shanghai

Rahul J, Sharma P (2012) Survey on web application vulnerabilities (SQLIA,XSS) exploitation and security engine for SQL injection. In: Proceedings on CSNT 2012 IEEE international conference (978-0-7695-4692-6/1). IEEE, Washington, DC

Raju H, Cortesi A (2010) Obfuscation-based analysis of SQL injection attacks. In: ISCC '10 proceedings of the IEEE symposium on computers and communications, p 931–938. IEEE, Riccione

Rattipong P, Bunyatnoparat P (2011) Protecting cookies from reflected cross sitescript attacks using dynamic cookies rewriting technique. In: Method for detecting cross-site scripting attacks. 13th International conference on advanced communication technology (ICACT), p 1090–1094. IEEE Conference Publications, 13–16 Feb 2011

Stephen WB, Keromytis AD (2004) SQLrand: preventing SQL injection attacks. In: Proceedings of the 2nd applied cryptography and network security (ACNS) conference, p 292–302. ACNS, Yellow Mountain

Stephen T, Williams L, Xie T (2009) On automated prepared statement generation to remove SQL injection vulnerabilities. Department of Computer Science, North Carolina State University, Raleigh