

EFFICIENT DESIGN OF STATIC ANALYSIS TOOL FOR DETECTING WEB VULNERABILITIES

**A N L KUMAR¹, P L N RAJU², SANGAPU VENKATA APPAJI³, D
RATNA GIRI⁴**

¹HOD & Assoc.Professor, Department of Computer Applications, Swarnandhra College of Engineering & Technology, Narsapur, AP, India – 534280

²Assistant Professor, Department of MCA, MVGR College of Engineering, Sangivalasa, Vizianagaram, AP, India.

³Assistant Professor, Department of IT, Gokaraju Ranja Raju Institute of Engineering and Technology, Bachupally, Hyderabad, AP, India.

⁴Assistant Professor, Department of IT, SRKR Engineering College, Bhimavaram, West Godavari, AP, INDIA.

Abstract :

The number and the importance of web applications have increased rapidly over the last years. At the same time, the quantity and impact of security vulnerabilities in such applications have grown as well. Since manual code reviews are time-consuming, error prone and costly, the need for automated solutions has become evident. Many web applications written in ASP suffer from injection vulnerabilities, and static analysis makes it possible to track down these vulnerabilities before they are exposed on the web. In this paper, we address the problem of vulnerable web applications by means of static source code analysis. To this end, we propose a new technique to detect XSS attacks and SQL injection vulnerabilities based on taint analysis. It tracks various kinds of external input, tags taint types, constructing control flow graph is constructed based on the use of data flow analysis of the relevant information, taint data propagate to various kinds of vulnerability functions, and detect the XSS or SQL Injection vulnerability in web application's source code. Results show the benefits of the tool in identifying potential security vulnerabilities.

Keywords: W3; Vulnerabilities; SQL Injection; XSS.

1. Introduction

Web applications have become one of the most important communication channels between various kinds of service providers and clients on the Internet. Along with the increased importance of web applications, the negative impact of security flaws in such applications has grown as well. Vulnerabilities that may lead to the compromise of sensitive information are being reported continuously, and the costs of the resulting damages are increasing. The main reasons for this phenomenon are time constraints, limited programming skills, and lack of security awareness on part of the developers. However, most public web hosting services do not enforce any kind of quality assurance on the applications that they run, which can leave a web server open to attacks from the outside[1].

Poorly written web applications are highly vulnerable to attacks because of their easy accessibility on the internet. One careless line of program code could potentially bring down an entire computer network. The reasons for the increase of threats in Web application [2,3] could be divided into two main parts: On one hand, software are developing in too large a scale together with the expanding complexity and extensibility of software while flaws still exist in their source codes; On the other hand, This is probably due to ease of detection and exploitation of web vulnerabilities, combined with the proliferation of low-grade software applications. At the moment, the overflow of Web application programs and Plug-in lead to the result that much of the code is alpha or beta, written by inexperienced programmers with easy-to learn languages such as ASP (Active Server Pages).

1.1. Causes of Vulnerabilities

Of all vulnerabilities identified in Web applications, problems caused by *unchecked input* are recognized as being the most common [41]. To exploit unchecked input, an attacker needs to achieve two goals:

Inject malicious data into Web applications. Common methods used include:

- **Parameter tampering:** pass specially crafted malicious values in fields of HTML forms.

- **URL manipulation:** use specially crafted parameters to be submitted to the Web application as part of the URL.
 - **Hidden field manipulation:** set hidden fields of HTML forms in Web pages to malicious values.
 - **HTTP header tampering:** manipulate parts of HTTP requests sent to the application.
 - **Cookie poisoning:** place malicious data in cookies, small files sent to Web-based applications.
- Manipulate applications using malicious data.** Common methods used include:
- **SQL injection:** pass input containing SQL commands to a database server for execution.
 - **Cross-site scripting:** exploit applications that output unchecked input verbatim to trick the user into executing malicious scripts.
 - **HTTP response splitting:** exploit applications that output input verbatim to perform Web page defacements or Web cache poisoning attacks.
 - **Path traversal:** exploit unchecked user input to control which files are accessed on the server.
 - **Command injection:** exploit user input to execute shell commands.

2. Related Work

Security vulnerabilities in web applications may result in stealing of confidential data, breaking of data integrity or affect web application availability. Thus the task of securing web applications is one of the most urgent for now: according to Acunetix survey [4] 60% of found vulnerabilities affect web applications. The most common way of securing web applications is searching and eliminating vulnerabilities therein. Examples of another ways of securing web application include safe development [5-7], implementing intrusion detection and/or protection systems [8-10], and web application firewalls [11].

According to OWASP [12], the most efficient way of finding security vulnerabilities in web applications is manual code review. This technique is very time-consuming, requires expert skills, and is prone to overlooked errors. Therefore, security society actively develops automated approaches to finding security vulnerabilities. These approaches can be divided into two wide categories: black-box and white-box testing.

The first approach is based on web application analysis from the user side, assuming that source code of an application is not available [13]. The idea is to submit various malicious patterns (implementing for example SQL injection or cross-site scripting attacks) into web application forms and to analyze its output thereafter. If any application errors are observed an assumption of possible vulnerability is made. This approach does guarantee neither accuracy nor completeness of the obtained results.

The second approach is based on web application analysis from the server side, with assumption that source code of the application is available. In this case dynamic or static analysis techniques can be applied. A comprehensive survey of these techniques was made [14] by Vigna et al. According to this survey several statements could be made:

- The most common model of input validation vulnerabilities is the Tainted Mode model. This model was implemented both by means of static [15-17] or dynamic analysis [18-21].
- Another approach to model input validation vulnerabilities is to model syntactic structure for sensitive operations arguments. The idea behind this is that the web application is susceptible to an injection attack, if syntactic structure for sensitive operation arguments depends on the user input. This approach was implemented by means of string analysis in static [22, 23] and it was applied to detect SQLI [24] and XSS [25] vulnerabilities in PHP. After all, this approach was implemented to detect injection attacks at runtime.
- One of the main drawbacks of static analysis in general is its susceptibility to false positives [26] caused by inevitable analysis imprecisions. This is made worse by dynamic nature of scripting languages. However, static analysis techniques normally perform conservative analysis that considers every possible control path.
- One of the contrary, one of the main drawbacks of dynamic analysis is that it is performed on executed paths and does not give any guarantee about paths not covered during a given execution. However, dynamic analysis having access to internals of web application execution process has the potential of being more precise.

3. System Architecture

A web application, as the name implies, is a computer application that is accessed through a web-based user interface. This is typically implemented through a client-server setup, with the server running an HTTP server software package (such as Apache or Microsoft IIS) capable of generating dynamic web pages, while the client

communicates with the server through a web browser (such as Microsoft Internet Explorer or Mozilla Firefox). The working of such a web application is roughly sketched in Figure 1.

Whenever the client interacts with the server, communication takes place in the form of HTTP requests. The client sends a request (typically a GET or POST request) to the server, along with a series of parameters (step 1 in Fig. 1). The HTTP server recognizes that this is a request for a dynamic web page, in this case a page that is to be generated by a ASP script. It fetches the corresponding ASP script from the web server's file system (step 2) and sends it off to be processed by the integrated ASP interpreter (step 3). The ASP interpreter then executes the ASP script, making use of external resources whenever necessary. External resources can for example be a database (steps 4 and 5), but also the file system or an API exposed by the operating system. The executed ASP script typically produces output in the form of an HTML page, which is sent back to the client and displayed in the web browser (step 6).

The most prevalent and most exploited vulnerabilities in web applications are cross-site scripting (XSS) and SQL injection (SQLI). According to a top ten composed by the Open Web Application Security Project (OWASP), XSS and SQLI were the top two most serious web application security flaws for both 2007 and 2010 [27]. According to this list, the top five of security flaws has not changed over the past three years. Vulnerabilities occurring in real-world applications are much more complicated and subtle than the examples appearing in this section. In many cases, user data is utilized in applications in ways that appear to be safe on the surface. However, due to complex interactions that is difficult to predict, unsafe data can still slip through in specific edge cases. Such vulnerabilities are hard to spot, even when using professional coding standards, careful code reviewing, and extensive testing.

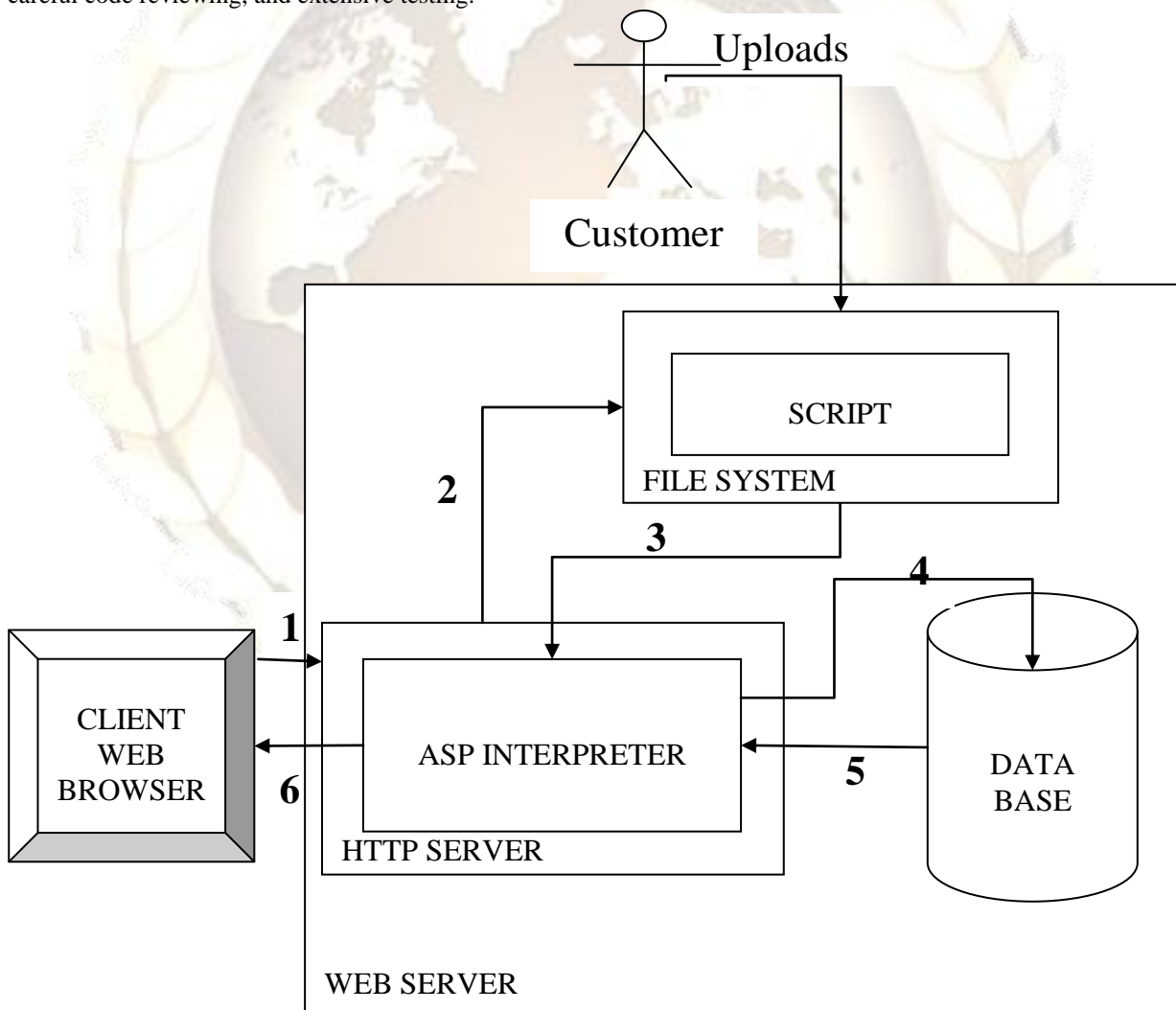


Fig 1 System architecture

3.1. Cross-Site Scripting

Cross-site scripting (XSS) is a type of vulnerability that allows attackers to inject unauthorized code into a web page, which is interpreted and executed by the user's web browser. XSS has been the number one web application vulnerability for many years, and according to White Hat Security, has been responsible for 66% of all website attacks in 2009 [28].

Web pages can include dynamic code written in JavaScript to allow the web page's content to be altered within the web browser as the user interacts with it. Normally, a web browser will only execute JavaScript code that originates from the same domain as the web page itself, and that code is only executed within a self-contained sandbox environment. This is the so-called Same Origin Policy [29]. This policy prevents attackers from making web browsers execute un-trusted code from an arbitrary location (Table 1 shows the XSS vulnerability example).

Table 1 Example of an XSS vulnerability

1	<html>
2	<body>
3	<%
4	'R e t r i e v e the user ' s name from a form
5	name = Request.Response('name')
6	// P r i n t the user ' s name back to them
7	Response.write "Hello there , \$name! How are you doing?"
8	%>
9	</body>
	</html>

3.2. SQL Injection

SQL injection is a taint-style vulnerability, whereby an unsafe call to a database is abused to perform operations on the database that were not intended by the programmer. White Hat Security's report for 2009 lists SQL injection as responsible for 18% of all web attacks, but mentions that they are under-represented in this list because SQL injection flaws can be difficult to detect in scans [28].

SQL, or Structured Query Language, is a computer language specially designed to store and retrieve data in a database. Most database systems (e.g. MySQL, Oracle, Microsoft SQL Server, SQLite) use a dialect of SQL as a method to interact with the contents of the database. Scripting languages such as PHP offer an interface for programmers to dynamically construct and execute SQL queries on a database from within their program.

It is common practice for programmers to construct dynamic database queries by means of string concatenation. Listing 2 shows an example of an SQL query that is constructed and executed from a PHP script using this method. The variable \$username is copied directly from the input supplied by the user and is pasted into the SQL query without modifications. Although this query is constructed and executed on the server and users can not directly see the vulnerable code, it is possible through a few assumptions and educated guesses to find out that there is a database table called users and that the entered user name

The test method of common SQL Injection attack is the use “'”, “union”, “--;” and so on key words, in test dynamic SQL sentence in program whether to exist injection vulnerability. For example, consider the login page of a web application that expects a user-name and the corresponding password. When the credentials are submitted, they are inserted within a query template such as the following:

```
“select * from admin where username =” + request.form(“username”) + “ and Password = ” + request.form(“passwd”)+“”
```

Instead of a valid user name, the malicious user sets the “username” variable to the string: ' or 1=1; - -', causing the Vbscript to submit the following SQL query to the database:

```
“select * from admin where username = ' or 1=1; - -' and Password = ' any_passwd' ”
```

Therefore, the password value is irrelevant and may be set to any character string. The result set of the query contains at least one record, since the “where” clause evaluates to true. If the application identifies a valid user by testing whether the result set is non-empty, the attacker can bypass the security check.

4. Static Analysis Methods

Detection software security vulnerabilities are mainly dynamic analysis, formal method validation and static analysis. Static analysis is divided as type inference, data flow analysis and constraints analysis [1,30,31].

4.1. Type Inference

Type system of programming language concludes type definition and rules for type equivalence, type inclusiveness and type dedication. Type dedication is to derive the types of variables and methods within a program automatically so as to determine whether or not their visit meet these type rules. This kind of dedication could be used to examine the bug in types and conduct necessary type transmission with proper operations. It boasts the characteristics of simplicity and high efficiency which makes it perfect for quick detection of security threats in software. Now it is mainly applied in detection of format string vulnerability, OS kernel vulnerable pointer use.

4.2. Data-Flow Analysis

Data-flow analysis is used in the process programming, which collect semantic information from programs and then define and use the variables with algebraic approach. It is used in program optimization, program validation, debugging, parallel, Vectorization and serial program environment. Its realization makes use of the pair "variable definition-quoting".

4.3. Constraint Analysis

Constraint analysis divides program analysis into constraint generation and constraint solution. The former constructs variable type with constraint generation rules or analyses constraint system among statuses. While the later solve such constraint systems. Constraint system is comprised of equation constraint, set constraint and incorporate constraint. In the first kind, only equation exists between constraint objects. Set constraint takes program variables as a set of values, whose evaluation is regarded as conclusion relation between set expressions? While the last constraint concludes equation constraint part and set constraint part.

4.4. The Comparison among three Main methods

The three main methods mentioned are all explain the abstract semantics of programs and construct mathematic models based on the program property, with which they determine the property of the program. In comparison, constraint analysis boasts the greatest ability in detection while the lowest speed of that, which makes it fit for security examination of software, data-flow analysis has relatively high speed and remarkable ability of detection which is appropriate in static analysis which should take control flow information and requires only simple operation among variable properties; when it comes to type dedication, it has the poorest ability and the fastest speed in examination and suits for security test in finite property domain and unrelated control flow.

The website that issued news or BBS forum is one kind of web application. Analyzing the logic characteristics of its services, it is not complex to find out that the process of dataflow could be give a summaries: data input (parameters) → data service processing (web server)→result output (HTML).

Based on principles of XSS threats and SQL injection threats, we could see vulnerability is mainly generated from the sanitation process of input data. Thus sanitation process of all the input data would neglect such vulnerability of taint data (outside client input data). In this paper, we would examine the code with data flow analysis; the function framework of system is shown in fig 2.

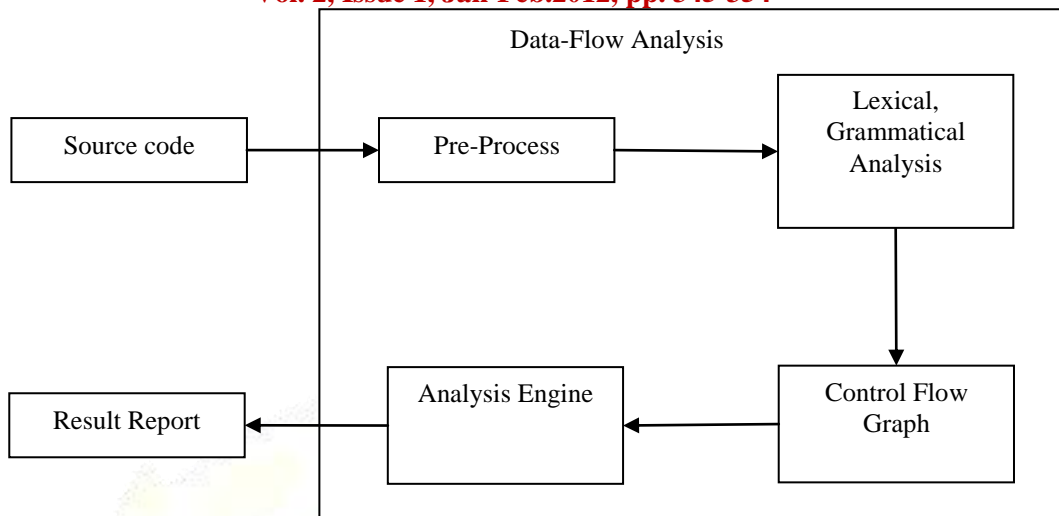


Fig 2. Function Framework of Code Review System

5. ASP VULNERABILITIES

ASP is a computer scripting language. Originally designed for producing dynamic web pages, it has evolved to include a command line interface capability and can be used in standalone graphical applications. PHP is a widely used general-purpose scripting language that is especially suited for web development.

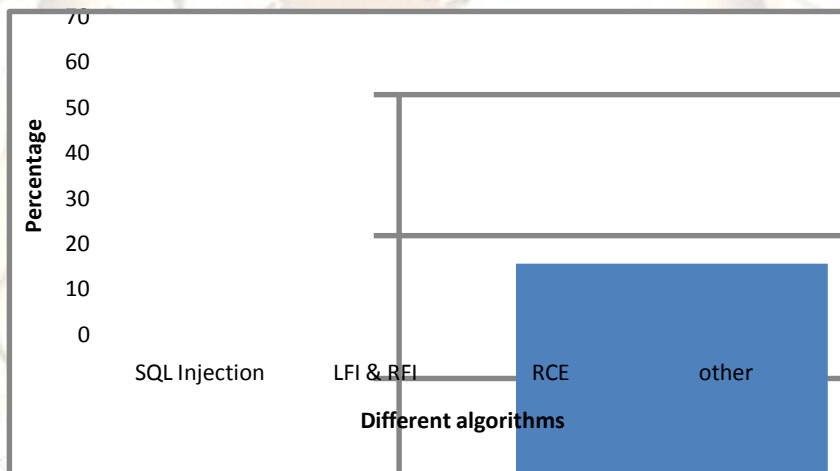


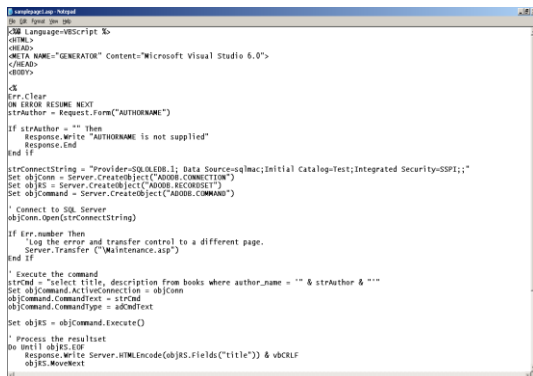
Fig 3 Percent of different vulnerabilities exploits released during 2007 and 2008

Several vulnerabilities and a weaknesses have been reported in PHP, where some have unknown impacts and others can be exploited by malicious people to disclose potentially sensitive information, bypass certain security restrictions. Among the most common of them are RFI, LFI, XSS, SQL Injection and RCE. According securityfocus[17] 60% of released exploits are on web applications. We obtained some static from Milw0rm [18] which indicates 27% of released exploits are caused by LFI and RFI vulnerability. Fig. 3 shows a statistical analysis of released exploits from October 2007 to February 2008 years.

6. IMPLEMENTATION

To test the validity of our approach, we select three open source program written by ASP. These software are commonly used a source codes of tools in network application layer and are representative. Test environment: Intel XEON CPU: 3.0GHz, 1GB cache, Windows 2003 Server, IIS6.0. Then we conduct penetrating examination in Acunetix Web with the results shown in Fig 4. We develop a tools named as ASPWC. The number of XSS reported by Acunetix Web tools is success of the XSS attack test.

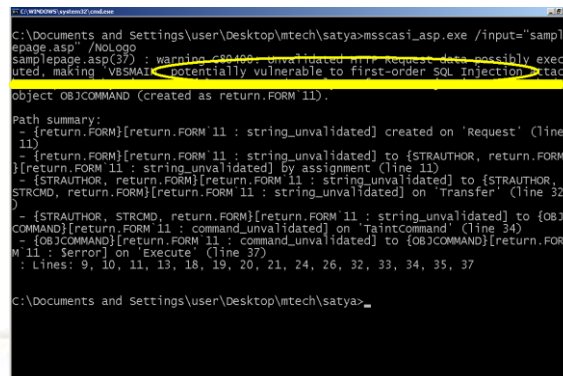
Possible SQL Injection vulnerability through data that is read from the Request object without any input validation. These warnings are very likely bugs that must be fixed. Sample Web Page is shown in Fig 4



```
...
<!-- Language=VBScript %>
<!--#VB
META NAME="GENERATOR" Content="Microsoft Visual Studio 6.0">
</HEAD>
<BODY>
<C
Err.Clear
ON ERROR RESUME NEXT
strAuthor = Request.Form("AUTHORNAME")
If strAuthor = "" Then
Response.Write "AUTHORNAME is not supplied"
Response.End
End If
strConnectionString = "Provider=SQLNCLI; Data Source=sqlmac;Initial Catalog=test;Integrated Security=SSPI;"
Set objConn = Server.CreateObject("ADODB.CONNECTION")
Set objRS = Server.CreateObject("ADODB.RECORDSET")
Set objCommand = Server.CreateObject("ADODB.COMMAND")
Connect to SQL Server
objConn.Open(strConnectionString)
If Err.number Then
Log the error and transfer control to a different page.
Server.Transfer ("Maintenance.asp")
End If
Execute the command
strCmd = "select title, description from books where author_name = '" & strAuthor & "'"
Set objCommand.ActiveConnection = objConn
objCommand.CommandText = strCmd
objCommand.CommandType = adCmdText
Set objRS = objCommand.Execute()
Set objRS = objCommand.Execute()
Process the resultset
Do Until objRS.EOF
Response.Write Server.HtmlEncode(objRS.Fields("title")) & vbCRLF
objRS.MoveNext
...

```

Fig 4 Sample web page



```
Path summary:
- {return.FORM}[return.FORM 11 : string_unvalidated] created on 'Request' (Line 11)
- {return.FORM}[return.FORM 11 : string_unvalidated] to {STRAUTHOR, return.FORM}
{return.FORM 11 : string_unvalidated} by assignment (line 11)
- {STRAUTHOR, return.FORM}[return.FORM 11 : string_unvalidated] to {STRAUTHOR,
STRCMD, return.FORM}[return.FORM 11 : string_unvalidated] on 'Transfer' (line 32)
- {STRAUTHOR, STRCMD, return.FORM}[return.FORM 11 : string_unvalidated] to {OBJ
COMMAND}[return.FORM 11 : command_unvalidated] on 'Taintcommand' (line 34)
- {OBJCOMMAND}[return.FORM 11 : command_unvalidated] to {OBJCOMMAND}[return.FOR
M 11 : Serror] on 'Execute' (line 37)
: Lines: 9, 10, 11, 13, 18, 19, 20, 21, 24, 26, 32, 33, 34, 35, 37
c:\Documents and settings\user\Desktop\mtech\satya>_

```

Fig 5 Output from our tool

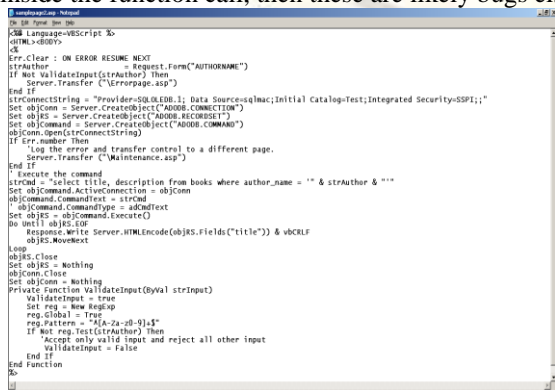
Comments: strAuthor is assigned a value from Request.QueryString(“AUTHORNAME”) on line number 11 and is eventually used in the construction of dynamic SQL and executed through OBJCOMMAND on line number 37 Use parameterized SQL query to mitigate the SQL Injection identified by the tool(Table 2 shows the sample code).

Table 2 Sample Code

```
' Execute the command
strCmd = "select title, description from books where author_name = ?"
Set objCommand.ActiveConnection = objConn
objCommand.CommandText = strCmd
objCommand.CommandType = adCmdText
Set param1 = objCommand.CreateParameter ("author", adWChar,
adParamInput, 50)
param1.value = strAuthor
objCommand.Parameters.Append param1

```

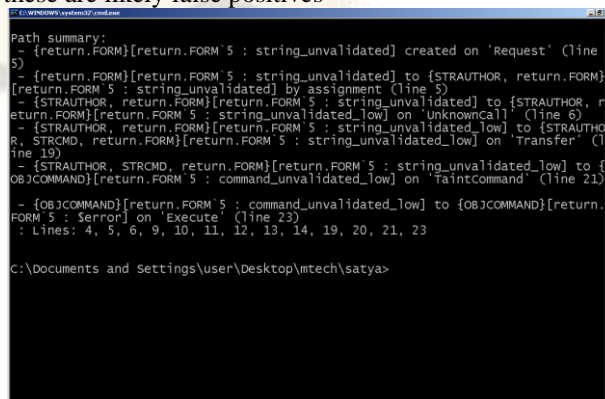
Possible SQL Injection vulnerability through data that is read from the Request object where the input is passed through some unknown function calls that might perform data validation. If there is no data validation done inside the function call, then these are likely bugs else these are likely false positives



```
...
Err.Clear : ON ERROR RESUME NEXT
strAuthor = Request.Form("AUTHORNAME")
If Not ValidateInput(strAuthor) Then
Server.Transfer ("Errorpage.asp")
End If
strConnectionString = "Provider=SQLNCLI; Data Source=sqlmac;Initial Catalog=test;Integrated Security=SSPI;"
Set objConn = Server.CreateObject("ADODB.CONNECTION")
Set objRS = Server.CreateObject("ADODB.RECORDSET")
Set objCommand = Server.CreateObject("ADODB.COMMAND")
objConn.Open(strConnectionString)
If Err.number Then
Log the error and transfer control to a different page.
Server.Transfer ("Maintenance.asp")
End If
Execute the command
strCmd = "select title, description from books where author_name = '" & strAuthor & "'"
Set objCommand.ActiveConnection = objConn
Set objCommand.CommandText = strCmd
objCommand.CommandType = adCmdText
Set objRS = objCommand.Execute()
Do Until objRS.EOF
Response.Write Server.HtmlEncode(objRS.Fields("title")) & vbCRLF
objRS.MoveNext
objRS.Close
Set objRS = Nothing
objConn.Close
Set objConn = Nothing
Private Function ValidateInput(ByVal strInput)
ValidateInput = True
Set req = New RegExp
req.Global = True
req.Pattern = "[a-zA-z0-9]*"
If Not req.Test(strAuthor) Then
'Accept only valid input and reject all other input
ValidateInput = False
End If
End Function
...

```

Fig 6 Sample web page



```
Path summary:
- {return.FORM}[return.FORM 5 : string_unvalidated] created on 'Request' (line 5)
- {return.FORM}[return.FORM 5 : string_unvalidated] to {STRAUTHOR, return.FORM}
{return.FORM 5 : string_unvalidated} by assignment (line 5)
- {STRAUTHOR, return.FORM}[return.FORM 5 : string_unvalidated] to {STRAUTHOR,
return.FORM}[return.FORM 5 : string_unvalidated_low] on 'UnknownCall' (line 6)
- {STRAUTHOR, return.FORM}[return.FORM 5 : string_unvalidated_low] to {STRAUTHOR,
STRCMD, return.FORM}[return.FORM 5 : string_unvalidated_low] on 'Transfer' (line 19)
- {STRAUTHOR, STRCMD, return.FORM}[return.FORM 5 : string_unvalidated_low] to {
OBJCOMMAND}[return.FORM 5 : command_unvalidated_low] on 'Taintcommand' (line 21)
- {OBJCOMMAND}[return.FORM 5 : command_unvalidated_low] to {OBJCOMMAND}[return.
FORM 5 : Serror] on 'Execute' (line 23)
: Lines: 4, 5, 6, 9, 10, 11, 12, 13, 14, 19, 20, 21, 23
c:\Documents and settings\user\Desktop\mtech\satya>

```

Fig 7 Output from our tool

The experimental results analysis: experimental data can be seen from the above, based on control flow graph; data-flow analysis of the vulnerability detection algorithm can be effectively used to detect XSS, SQL injection vulnerabilities which exist in the source code. The blacklist is applied to check the input data in OK3W and Leichinews program. They have a common function to check all input string. The programs produce a certain false positive. Despite the weaknesses found in the report contains false positives, reporting the total number of articles, or less, from the relatively small number of reports of these find the true vulnerabilities have been greatly reduced the workload.

7. RESULTS

In this section, we summarize the experiments we performed and describe the security violations we found. To evaluate our method, we developed a ASP application based on our algorithm and it was run on four randomly selected open source PHP programs from 100 application on which LFI and RFI have been reported, and TIKIWIKI (which is one of popular and famous ASP application), to demonstrate the feasibility of the proposed algorithm. After that, the results were compared with the output of DAPHPS can method and put in table which you could see it as Table 3. Fig 8 show that different application versions with vulnerability detection rate in percentage. Fig 9 show that number of files with vulnerability detection rate in percentage. Fig 10 shows that different application versions with vulnerability detection rate in percentage number files as well different algorithms.

Table 3 Results of applying our algorithm to set of programs

Application Name&Version	Number of Files & Folders	Number of registered Vulnerabilities	Number of RFI and LFI otential	Correct Report	FP	Detection rate
Tikiwiki 1.9.8	1408 & 329	2	94	2	98%	100%
Scwiki Beta2	111 & 25	1	47	3	94%	60%
Php help agent 1.1	49 & 7	1	7	2	71%	100%
Phportal 1.2	35 & 100	3	190	4	98%	50%
Pragyan 2.6.2	141 & 163	1	107	5	95%	100%

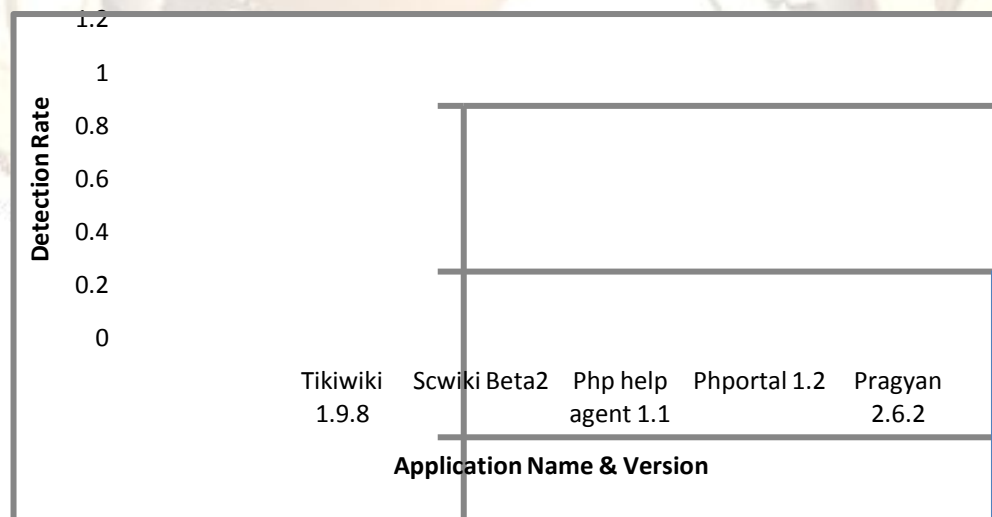


Fig 8 Application VS Detection rate

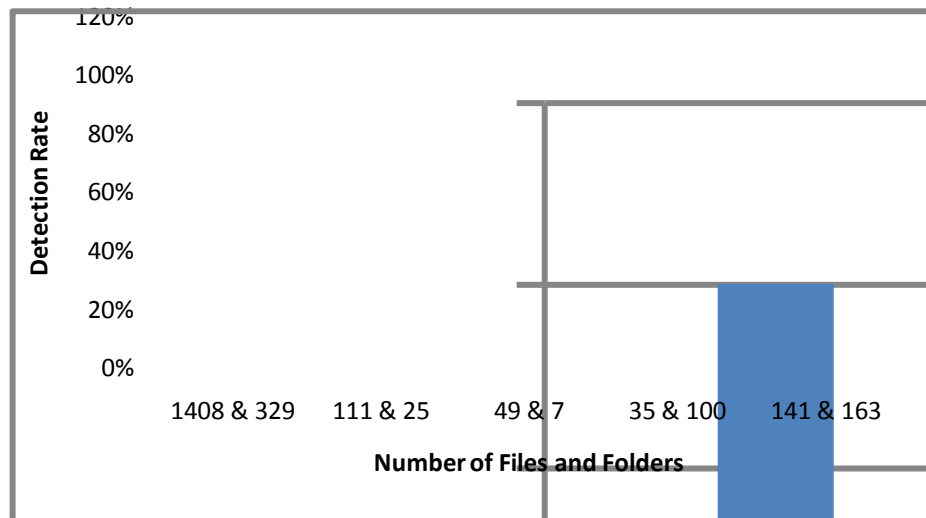


Fig 9 No. of files VS Detection rate

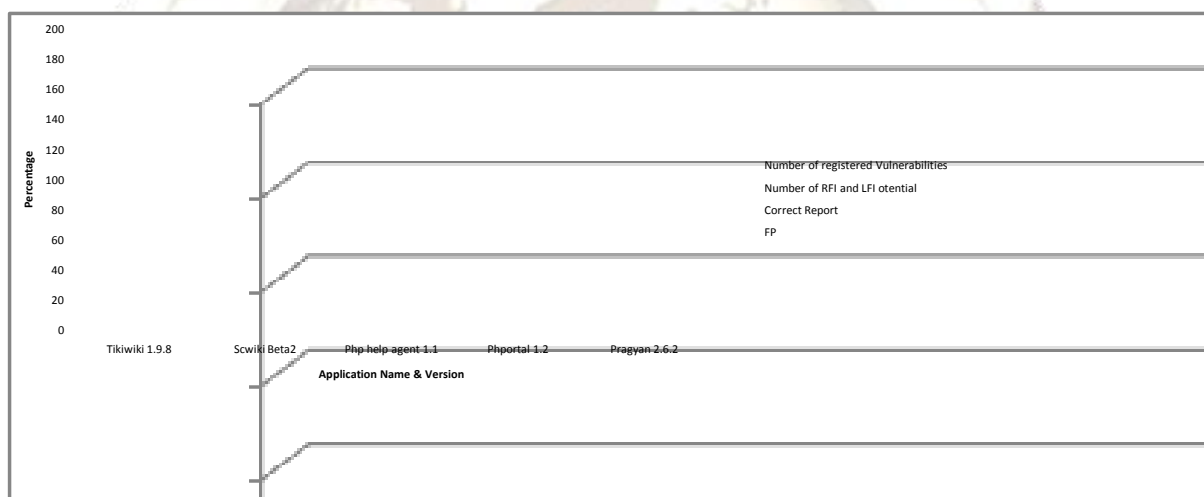


Fig 10 Application with different percentages

8. Conclusions

Software needs to be secure in order to allow parties of different trust levels to interact with each other, without risking that un-trusted parties exploit critical parts of the software. Web applications are mostly susceptible to input validation vulnerabilities, also known as taint-style vulnerabilities, the most common of which are cross-site scripting and SQL injection. Because web applications are made to be easily accessible through the internet, they are particularly exposed to attacks. Static analysis allows programmers to look for security problems in their source code, without the need to execute it. Static analysis tools will always produce false positives and false negatives, because they need to make trade-offs between accuracy and speed, and because program analysis is inherently un-decidable. This tool has manifests its usefulness in examining the web sites based on ASP of the virtual host computer in a high school. Despite the fact that fault rate remains high now, we would use data-flow analysis and add rules to detect sensitive information so as to yield higher accuracy of examination in source code and lower false positive amount within an acceptable bound.

FUTURE WORK: Future works include the extension of this algorithm for other common vulnerabilities like SQL injection, XSS and RCE.

References

- [1] Brian Chess, Jacob West (2007). "Secure Programming with Static Analysis". Addison-Wesley. ISBN 978-0321424778.
- [2] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, Chung-Hung Tsai. "Web Application Security Assessment by Fault Injection and Behavior Monitoring". In Proceedings of the Twelfth International Conference on World Wide Web (WWW2003), pages 148-159, May 21-25, Budapest, Hungary, 2003.
- [3] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D.T. Lee, Sy-Yen Kuo. "Verifying Web Applications Using Bounded Model Checking". In Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN2004), pages 199-208, Florence, Italy, Jun 28-Jul 1, 2004.
- [4] Andrews, M.: "The State of Web Security". IEEE Security & Privacy, vol. 4, no. 4, pp. 14-15 (2006).
- [5] Cook, W. R., Rai, S.: "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries". In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), (2005).
- [6] McClure, R., Krüger, I.: "SQL DOM: Compile Time Checking of Dynamic SQL Statements". In: Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pp. 88-96, (2005).
- [7] Meier, J.D., Mackman, A., Wastell, B.: "Threat Modeling Web Applications", Microsoft Corporation, (2005).
- [8] Su, Zh., Wassermann, G.: "The essence of command injection attacks in web applications". In: ACM SIGPLAN Notices, vol. 41, no.1, pp. 372-382 (2006).
- [9] Pietraszek, T., Berghe, C. V.: "Defending against Injection Attacks through Context-Sensitive String Evaluation". In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, (2005).
- [10] Halfond, W., Orso, A.: "AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks". In: Proceedings of the International Conference on Automated Software Engineering, pp. 174-183, (2005).
- [11] Ristic, I.: "Web application firewalls primer". (IN)SECURE, vol. 1, no. 5, pp. 6-10, (2006).
- [12] Curphey, M., Wiesman, A., Van der Stock, A., Stirbei, R.: "A Guide to Building Secure Web Applications and Web Services". OWASP (2005).
- [13] Auronen, L.: "Tool-Based Approach to Assessing Web Application Security". Seminar on Network Security (2002).
- [14] Cova, M., Felmetzger, V., Vigna, G.: "Testing and Analysis of Web Services". Springer (2007).
- [15] Jovanovic, N., Kruegel, Ch., Kirda, E.: "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities".
- [16] Livshits, V., Lam, M.: "Finding security errors in Java programs with static analysis". In: Proceedings of the 14th Usenix Security Symposium (2005).
- [17] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., Kuo, S.-Y.: "Securing web application code by static analysis and runtime protection". In: WWW '04: Proceedings of the 13th International Conference on World Wide Web (2004).
- [18] Ragle, D.: "Introduction to Perl's Taint Mode." <http://www.webreference.com/programming/perl/taint>
- [19] Thomas, D., Fowler, Ch., Hunt, A.: "Programming Ruby: The Pragmatic Programmer's Guide". Addison Wesley Longman, Inc (2001).
- [20] Anh, N.-T., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: "Automatically Hardening Web Applications Using Precise Tainting". IFIP Security Conference (2005).
- [21] Haldar, V., Chandra, D., Franz, M.: "Dynamic Taint Propagation for Java". In: Proceedings of the 21st Annual Computer Security Applications Conference (2005).
- [22] Christensen, A. S., Miller, A., Schwartzbach, M. I.: "Precise analysis of string expressions". In: Proceedings of International Static Analysis Symposium, SAS '03 (2003).
- [23] Gould, C., Su, Z., Devanbu, P.: "Static checking of dynamically generated queries in database applications". In: Proceedings of 26th International Conference on Software Engineering, IEEE Press (2004).
- [24] Wassermann, G., Su, Z.: "Sound and precise analysis of web applications for injection vulnerabilities". In: Proceedings of the SIGPLAN conference on Programming language design and implementation (2007).
- [25] Minamide, Y.: "Static approximation of dynamically generated web pages". In: Proceedings of the 14th International Conference on World Wide Web (2005).
- [26] Chess, B., McGraw, G.: "Static analysis for security". IEEE Security & Privacy, vol. 2, no. 6, pp. 76-79 (2004).
- [27] OWASP: Top Ten Project. http://www.owasp.org/index.php/OWASP_Top_Ten
- [28] The WhiteHat Website Security Statistics Report - 8th Edition - Fall 2009. <http://www.whitehatsec.com/home/resource/stats.html>
- [29] David Scott, Richard Sharp. "Abstracting application-level web security". In Proceedings of the 11th international conference on World Wide Web (WWW2002), pages 396-407, Honolulu, Hawaii, May 17-22, 2002.
- [30] Xia Yiming. Security Vulnerability Detection Study Based on Static Analysis[J]. Computer Science, 2006, 33(10): 279-283.
- [31] Paul Biggar, David Gregg. "Static analysis of dynamic scripting languages". August, 2009