DEFENDING THE SECURITY ATTACKS IN ASP.NET WEB APPLICATIONS

SANDRA SARASAN

Vidya Academy of Science & Technology, Calicut University, Kerala
Email: sandrasarasan002@gmail.com

Abstract— Web application has become one of the most essential communication channels between service providers and users now-a-days. The global distribution of these web applications make them prone to attacks that uncover and maliciously exploit a variety of security vulnerabilities. Research reports indicate that more than 80 percent of the web applications are vulnerable to security threats. Since these applications might contain security leaks that are not seen to the owner of the web applications. In this paper, I present a scanner tool that checks all the ASP.NET (Active Server Pages) web application files for the two top most security attacks: SQL (Structured Query Language) Injection and Cross-Site Scripting (XSS). If leaks occur in the file it will generates a report mentioning the infected file name, leak description and its location in the file. Otherwise it reports the file as secure. This scanner tool helps the developers to fix the vulnerabilities in the application and improve the overall security.

Keywords— ASP.NET, Cross-Site Scripting, SQL Injection, SQL Injection and Cross-Site Scripting Prevention

I. INTRODUCTION

Today majority of the security attacks are caused by string-based code injection which will result in dangerous security flaws. Online data theft has become a serious problem in security of web applications. As huge amount of data is stored in databases connected to some web applications all across the globe, the users of the web application keep on inserting, deleting and updating data in these databases. These web applications maintain the security of data stored in databases, if they are not secured allow well-designed injection to perform unwanted operations on back-end database. This allow the attacker to access confidential data and gain complete control over the database or web application which will result in theft of user sensitive data or destruction of the whole system.

SQLI (Structured Query Language Injection) and Cross-Site Scripting (XSS) are the two top most attacks according to OWASP (Open Web Application Security Project) that has been used frequently to implement the attacks. SQL Injection is a type of injection or attack in a Web application, in which the attacker provides SQL code to a user input box of a Web form to gain information access from databases which results in its loss of confidentiality, integrity and authority. Cross-Site Scripting occurs when a web application uses inputs received from users in web pages without properly checking them. This allows an attacker to inject malicious scripts in web pages which results in account hijacking and cookie theft. XSS usually affects victim's web browser on the client-side where as SQL injection occurs in server side. As a result, the system could bear heavy loss in giving proper services to its users or it may face complete destruction.

II. DISCUSSION

The organization of the paper will start in Section III by briefly discussing categories of security attacks and their different types. In Section IV, we discuss various approaches suggested by different authors to detect and prevent SQL injection and XSS, their workings and limitations. Then in Section V, the implementation details of proposed work including the evaluation parameters are discussed. In Section VI we will put our focus on discussing the performance results of the system and finally in Section VII this paper is concluded.

III. TYPES OF VULNERABILITY

Currently, there are different types of vulnerabilities that vary in terms of complexity, detection and recovery. In my paper it deals with two types of attacks. They are:-

A. SQL Injection Attack

SQL Injection is a method of injecting SQL meta-characters or commands inside web-based input fields so that execution of the backend SQL queries can be manipulated. Web servers belonging to an organization is primary target of these kinds of attack. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database, and recover the content of a given file present on the DBMS (Database Management System) file system and in some cases

issue commands to the operating system. Different SQLI types [1][2] are:-

1) Tautology: This SQLI attack alters the database query by inserting vulnerable SQL tokens into the conditional query statements which always evaluates to true.

Example: SELECT * FROM <tablename> WHERE userId = <id> AND password = <wrongPassword> OR 1=1;

2) Union Queries: This query uses UNION keyword to access the information from other tables in the database. Such queries can be exploited by the attacker to get valuable data from the database.

Example: SELECT * FROM <tablename> WHERE userId
= <id> AND password = <rightPassword> UNION SELECT creditCardNumber FROM CreditCardTable;

3) Piggy-backed Queries: In this attack the hacker appends ';' and a query to a database query to be executed on the database which will result in huge loss of data [3]. It could be one of the dangerous attacks that damage or destroy a table.

Example: SELECT * FROM <tablename> WHERE userId = <id> AND password = <rightPassword>; DROP TABLE <tablename>;

3) Blind Injection: Here the web developers hide the error messages from the user coming from the database so that the user is sent with a generic error displaying page. At this point the attacker sends a set of true/ false questions to steal data.

Example: SELECT name FROM <tablename> WHERE
id=<username> AND 1 = 0 – AND pass = SELECT name FROM <tablename> WHERE id=<username> AND 1 = 1 – AND pass =

Both the queries will return an error message in case the web application is secure, however if the input is not validated then chances of injection exist [5].

B. *Cross-Site Scripting (XSS)*

XSS attacks are those attacks in which an attacker gets control over the user's browser in order to execute a malicious script (usually a HTML (Hyper Text Markup Language)/ JavaScript code) within the context of trust of the web application's site. If the code is successfully executed, the attacker will be able to access the sensitive browser resources associated with the web application (e.g., cookies, session IDs, etc.). Typical input sources that attackers manipulate include HTML forms, cookies, URLs (Uniform Resource Locators), JavaScripts and external files which the browsers interpret could cause XSS. Cross-Site Scripting attacks also work over HTTP (Hyper Text Transfer Protocol) and HTTPS (Hyper Text Transfer Protocol Secure (SSL) Secure Socket Layer) connections [4]. Different types of XSS attacks are:-

1) Persistent/ Stored XSS: The persistent XSS vulnerability is a more devastating variant of a cross-site scripting flaw. It occurs when the data provided by the attacker is saved by the server, and then permanently displayed as 'normal' web pages returned to the users during browsing, without proper HTML escaping.

For example, with online message boards where users are allowed to post HTML formatted messages for other users to read.

2) Non-persistent/ Reflected XSS: This attack targets vulnerabilities that occur in some websites which deals with dynamic web page generation. An attack is successful if it can send code to the server that is included in the web page results sent back to the browser, which may not be encoded using HTML special character encoding, thus being interpreted by the browser rather than being displayed as valid visible text. The attack can be done by using a malformed URL, such that a variable passed in a URL to be displayed on the page contains malicious code.

An example is, the non-persistent XSS vulnerabilities in Google, could allow malicious sites to attack Google users who visit them while logged in.

3) Local or DOM-based XSS: DOM-based flaws targets vulnerabilities within the code of a web page itself. These types of vulnerabilities are the result of incautious use of the Document Object Model in JavaScript so that opening another web page with malicious JavaScript code in it at the same time might alter the code in the first page on the local system. DOM is the standard model for representing HTML or XML contents which is called the Document Object Model (DOM).

Common vulnerabilities that make your web application susceptible to cross-site scripting include:

- Failing to constrain and validate input.
- Failing to encode output.
- Trusting data retrieved from a shared database.

IV. RELATED WORK

A. *SQL Injection Defenses*

Preventing SQL Injection requires keeping untrusted data separate from commands and queries. The preferred option is to use a safe API (Application Program Interface) which avoids the use of the interpreter entirely or provides a parameterized interface. Positive or white list input validation with appropriate canonicalization also helps protect against injection, but is not a complete defense as many applications require special characters in their input [5][6]. Thus SQL Injection defenses are broadly classified into of two types:-

1) Defensive coding: The use of proper coding practices is a straightforward solution for SQL Injection vulnerability since it is due to insecure

coding of web developers [4]. The defensive coding methods are use of parameterized queries or stored procedures which enforce developers to define the structure of SQL code first before including parameters to the query. Because parameters are bound to the defined SQL structure, no injection of additional SQL code is possible. Another method is data type validation which validates whether the input is string or numeric so that the input could be easily rejected if there is a mismatch in input type. This facilitates safe use of inputs in queries.

Whitelist filtering is another defensive coding technique against SQLI. Often blacklist filtering is used by developers to reject known bad special characters such as "'" and ";" from the parameters to avoid SQL Injection. However the safer approach is to accept only legitimate inputs. Here the developers could keep a list of legitimate data patterns and accept only the input data which match them. Although these methods can defeat SQL injection, their applications are manual, labour intensive, error-prone, and difficult to be rigorous and complete. To solve these problems, one approach is to use SQL DOM (Document Object Model) - a set of classes that provide automated data type validation and escaping. Both defensive coding and SQL DOM have no vulnerability locating & verification assistance and require high user involvement.

2) SQL Injection Vulnerability Detection: One method to detect vulnerabilities is code-based vulnerability testing.It uses static analysis to track user inputs to database access points and generate unit test cases for these points. Test cases contain SQL Injection attack patterns which check for SQLI vulnerabilities in the user inputs and then trace out and remove them. However, they do not explicitly find vulnerable program points and manual inspection is required to locate them. Another method is concrete attack generation which uses symbolic execution techniques for automatic generation of test inputs that exposes the SQLI vulnerabilities in the programs. It generates the test inputs by solving the constraints imposed on the inputs along the path to be exercised.

B. *XSS Defenses*

Preventing XSS requires keeping un-trusted data separate from active browser content. The preferred option is to properly escape all un-trusted data based on the HTML context (body, attribute, JavaScript, CSS (Cascaded Style Sheet), or URL) that the data will be placed into. Positive or white list input validation with appropriate decoding also help protect against XSS but not a complete defensive approach [7]. XSS defenses can be broadly classified into following types:-

1) Defensive Coding: Defensive coding validates and sanitizes the user inputs to eliminate the XSS vulnerabilities that may arise due to improper handling of inputs. There are four basic input sanitization options. Replacement and removal methods search for known bad characters (blacklist comparison). The replacement method replaces them with non-malicious characters, whereas the removal method simply removes them. Escaping methods search for characters that have special meanings for client-side interpreters and remove those meanings. Restriction techniques limit inputs to known good inputs (whitelist comparison). Checking blacklisted characters in the inputs is more scalable, but blacklist comparisons often fail as it is difficult to anticipate every attack signature variant. Whitelist comparisons are considered more secure, but they can result in the rejection of many unlisted valid inputs. Defensive coding practices have some limitations such as: labour-intensiveness, prone to human error, and difficult to enforce in deployed applications.

2) Vulnerability Detection: Static-analysis-based approaches avoid vulnerabilities in server-side scripts, but they tend to generate many false positives. Recent approaches combine static analysis with dynamic analysis techniques to improve accuracy. The static analysis identify tainted inputs accessed from external data sources, track the flow of tainted data, and check if any reached sinks such as SQL statements and HTML output statements. An open source vulnerability scanner performs alias analysis to improve accuracy. This is done because sometimes static analysis generates false positives, i.e., a reported vulnerable statement may be actually sanitized by escaping methods. But sometimes it will miss a real vulnerable statement. Static-analysis-based techniques quickly detect potential XSS vulnerabilities in source code and are relatively easy for security personnel to implement and adopt. However, they cannot check the correctness of input sanitization functions and assumes that unhandled or unknown functions return unsafe data.

V. IMPLEMENTATION DETAILS

Although the web application's development ha efficiently evolved since the first cases of SQLI and XSS attacks were reported, such attacks are still being exploited day after day. Since late 90's, attackers have continued to exploit the SQLI and XSS vulnerabilities across Internet web applications even though they were protected by traditional network security techniques, like firewall and cryptography based mechanisms. The use of specific secure development techniques can help to mitigate the problem. However they are not enough.

This situation shows the inadequacy of using basic security recommendation ass single measures to guarantee the security of web applications and leads to the necessity of additional security mechanisms to

cope with SQLI and XSS attacks when those basic security measures have been evaded. Here we present a scanner tool that checks the ASP.NET web application source code in offline for SQLI and XSS leaks in order to improve the overall security of the web application. This ensures the security of the ASP.NET web applications before making it public.

The scanner tool is implemented as a Windows .NET application in Csharp (C#). Here .NET is chosen as the implementation platform because a significant proportion of Internet users surf the web under MS (Microsoft) Windows. The scanner tool first authenticates the user of the tool and then allows them to check or test their ASP.NET web applications for vulnerability scanning. The scanning process is carried out by checking the ASP.NET files and code behind files such as .config file, .aspx file and .aspx.cs or .aspx.vb file extracted from the input directory path name and its subdirectories. The code behind files are checked according to the language in which the web application has been developed, in either Csharp (C#) or Visual Basic (VB). The two compiled languages C# and VB are adopted since they are the most common languages in use around the world with ASP.NET files in addition to aspx files.

The detection process for security vulnerabilities in ASP.NET websites/ applications is a complex process since in most cases there may not be any documentation to determine the purpose of the code and the code might be written by somebody else. Also the fact that ASP.NET which is a part of .NET framework, separate the HTML code from the programming code in two files, one for .aspx file and compiled language (Visual Basic VB, C sharp C#). Thus, three types of files are inspected during the scanning process such as .config, .aspx and C# or VB. After scanning each file it generates a report for each, which includes the information about the SQLI and XSS leaks in the file. The scanning report include the path name of the infected file from the input directory, it location and description of the leak. If the file include some leaks then it is rectified by removal or replacement method and also suggestions are provided to prevent attacks otherwise, it reports as the file is secure.

C. *Evaluation*

For evaluation purpose, a number of test scenarios have been executed. This includes: checking the debug, *validateRequest* and *customErrors* attributes in web.config file and aspx file; validating the textboxes in .aspx file with *RangeValidator* or *RegularExpressionValidator*; and avoiding the use of concatenation queries & encoding the HTML outputs and HTML outputs with input parameters in .aspx.cs or .aspx.vb file with AntiXSS methods in ASP.NET web applications [8][9]. Following are the test scenarios discussed to avoid SQLI and XSS leaks.

1) Set debug="false" in production environment: In production environment the *debug* attribute in web.config file must be set to false otherwise it will affect the application's performance by harming the website lifecycle, a lot of more files in temporary ASP.NET files folder will be downloaded so that it can slow down the user experience and any other third-party control or custom code that deploys client resources, also the pages does not timeout correctly. When <compilation debug="false"/> is set, the WebResource.axd handler will automatically set a long cache policy on resources retrieved via it – so that the resource is only downloaded once to the client and cached there forever. So much more memory is used within the application at runtime.

2) Enable the validateRequest attribute: The validateRequest property is set to true if the ASP.NET examines input from the browser for dangerous HTML markup in query strings, cookies or form fields; otherwise false. Request validation is performed by comparing all input data to a list of potentially dangerous values. If a match occurs, ASP.NET raises an HttpRequestValidationException and request is aborted.

3) Set the customErrors attribute to "On" or "RemoteOnly": When your application displays error messages, it should not give away information that a malicious user might find helpful in attacking our web application. For example, if your application unsuccessfully tries to log into a database, it should not display an error message that includes the user name it is using. There are some ways to control the error messages, such as:

- Configure the application not to show verbose error messages to remote users. So we set the customErrors attribute to "*RemoteOnly*". You can optionally redirect errors to an application page by including a *defaultRedirect* attribute that points to an application error page. This is done by the web application developer.
- Include error handling whenever practical and construct your own error messages by optionally including <error> elements that redirect specific errors to specific pages. This can be done by the web developer. For example, you can redirect standard 403 errors (No Access Allowed) to your own application.
- Create a global error handler at the page or application level that catches all unhandled exceptions and routes them to a generic error page. That way, even if you did not anticipate a problem, at least users will not see an exception page. In our system by default, we set the *customErrors* attribute to "*On*". According to developer's need he can set it to "*RemoteOnly*".

An example code for customErrors attribute is given:

```
<customErrors mode=”On” defaultRedirect=”GenericErrorPage.htm”>
<error statusCode=”403” redirect=”NoAccess.htm”>
<error statusCode=”404” redirect=”FileNotFound.htm”>
</customErrors>
```

4) Validate all textboxes in web page with Range Validator or Regular expression Validator: Use ASP.NET validator controls such as *RegularExpressionValidator* or *RangeValidator* to constrain the input supplied through server controls.

5) Avoid using concatenation select queries: The concatenation of user input to form SQL commands result in SQL Injection. So use Stored Procedures to create SQL queries. The scanner tool detects if there is a concatenated query and reports leak.

6) Encode the HTML outputs and HTML outputs with input parameters with AntiXSS methods: To prevent cross-site scripting [10] constrain the input supplied through client-side HTML input controls or input from other sources such as query strings or cookies by including the System.Text.RegularExpressions.Regex class in the server-side code to check for expected inputs using regular expressions [11][12], and the HTML outputs such as:

- Response.Write
- <%=
- <%#

and the input parameters from sources included in HTML outputs such as:

- Form fields as:
  - Request.Form[]
  - Request.Params[]
  - Request[]
- Query Strings as:
  - Request.QueryString[]
- Cookie collection as:
  - Request.Cookies[]

are encoded with user defined AntiXSS methods such as:

- AntiXSS.HtmlAttributeEncode()
- AntiXSS.HtmlEncode()
- AntiXSS.XmlAttributeEncode()
- AntiXSS.XmlEncode()
- AntiXSS.JavaScriptEncode()
- AntiXSS.VisualBasicScriptEncode()
- AntiXSS.UrlEncode()

Then we check whether the HTML outputs have been encoded with AntiXSS method, if not then reports leak and provide suggestion to encode with AntiXSS method.

VI. PERFORMANCE RESULTS

We perform the scanning process of the web application in offline i.e., before making it public, for vulnerabilities and generate the report to make the web application secure. For performing the scanning process we have created two ASP.NET web applications; one in Csharp and another one in Visual Basic. Then their files are checked for leaks to generate the report specifying the path name of the file from the input directory, location of the leak and description of the leak in the file, if there is leak otherwise it reports as file is secure. The results of scanning process are shown below.

D. Check the web.config file

If the config file contains any leak then the scanner tool generate a report about the leaks and these leaks are replaced or removed from the file. Otherwise scanner tool reports the file as secure.

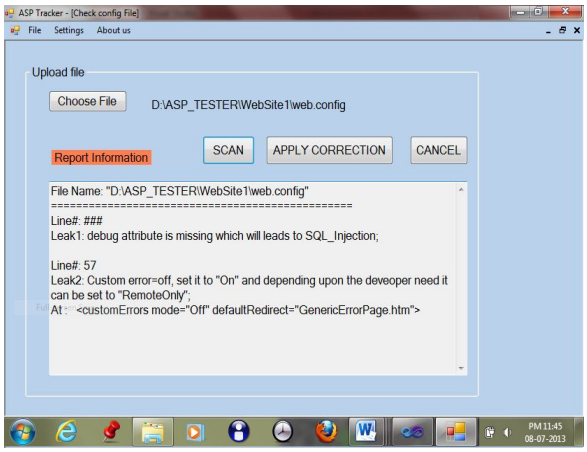

Fig. 1: Scanner tool report showing leaks in web.config file.

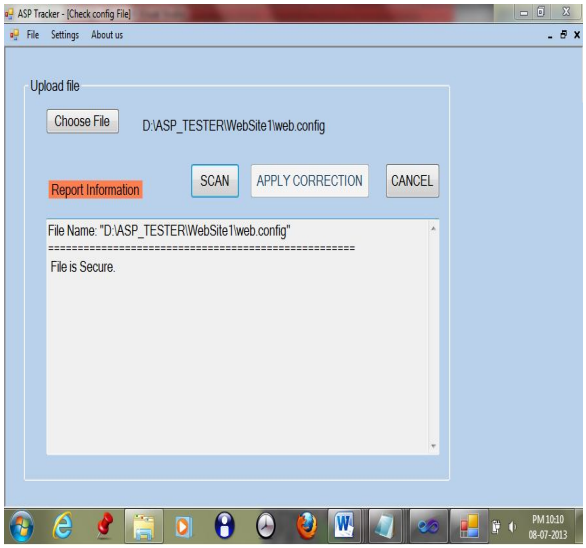

Fig. 2: Scanner tool report showing web.config file as secure.

E. Check the .aspx file

If the aspx file contains any leak then the scanner tool reports the leaks and leaks are replaced or suggestions are provided to avoid the leak. Otherwise reports as file is secure.

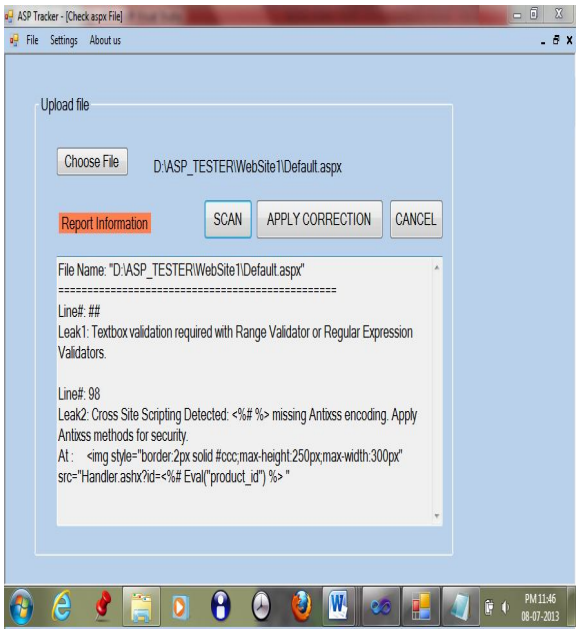

Fig. 3: Scanner tool report showing leaks in the .aspx file.

F. Check the .aspx.cs file

If the aspx.cs file contains any leak then the scanner tool generates a report about the leaks and suggestions are provided to remove the leaks in the file. Otherwise scanner tool reports the file as secure.

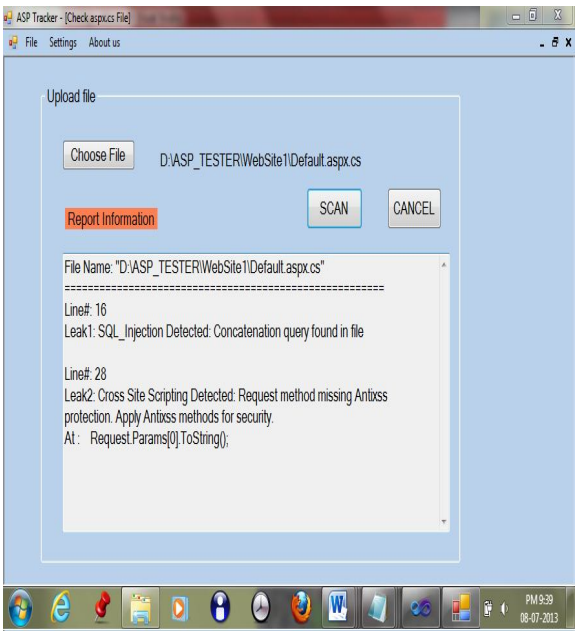

Fig. 4: Scanner tool report showing leaks in the .aspx.cs file

G. Check the .aspx.vb file

If the aspx.vb file contains any leak then the scanner tool generates a report about the leaks and suggestions are provided to avoid leaks from the file. Otherwise scanner tool reports the file as secure.

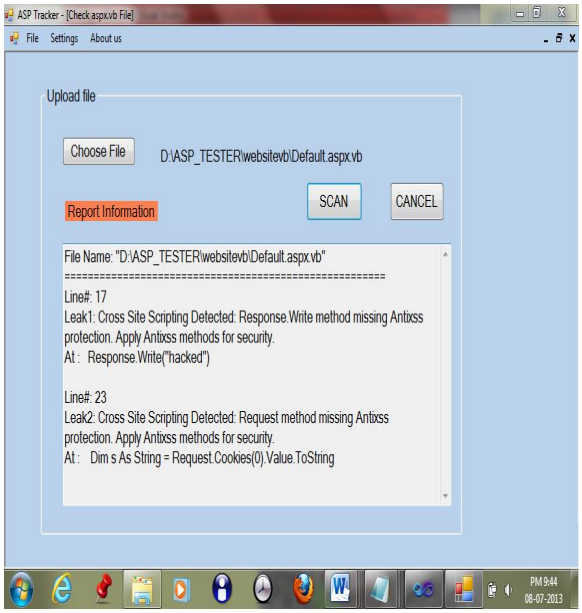

Fig. 5: Scanner tool report showing leaks in the .aspx.vb file.

CONCLUSION

In web applications in order to alleviate the vulnerabilities the web developer should be conscious about two things. One is the developer must be responsible to ensure security of the application from the beginning of coding and another one is developer must check their web applications for leaks before making them public. But implementing security is only part of the solution. Another important part is vigilance. Even if our application has many safeguards, we need to keep an eye on our web application to protect it from newly arrived security attacks. So monitor the web application's event logs and perform repeated attempts to log into your application. Continually keep the application server up to date with the latest security updates. Our developed tool considers the web application as three-tiered: Presentation, application and storage. Here web browser is the presentation, ASP.NET is the application and the database is the storage. The scanner tool study the web application source code for leaks and report about the discovered leaks if exist, otherwise report the code as secure.

REFERENCES

[1] Huyam AL-Amro and Eyas El-Qawasmeh, "Discovering Security Vulnerabilities And Leaks In ASP.NET Websites", pp. 329-333, December 2012.

[2] A.K. Baranwal, "Approaches to detect SQL Injection and XSS in web applications," EECE 571B, Term Survey Paper, Canada, April 2012.

[3] D.A. Kindy and A.K. Pathan, "A Survey on SQL Injection: Vlnerabilities, Attacks, And PreventionTechniques," Proc. Of ISCE2011, November 2011.

[4] Open Web Application Security Project, XSS (Cross-Site Scripting), Prevention Cheat Sheet, 2011; https://www.owasp.org/index.php/XSS (Cross-Site Scripting) Prevention Cheat Sheet.

[5] L.K. Shar and H.K. Tan, "Defeating SQL Injection," IEEE Computer publications, pp. 283-294., 2012, in press.

[6] M.Junjin. "A Approach for SQL Injection Vulnerability Detection" Proc. Of ITNG '09, pp. 27-29, April 2009.

[7] L.K. Shar, "and H.K. Tan, "Defending against Cross-Site Scripting Attacks," IEEE Computer Society Publications, pp. 55-62, March 2012.

[8] Security in ASP.NET Websites http://msdn.microsoft.com/en-us/library/91f66yxt%28v=vs.80%29.aspx

[9] https://www.owasp.org/index.php/Guide_to_SQL_Injection

[10] S. Fogie, J. Grossman, R. Hansen,A. Rager and P.D. Petkov, "XSS Attacks: Cross-Site Scripting Exploits and Defense," Syngress, 2007.

[11] Suman Saha, "Consideration Points: Detecting Cross-Site Scripting," IJCSIS 2009, Vol. 4, No. 1& 2, 2009.

★★★