Risk Mitigation for Cross Site Scripting Attacks Using Signature Based Model on the Server Side

Jayamsakthi Shanmugam, Dr.M.Ponnavaikko
[1]*Research Student, BITS, Pilani,* [2]*Director of Research and Virtual Education, SRM University, Chennai*
[1]*s_jayamsakthi@yahoo.com,* [2]*ponnav@srmuniv.ac.in*

Abstract

Researchers and industry experts state that the Cross-site Scripting (XSS) is the top most vulnerability in the web applications. Attacks on web applications are increasing with the implementation of newer technologies, new html tags and new JavaScript functions. This demands an efficient approach on the server side to protect the users of the application. The proposed Signature based misuse detection approach introduces a security layer on top of the web application, so that the existing web application remain unchanged whenever a new threat is introduced that demands new security mechanisms. The web pages that are newly introduced in the web application need not be changed to incorporate the security mechanisms as the solution is implemented on top of the web application. To test the effectiveness of this approach, the vulnerable web inputs listed in research sites, black-hat hacker sites and in the black hat hacker sites are considered. The proposed security system was run on JBoss server and tested on those vulnerable inputs collected from the above sites. There are around 100 variants of XSS attacks found during the testing. It has been found that the approach is very effective as it addresses the vulnerabilities at a granular level of tags and attributes, in addition to addressing the XSS vulnerabilities.

Keywords: Application-level web Security, cross-site scripting, security vulnerabilities.

1. Introduction.

Cross Site Scripting is one of the most common application level attacks that hackers use to sneak into web applications [1]. A typical scenario involves, a victim with an already established level of privilege with the target site and an attacker who initiates unauthorized actions using the victim's privilege. The web site is the target of attack and the user is both the victim and the unknown accomplice. However, the threat is not limited to the scenario quoted above.

While developing applications, the challenging tasks include authentication, identification and authorization. In spite of achieving maximum security regarding these tasks, a XSS attack can still be successful, because it allows a user to bypass traditional safeguards. One of the goals of the XSS attack is to steal the client cookies or any other sensitive information which can identify the client with the web site [2]. With the token of the legitimate user at hand, the attacker can act as the user in his/her interaction with the site – specifically impersonate the user. XSS occurs when a web application gathers malicious data from a user. The data is gathered in the form of a hyperlink which contains malicious content within it. The user gets affected when he clicks on the vulnerable link from another website, instant message, or simply just reading a web board or email message [3].

Users interact with the web sites by clicking on the links or filling and submitting the HTML forms in the browser. This results in a list of name/value pairs being sent to the server in the form of an http request. The request may contain other information such as a list of cookies, the referrer URL etc. XSS exploits the hyperlinks or client-side scripts (aka Scriptlets) such as JavaScript, VBScript, ActiveX, XHTML, Flash etc of the web based applications [4,5]. An XSS attacker typically uses a scriptlet mechanism to inject malicious code into a user session or its target web server to redirect the user with a malicious hyperlink or trigger a script that hijacks the user session to another web site. This XSS attack potentially leads to hijacking the user's account information, stealing cookie or session information, poisoning the user-specific content, defacing the web site, changing user privileges and so on [6,7].

The following input/output processing of web application provides the means for XSS attack:

0-7695-3039-7/07 $25.00 © 2007 IEEE
DOI 10.1109/IMSCCS.2007.82

- Injection points to the program: There are two ways by which the input is sent to the web server: GET and POST.

- All routines that returns data to the browser such as error messages, information to the users and warnings

Almost all of the web applications use cookies to associate a unique account with a specific user. In a web application logon scenario, two authentication tokens are exchanged. i.e a username and password is exchanged for a cookie [3]. Thereafter the values stored in a cookie are part of the authentication token. User's web session is vulnerable to hijacking if an attacker captures that user's cookies. So, if the attacker obtains the cookie by using the XSS technique, then he can load the cookie, point the browser to the appropriate web application site and access the victim's account without the correct combination of username and password [5]. The impact of cookie theft depends on the application. For instance the hacker can read a victim's email inbox, access bank records or buy items using cached retail credit card, before the legitimate user's session expires.

For better understanding of the vulnerabilities, few examples of XSS are listed below:

Assume a user is searching for the keyword "XML Tutorial" on a site called mydomain. The user's return URL would look like the following:

http://mydomain.com/index.asp?search=XML+Tutorial

The Attacker can craft another URL and make the user click on it through many media such as links in email, mouseover events on images etc.

The attacker's URL may look like,

http://mydomain.com/index.asp?search=</form><form action="hackerdomain.com/hack.asp">

This results in the execution of the script written in hack.asp that could log the user's cookie information.

Applications are constantly probed for vulnerability and when found to be vulnerable, are attacked with sustained belligerence. Recent researches show that the attacks on web applications are increased, since the attacks are launched on port 80 that remains open. SSL and firewalls are ineffectual against application level attacks as it cannot prevent the port 80 attacks. These attacks can bring down the web application server and can also provide access to the internal databases containing sensitive information like customer credit card numbers, account information and personal information. The literature survey carried out on the existing area of XSS attacks brings out the following facts:

- Cross-site scripting (XSS) variants dominated the top 10 vulnerabilities in commercial and open source Web applications, according to Cenzic Inc.'s *Application Security Trends Report* for the first quarter of 2007. In Cenzic's study, the company identified 1,561 unique vulnerabilities during the first quarter of 2007. File inclusion, SQL injection, XSS and directory traversal were the most prevalent, totaling 63%. The majority of vulnerabilities affected Web servers, Web applications and Web browsers [9].
- The standard on information security vulnerability that maintains the Common Vulnerabilities and Exposures (CVE) list, lists the top most vulnerability as XSS in web based applications. For 2006, 21.5 percent of the CVEs were found as XSS [8]. The data indicates hackers are exploiting XSS vulnerabilities in the web applications.
- 70% of attacks occur via the application layer, according to Stamford, Conn.-based research firm Gartner Inc.
- The Open Web Application Security Project (OWASP), dedicated to finding and fighting the causes of insecure software, it is stated that XSS attacks is the top most critical web application security flaw [9].
- Billy Hoffman, lead research engineer with Atlanta-based SPI Dynamics Inc. warned during his presentation at Black Hat conference USA 2006 that the XSS threats will only get worse and make life more difficult for IT security professionals [10,11].

Establishing a comprehensive security solution for XSS attacks becomes complicated due to the following reasons:

1.There are quite a few tags that are allowed in web applications for formatting the text. Hence, simple filtering mechanisms of the tags will not help in protecting those web applications from XSS attacks.

2. XSS vulnerabilities arise due to coding issues. The coding vulnerabilities vary from site to site and there is no single patch available to fix all the XSS vulnerabilities.

3. New evasive mechanisms are found by the hackers every day.

4. Web pages are not static. To increase the number of users, web application developers change the content of the application every day without concern for security mechanisms.

5. The entry points of the vulnerable XSS web applications can be found using automated tools inclusive of google [12].

Since tags are allowed to be entered in web pages for formatting the text displayed in web pages, hackers

find new ways to hack the web application using the features provided in the web application. As discussed in the following section the client side solution approach relies fully on the user configuration and when a new vulnerability is introduced, the new solution for the vulnerability installed at the central server cannot protect the client immediately till the automatic download takes place to have the security mechanism in client place. Because of this limitation this research focuses on server side solution. The currently available server side solutions also have certain limitations as discussed in the following section. Mainly all solutions provided by the earlier researchers do not address the XSS threats introduced because of the allowed tags in the web application. Literature survey carried out indicates all the approaches can lead to either too many false positives or false negatives. Our approach is to reduce the false positives and to protect the application from XSS threats.

2. Current Status.

Researchers in the past have proposed quite a few client side solutions, server side solutions, information flow based security mechanisms and scanners to address XSS vulnerabilities [13-29].

David Scott et al, suggested to define the security policies for input validation [13-14]. However, even though it provides immediate assurance of web application security, it requires the correct identification and validation policy for each individual entry point to a Web application. Bobbitt also observes that this is a difficult security task that requires careful configuration by "highly technical, experienced individuals" [15]. One another problem with this approach is on the response time from the server. If the number of hits increases, the dynamic generation of web pages will slow down the server performance.

The researches Engin Kirda et Al [16] and O.Ismail et al [17] provided a client side solution that fully relies on the user's configuration and number of researches have proven that client side solution is not reliable. If a new vulnerability is introduced, the new fix introduced at a central server to prevent the hacking cannot protect the user immediately as it needs an update on the client side system. Further according to Kruegel et al [18], it is not possible to maintain the misuse type IDS [19] (IDS are categorized basically into misuse and anomaly) due to large dynamic signature in an everyday attack scenario. CERT-Center of internet security expertise, a federally funded research and development center also states that none of the client side solutions prevent the vulnerabilities completely and it is up to the server to eliminate these issues [19].

Yao-Wen Huang et al [20] suggested a lattice based static analysis algorithm derived from type systems and type state. During the analysis, sections of code considered vulnerable are instrumented with runtime guards, thus securing web applications in the absence of user intervention. Though runtime protection is provided, it is tightly coupled with the web application. The main limitation is that it will take more processing time as the safeguards need to be revised and inserted in all pages.

The solution provided by Zhendong Su et al [21] provides a runtime checking for SQL command injection and claims that this approach will prevent XSS attacks. There are quite a few solutions proposed on the same lines of research [21-25]. Wes Masri and Andy Podgurski have stated [28] that information flow based work will increase the false positives and it is not an indicative strength if the information flow is high.

There are validation mechanisms [26] and scanners proposed to prevent XSS vulnerabilities [26-28]. Some software engineering approaches are also proposed such as WAVES [29] for security assessment. But none of the solutions are built for the latest developments and would fail if tags are permitted in the web applications.

The server side solutions proposed by earlier researches have the following limitations:

- When a new threat is introduced the new solution needs to be developed and incorporated in all the existing web pages.
- When a new web page is introduced, the security mechanisms need to be introduced at a web page level. This is a overhead for maintaining the web application.
- Each and every entry point in the web application should be known to the security administrator to implement the security mechanisms [30].

The proposed new signature based misuse detection solution overcomes the above difficulties with the following features.

- Configurable black listed tags, its attributes and object implementation procedure for misuse detection at the server side, and hence the existing web pages need not be modified for new threats.
- Whenever a new web page is introduced there is no need to modify the web page, since the security mechanism is separated from page level implementation and is placed at the top most layer of the web application.

- Security administrators need not know the entry points of individual web pages as there is a clear demarcation between the web application and security mechanisms implemented in this approach.

This research takes advantage of Signature based misuse detection on the server side to secure the web application and to block the known signatures to reduce the false positives. This approach is very effective in social networking sites, e-mail web application etc, where many web pages are dynamic, and aims at mass user base.

A negative security model blocks recognized attacks by relying on a database of expected attack signatures. This is also known as "blacklist " security model, which defines what is disallowed, while implicitly allowing everything else. The approach is as follows.

1. The policy is created with a set of known attack signatures.
2. There is no downstream page analysis to update the policy.
3. Recognized attacks are blocked, and unknown requests (good or bad) are assumed to be valid and passed to the server for processing.
4. All users share the same static policy.

Negative security model countermeasures identify bits of traffic known to be threatening. Anti-virus and intrusion detection/prevention systems are classic examples, both of which depend upon checking traffic flows against attack signatures. Negative security model monitors requests for anomalies, unusual behavior, and common web application attacks. It keeps anomaly scores for each request, IP addresses, application sessions, and user accounts. Requests with high anomaly scores are either logged or rejected altogether.

3. Proposed solution Procedure.

Solution Procedure and the model developed:

In the literature, a model that denies all transactions by default, but uses rules to allow only those transactions that are known to be safe is defined as a positive security model. But positive security model will increase the false positives. In negative security model all transactions are allowed by default. Our research aims to use the negative security model to reduce the false positives, and to protect the web application using configurable security mechanisms to increase the maintainability of the application. The proposed solution comprises of four components namely Blocker, parser, verifier and black listed tag cluster. This section describes the functionality of each component and the interactions between them.

3.1 Blocker:

When the HTTP request is received, the Blocker is called to initiate the actions. The first condition checked by the Blocker is the existence of special characters. This is because the script functions can only be executed when it is embedded using the tags and special characters. For example '<', '>', '%', '&', '\\', '&#' are few of the special characters used to embed JavaScript functions in the tags.

If special characters exist in the input, then the input is passed to the parser. Other wise the request is forwarded to the web application. Following two main methods are used in the Blocker class.

checkSpecialChars(str) - It checks whether there are any special characters exist in the input. processUserStatus() - This method receives the status from the parser, which in turn gets the status from verifier and redirects the user based on the status.

3.2 Parser

When the parser is called by the Blocker to process the input, parser breaks the input into multiple tokens, as tags and attributes and stores it as a element in a vector object. The input is then passed to the verifier component which is described below to assess the vulnerability. The following methods form the main part of parser class. setInput() - This method sets the input data.

isDataMalacious(vInput) –vInput is the vector object created by the parser component and it invokes verifier component to receive the processed status from the verifier class.

For instance if <img src=http://www.sample.com/image1.gif> is provided as an input then the vector element would contain the value as img, src=http://www.sample.com/image1.gif.

3.3 Verifier

Verifier checks the provided input for its vulnerability by executing the rules using the tag cluster defined below in section 3.4. If either the tag or the tag's attribute is in the black listed tag cluster, then it is concluded as tainted. The following two methods assess the vulnerability.

Verifier() - Constructor which sets the input data as vector. detectMalicious() - This method access the black listed cluster mentioned in table 1. This checks whether all the tags present in the input and its

respective attributes are in the black listed cluster are present in the XML mentioned below in table 1. It returns the Boolean value based on whether the assessed input is malicious or not.

3.4 Tag Cluster

Tag cluster is used by the verifier component described above. Cluster is a term defined by the authors in this context refers to the tags, attributes and its corresponding data type. With this, the clusters are categorized as follows:

Black listed cluster: The prohibited tags and the prohibited attributes of those tags are categorized as black listed cluster that are permitted in the web application. The tags that make the application vulnerable for XSS attacks are categorized in this cluster. These are used to formulate the problem of negative security model. The following is an example of black listed tag:

<Script>alert('XSS')</ Script >

This approach compares the provided input with the black listed cluster. The following defined rule is used to identify the vulnerability by the verifier component.

Rules for vulnerability identification:

The following definitions are made to define the tags with respect to the group of tag clusters described in section 3.4. Further the definitions are used to form the rules to identify the vulnerability.

Let I= $\{I_1, I_2, I_3 \ldots I_n\}$ be a finite set of tags in the input.

Let B = $\{B_1, B_2, B_3 \ldots B_n\}$ be the finite set of black listed tags.

$\{MS_1, MS_2, MS_3 \ldots MS_n\}$ be the corresponding set of security classes for the tag Bi to identify the attribute or the value of the tag content to determine whether the input provided is malicious. Few tags that are included in this cluster need to be checked for vulnerability in the value of attributes. For instance in the below stated example, IMG is the tag and SRC is its attribute. The value of the attribute is javascript:alert('XSS').

<IMG SRC="javascript:alert('XSS');">

It is clear from the above example that IMG SRC attribute is not pointing to an image, but a JavaScript function. Hence, the SRC attribute should be checked for the value it contains to identify the vulnerability.

In the above stated example under black listed cluster, a class is associated with the tag IMG to check the content of the source attribute. If it is not the type of the image like .jpg or .gif or .bmp etc, then the input is identified as tainted. In problem formation, the authors use the following rules to conclude whether the input is tainted or not.

Rules to conclude an input as untainted input is defined as follows:

Ii is untainted, only if it is not a subset of $\{B_1, B_2, B_3 \ldots B_n\}$ *where* I_i *is the tag in the input and if security classes identify the attribute's value as untainted.*

Rules to conclude an input as tainted input is defined as follows:

If Ii is a subset of Bi then it is concluded as tainted.

If Ii is a part of black listed tags and if security classes identify the attribute's value as malicious, then the input is concluded as tainted.

Once the process execution is complete by the verifier, the status is returned to the parser class. It can either be 'Yes' or 'No' depending up on the vulnerability detected in the input. Parser class passes the status to Blocker class. Based on the status, Blocker either redirects the request to error page or to the web page.

The solution procedure is explained in Figure 1.

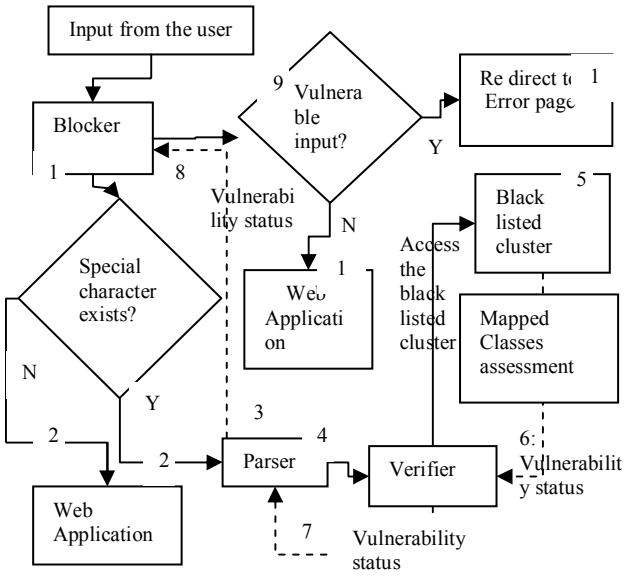

Figure 1: Flow of input through the components

Figure 1 describes the flow of the system. The execution sequence is numbered in the above diagram for better understanding of the process. Blocker checks for the special character existence in the input and if it exists then it forwards the request to the parser. The parser splits input to tokens and sends it to the verifier. The verifier accesses the black listed cluster and checks for its vulnerability. If there is no vulnerability detected then the verifier returns the status to parser. The parser then returns the status to Blocker. Based on the status returned, Blocker either redirects the request

to the error page or forwards the request to the web application as depicted in Figure 1.

4. Implementation.

4.1 Technical details of implementation

The proposed solution is implemented in JSP/Servlets using JBoss server. The following entries are made in struts framework's web.xml file to redirect the HTTP requests to the class Blocker. Blocker is the class where the special character analysis of the input is implemented. The configuration is as follows:

```
<filter>
   <filter-name>struts-Blocker</filter-name>
<filter-class>org.apache.struts2.dispatcher.Blocker
</filter-class>
</filter>
<filter-mapping>
   <filter-name>struts- Blocker </filter-name>
   <url-pattern>*.do</url-pattern>
</filter-mapping>
```

Around 1500 lines of code has been developed and also 108 unique XSS test cases are created to test this approach. This approach is also tested with about 2200 vulnerable input data collected from various research sites and in the black hat hackers' site where the proof of code is provided for XSS vulnerability. The list of vulnerable web pages and the test cases are available for researchers and they can contact the authors through email to get the list. Out of 2200 XSS vulnerable web pages found, around 160 web sites are SSL protected banking applications.

The following is the snippet of code used in Blocker to diagnose the input for special characters:

```
public static final String REGEX = "(<[a-zA-Z][^<>]*>)|( <>]*>)";
private static final Pattern HTML_PATTERN = Pattern.compile(REGEX);
```

As could be seen in the above snippet, regular expression is used for diagnosis of special characters and if special characters are found, it is passed on to the parser. For implementation purposes, the StringTokenizer class in Java is used in the parser class, which is described in section 3.2. Parser class calls the verifier class in a loop as there could be other nested tags within the input. Verifier class uses the following respective structure of XML described in table 1:

Table 1: Sample Structure of the Tag Clusters:

Black listed cluster XML Structure
<BlackList> <TagCluster>

```
  <Tag>
    <TagName>someTag</TagName>
    <attributeName>attributeName</attributeName>
    <attributeName>attributeName </attributeName>
     <ClassName>someClassName</ClassName>
  </Tag>
  <Tag>
    <TagName>someTag</TagName>
    <attributeName> attributeName</attributeName>
     <ClassName>someClassName</ClassName>
  </Tag>
</TagCluster>
</BlackList>
```

5. Evaluation of the approach.

We have adopted two approaches to test our solution. First, the solution is applied on a banking web application and tested for its performance with and without the Signature based misuse detection procedure.

The proposed solution has been tested with 6000 malicious inputs and 5000 non-vulnerable input with black listed tags The average time has been taken for 10 cycles of execution of each approach and the results are presented in table 3. The average time is taken because there are minor variations found in the time of completion of each run as the execution depends on the operating system, and the other processes that run in the machine during the process of testing.

The performance has been observed by logging the time of the verifier process before it initiates the vulnerability assessment and after the status is received from the parser. The approach is tested in a Pentium 4, 256 MB RAM and 1.69GHz.machine.

Though the vulnerable input collected is around 2200, the authors increased the data by deriving the combinations of vulnerability for the remaining 4000 vulnerable input to test the performance speed of the proposed approach. The approach is also tested by a random generator program that picks the vulnerable and non vulnerable inputs from a file of about 5000 inputs for an average of 10 runs and the results are presented below.

Table 3: Before and after the security mechanisms are applied.

	Vulnerable input processing time in milli seconds to process 6000 vulnerable inputs	Non vulnerable input processing time in milli seconds to process 5000 inputs	Random generator program test for 5000 inputs, represented in milli seconds with a mixture of vulnerable

			and non vulnerable inputs.
Security Mechanisms applied	2100	549	850
No Security Mechanisms applied	2000	500	836

It has been observed that there is an increase in the processing time to process a single vulnerable input request from 0.33 to 0.40 milliseconds after the implementation of the security mechanisms, which is 0.075 milliseconds increase per request, which is a very minor increase in the processing time. To process a non-vulnerable input, on an average the proposed system takes .014 milliseconds higher than the system with out the security mechanisms implemented. The authors perceive that the performance could be improved by stopping the verifier process once the vulnerability is detected. Also, the authors are working towards reducing the processing time by using other parsers which could maximize the process utilization.

During the processing of testing it has been observed that more than 100 variants of XSS attacks exist and the approach is tested with the data collected from various research sites, black hat and black hat sites. The following are few of the test conditions tested in the input fields of the web page:

Table 4: Test Results excerpts:

Sr. number	Test Condition	Test Result
1	<SCRIPT/SRC="http://Url/xss.js"></SCRIPT>	Test condition Passed
2	<IMG SRC=javascript:alert('XSS')>	False negative, as the input is completely encoded.

Though the approach is fully functional, all types of encoded attacks are not addressed in this approach, and this leads to few false negatives and false positives. For instance the 2nd test case mentioned in table 4, lead to false negative, since the approach addresses basic encoding attacks. As of now, if the input is encoded it is rejected to avoid the threats to the system. The authors are working to provide an efficient solution to address encoding attacks also. As the solution relies on the black listed cluster, it requires careful configuration, else this approach could lead to more false positives.

6. Conclusion.

The web applications are facing severe threats due to the loopholes/vulnerabilities present in new technologies like Ajax. Available methods do not provide adequate solution for protecting the web applications. The proposed server side solution approach meets the need to protect the web applications with the perspective to improve the response time while addressing the XSS attacks. The results are highly encouraging and the proposed solution approach is found to be very effective for securing the web pages from XSS attacks. The Signature based misuse detection approach for prevention of XSS threats has the following advantages:

1. This approach allows tags to be entered in the web application and at the same time provide security for the web application.
2. The research work uses the positive security model to reduce the processing time. In the negative security model, the processing time of the server increases for every new threat introduced, since the input should be matched with the larger number of signatures as the XSS attack surface is very high. In the authors approach, the attack surface is minimized using the positive security model.
3. The solution provided is highly configurable unlike other solutions provided. Black listed cluster is configurable which is described in section 3.4.
4. The solution is modularized, so there is a clear demarcation of functionality performed by each module and hence functions can be added with least effort. This makes the security application maintainable.
5. The solution need not be included in each and every page like earlier works. The solution stays on top of the web application and does not require changes in the web application.
6. This approach is not prone to zero-day attacks since the approach checks only for the known or goodness of the input. Even if a new threat is introduced this approach would reject the input as the signature would not be known in black listed cluster.
7. Addresses basic encoded attacks.

This approach needs an updation in the black listed cluster XML data, when a new tag needs to be permitted. As of now the Signature based misuse detection on the server side approach does not address all the encoding patterns, and the authors are working towards it to include the solution in the next release. The authors are working in this direction for further improvement of the proposed solution.

8. References.

[1] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, P. Tramontana, "Identifying Cross Site Scripting Vulnerabilities in Web Applications," *Sixth IEEE International Workshop on Web Site Evolution(WSE'04)*, pp. 71-80, , 2004.

[2] M. M. Burnett and J. C. Foster, "Hacking the Code: ASP.NET Web Application Security," Chapter 5 - Filtering User Input, Syngress Publishing © 2004

[3] T. Gallagher, B. Jeffries, and L. Landauer, "Hunting Security Bugs," Chapter 10 - HTML Scripting Attacks, Microsoft Press © 2006.

[4] D. Scott, R. Sharp "Abstracting Application-Level Web Security." In: *Proc. 11th Int'l Conf. World Wide Web (WWW2002)*, Honolulu, Hawaii, May 17-22, pp. 396-407, 2002.

[5] Scott, D., Sharp, R. "Developing Secure Web Applications." *IEEE Internet Computing*, 6(6), pp. 38-45, Nov 2002.

[6] Joel Scambray, Mike Shema, and Caleb Sima, "Hacking Exposed: Web Applications," 2nd Edition, pp. 215- 221, McGraw-Hill Companies, Jun 19 2002

[7] CBS news, "Cyper Criminals Target Web Services, Yahoo, Google, PayPal Seek To Close Security Holes, SAN FRANCISCO, June 23, 2006

[8] Common Vulnerabilities and Exposures, "The standard for information security vulnerability names," http://cve.mitre.org/, last accessed May 24, 2007.

[9] Colleen Frye, "XSS the top vulnerability in most Web applications in Q1," http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1256570,00.html?track=NL-498&ad=590666&asrc=EM_NLN_1501330&uid=5685607

[10] Bill Brenner, Senior News Writer, "Ajax threats worry researchers,"http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci1207759,00.html, 04 Aug 2006, last accessed May 24, 2007.

[11] Vulnerable sites information posted by hackers, http://sla.ckers.org/forum/read.php?3,44,632, last accessed May 24, 2007

[12] Jaikumar Vijayan, "'Less than zero-day' threats too often overlooked, analysts warn Companies tend to focus only on patching known flaws, ignoring other threats,"Computerworld, October 26, 2006

[13] Scott, D., Sharp, R. "Abstracting Application-Level Web Security." In: Proc. 11th Int'l Conf. World Wide Web (WWW2002), pages 396-407, Honolulu, Hawaii, May 17-22, 2002.

[14] Scott, D., Sharp, R. "Developing Secure Web Applications." IEEEInternet Computing, 6(6), pp. 38-45, Nov 2002.

[15] Bobbitt, M. "Bulletproof Web Security." Network Security Magazine, TechTarget Storage Media, May 2002., http://infosecuritymag.techtarget.com/2002/may/bulletproof.shtml, last accessed May 24, 2007.

[16] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks" In The 21st ACM Symposium on Applied Computing (SAC 2006), pp. 330 - 337 , April 23-27, 2006.

[17] O.Ismaill,M.E.Youki,K.adobayashi, S.Yamaguch, " A proposal and Implementation of Automatic Detection/Collection system for Cross-Site Scripting Vulnerability" Proceeding of the 18th Internation conference on Advanced Information Networking and Application (AINA'04).

[18] Christopher Krugel, G.Vigna, William Robertson, "A multi-model approach to the detection of web based attacks," Computer Networks 48 (2005) pp.717-738 – ELSEVIER, 2005.

[19] CERT® Advisory CA-2200-02, "Malicious HTML Tags Embedded in Client Web Requests," http://www.cert.org/advisories/CA-2200-02.html, last accessed May 24, 2007.

[20] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, Sy-Yen Kuo, " Securing web application code by static analysis and runtime protection," . In Proceedings of International WWW Conference, New York, USA, 2004, pp.40 – 52, May 17–22, 2004.

[21] Zhendong Su, Gary Wassermann, "The essence of command injection attacks in web applications," Annual Symposium on Principles of Programming Languages, Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 372 - 382, Jan 11-13, 2006.

[22] Wes Masri American University of Beirut, Beirut, Lebanon , Andy Podgurski Case Western Reserve University, Cleveland, OH , "Using dynamic information flow analysis to detect attacks against applications," ACM SIGSOFT Software Engineering Notes Volume 30 , Issue 4 July 2005.

[23] N.Jovanovic, C. Kruegel and E.Kirda, "Pixy: A Static Analysis tool for Detecting web application vulnerabilities," Proceedings of the 2006 IEEE Symposium on Security and Privacy(S&P'06).

[24] Wes Masri, Andy Podgurski, David Leon, "Detecting and Debugging Insecure Information Flows," issre, pp. 198-209, 15th International Symposium on Software Reliability Engineering (ISSRE'04), 2004.

[25] Jin-Cherng Lin, Jan-Min Chen, "An Automatic Revised Tool for Anti-Malicious Injection," cit, p. 164, Sixth IEEE International Conference on Computer and Information Technology (CIT'06), 2006.

[26] "The JWIG Project," http://www.brics.dk/JWIG/.

[27] Wes Masri, Andy Podgurski, "An Empirical Study of the Strength of Information Flows in Programs," ACM 1-59593-085-X/06/0005, WODA'06, May 23, 2006, Shanghai, China.

[28] Yao-Wen Huang, Chung-Hung Tsai, D. T. Lee, Sy-Yen Kuo, "Non-Detrimental Web Application, Security Scanning," issre, pp. 219-230, 15th International Symposium on Software Reliability Engineering (ISSRE'04), 2004.

[29] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, Chung-Hung Tsai, "Web application security assessment by fault injection and behavior monitoring," International World Wide Web Conference, Proceedings of the 12th international conference on World Wide Web, SESSION: Data integrity table of contents, pp. 148 - 159 Year of Publication: 2003.

[30] Yao-Wen Huang, Chung-Hung Tsai, D. T. Lee, Sy-Yen Kuo, "Non-Detrimental Web Application, Security Scanning," issre, pp. 219-230, 15th International Symposium on Software Reliability Engineering (ISSRE'04), 2004.