A Server Side Solution for Protection of Web Applications from Cross-Site Scripting Attacks

A. Duraisamy, M.Sathiyamoorthy, S.Chandrasekar

Abstract — Cross-Site scripting attacks occur when accessing information in intermediate trusted sites. Cross-Site Scripting (XSS) is one of the major problems of any Web application. Web browsers are used in the execution of commands in web pages to enable dynamic Web pages attackers to make use of this feature and to enforce the execution of malicious code in a user's Web browser. This paper describes the possibilities to filter JavaScript in Web applications in server side protection. Server side solution effectively protects against information leakage from the user's environment. Cross-Site scripting attacks are easy to execute, but difficult to detect and prevent. The flexibility of HTML encoding techniques, offers the attacker many possibilities for circumventing server-side input filters that should prevent malicious scripts from being injected into trusted sites. Cross site scripting (XSS) attacks are currently the most exploited security problems in modern web applications. These attacks make use of vulnerabilities in the code of web-applications, resulting in serious consequences, such as theft of cookies, passwords and other personal credentials. It is caused by scripts, which do not sanitize user input.

Keywords-Web Application; Cross Site Scripting; Server Side Solution; Detection of XSS Attacks, XSS Filter, HTML Input Filter

I. INTRODUCTION

Cross Site Scripting (XSS), is the most widespread and harmful web application security issue. It was first noticed, when CERT (Computer Emergency Response Team) published an advisory on newly identified security vulnerability affecting all web applications. This flaw occur whenever a web application takes data that originated from a user and sends it to a web browser without first validating or encoding that content.

XSS is used to allow attackers to execute script in the victim's browser, which can hijack user sessions, deface web sites, insert hostile content, and conduct phishing attacks. Any scripting language supported by the victim's browser can also be a potential target for this attack. Web based applications are accessed using Web based communication protocols and use Web browsers as graphical user interface. Many number of Web applications make use of either basic HTTP or higher level protocols based on HTTP such as SOAP.

Manuscript received on March, 2013.

A.Duraisamy, Department of Information Technology, University College of Engineering, Tindivanam (T.N), India.

M.Sathiyamoorthy, Department of Information Technology, University College of Engineering, Tindivanam (T.N), India.

S.Chandrasekar, Department of Information Technology, University College of Engineering, Tindivanam (T.N), India.

The Web browser is used as graphical user interface (GUI); these applications must provide HTML data for the browsers to be displayed to the users. In earlier only static HTML Web pages has been used, but quickly applications have been developed that generated the HTML code dynamically.

To provide more flexibility in the HTML display and to reduce round-trip delays, browsers offered the possibility to insert program code into the HTML document that is read and executed on the fly by an interpreter integrated into the browser. Java Script code may not be mixed up with Java Server Pages (JSP); JSP code is executed at the server side and not at the client browser. The Java Applets is a different client side technology that allows the download and execution of Java applications to and at the client machine. The java Applets normally does not directly manipulate the browser or HTML document.

Cross-site scripting (XSS) [15] continuously leads the most wide-spread Web application vulnerabilities lists. Estimates in [21] suggest that 87 percent of all current Web sites are vulnerable to XSS. Even though, not even Google search is spared from such attacks. With XSS vulnerability in Google's online spreadsheet application [5] it was possible to steal a user's cookie (which was valid for all of google.com's subdomains, e.g., mail.google.com, code.google.com, spreadsheets.google.com). Frequently online banking applications, which make a very attractive target for XSS in order to set up phishing sites, are vulnerable to XSS, as has been demonstrated by Phishmarkt [1, 2]. Technically, XSS attacks leverage insufficient input/output validation in the attacked Web application to inject JavaScript code, which is then executed on the victim's machine within the exploited Web site's context, thus bypassing the same origin policy. The attacker can craft the injected script such, that it discloses the victim's confidential information, e.g., a session ID. Then, by hijacking the session, the victim can be impersonated. Also, XSS enables the construction of very powerful phishing pages, since the page content is actually delivered by the correct, trusted site. The HTML document is scanned by the browser for the presence of JavaScript code. The code will be read and executed by the browser without displaying to the user. JavaScript code is written using several HTML markups as follows: JavaScript code is enclosed by start and end script tags.

```
<script> alert('XSS')</script>
```

JavaScript code is indicated by a protocol specifier as follows:

```
javascript:alert('XSS');
```

JavaScript code allows functions that are not executed directly and can be called later on. These function definitions occur anywhere in JavaScript section.

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

The code parts are placed in separate files to provide code modularization.It can be loaded using optional src parameter as shown in following example.

<script src="Js-lib.Js"></script>

Uniform resource locator (URL) is used to load the script code by the web browser. Statements within function definitions are executed only when the function is called, while the normal statements are executed directly. Every HTML tag supports special parameters that allow JavaScript functions to be called automatically if specific events occur such as initial load of the document, as shown in example:

<body onload="Body_Execute()"> … </body>

The above function Body_Execute() is called in JavaScript environments when the page containing the tag is loaded. Some other examples of event related parameters are:

- *onClick* for the HTML tags button, checkbox, radio, reset, submit
- *onChange* for the HTML tags select, text, text area, text field
- *onSubmit* for the HTML tag Submit button, reset button within a form environment
- *onMouseOver* for any HTML tag and script tag.

Cross Site Scripting

The possible ways to manipulate HTML documents displayed by the browser with JavaScript or to influence the operation of the browser itself are dangerous features if misused. Some time, unfortunately JavaScript code provides full access to HTML documents using the document object model (DOM). A script code can modify at least the document it is residing in arbitrarily: it is also possible to completely delete the document and create a totally different document. Any attacker's point of view two things are of special interest: cookies associated to a document and access credentials, JavaScript also provide access possibilities to this information. A document can be accessed using the function call *document1.cookie1* and application level access credentials are often acquired using form based login. The credentials data are input into input fields residing in a form environment, since the form is part of the document a script can access all information in all fields or can simply modify the target URL of the form, and then the credentials are sent to the new target, which is under the control of the attacker.

These above few example shows, that JavaScript's native function provides all possibilities for attackers, if malicious script code can be inserted into a HTML document. To detect and prevent that script code contained in a document loaded from some Web site accesses documents loaded from some other Web site, browsers do not allow access between documents loaded from different sites (i.e. cross-site access). Generally there are two types of Cross-Site Scripting attacks are available:

- Stored or Persistent Cross Site Scripting attacks
- Reflected or non persistent Cross Site Scripting attacks

1. ***Stored XSS Attacks***

Persistent or Stored Cross-Site Scripting flaws are those where some data sent to the server is stored to be used in the creation of pages that will be served to other users later. This type of Cross-Site Scripting flaws can affect any user to our website, if our site is subject to Persistent Cross-Site Scripting vulnerability. One of the familiar examples of persistent or stored vulnerability is content management software such as forums and bulletin boards where users are allowed to use raw HTML and XHTML to format their posts. Preventing reflected flaws, the key to securing our web site against stored flaws is ensuring that all submitted data is translated to display entities before display so that it will not be interpreted by the browser as code.

An unprotected site providing a forum where users must identify with user name and password is the ideal environment for stored Cross-Site Scripting attacks. In case of forums where cookies are used to provide successful authentication, then the attack is performed as follows:

```
<SCRIPT>document.location('http://evil.org/steal.cgi?c
=+escape(document.cookie);')</SCRIPT>
```

2. Reflected XSS Attacks

It is the most familiar type of Cross-Site Scripting exploit. It targets vulnerabilities that occur in some websites which deals with dynamic result 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, and when those results are sent the code is not encoded using HTML special character encoding, thus being interpreted by the browser rather than being displayed as inert visible text. The attack can be done by using a link using a malformed URL, such that a variable passed in a URL to be displayed on the page contains malicious code. Another Uniform Resource Locator (URL) used by the server-side code to produce links on the page, can also become a vulnerability employed in a reflected Cross-Site Scripting flaws. If the username parameter of the login page is vulnerable, then the attacker can setup an attack URL with a script that rewrites the action target which presets the username in the form as shown below:

```
<SCRIPT>document.forms.action='http://evil1.org/steal.cgi
?c=+escape (document.cookie) ;'< /SCRIPT>
```

This attack mainly involves a link which contains malicious code in the variable passed in URL. Vulnerabilities may be the URL used by the server-side code to produce links on the page, or may be even a user's name to be included in the text page so that the user can be greeted by name, can become a vulnerability employed in a reflected cross-site scripting exploit.

Impact of XSS-Attack

- Access to authentication credentials for Web application
- Cookies, Username and Password
  - XSS is not a harmless flaw
- Normal users
  - Access to personal data (Credit card, Bank Account)
  - Misuse account (order expensive goods)
- Denial-of-Service
  - Crash Users `Browser, Pop-Up-Flooding, Redirection Access to Users` machine
  - Use ActiveX objects to control machine
  - Upload local data to

attacker's machine

- Spoil public image of company
    - Load main frame content from other locations
    - Redirect to dialler download

Reflected or non persistent XSS detection by request/response pattern matching

This detection mechanism for reflected XSS attacks is based on the observation that reflected XSS implies a direct relationship between the input data (e.g., HTML parameters) and the injected script. The injected script is fully contained both in the HTTP request and the HTTP response. Reflected XSS attacks should be detectable by simply matching incoming data and outgoing JavaScript using an appropriate similarity metric using pattern matching techniques. It is to emphasize that we match the incoming data only against script code found in HTML with predefined white list parameter. Non-script HTML content is ignored, the sake of readability we will uses the term parameters as a generalized term for all user-provided data in the sequel.

We can define the problem definition to be solved as follows:

Problem Definition

Given a set of parameters P = {p1, p2… pm} and a set of scripts S = {s1, s2… sn} find all matches between P and S in which pi was used to define parts of sj.

In this paper, we present a novel approach to protect users against XSS attacks that offers the same level of protection as previous work, but without the necessity for client-side modifications. To avoid the disadvantage of involving the end-user, we position a Web browser on a reverse proxy and XSS filter before the server. Our idea is based upon the fact that a Web browser on the client's machine is the ultimate receiver of JavaScript code, and a straightedge for script interpretation capabilities. By utilizing a Web browser, we are able to distinguish between benign and injected Java Script code. First, we encode all benign JavaScript calls to syntactically invalid identifiers. Second, we load each requested page in the Web browser attached to the reverse proxy and XSS filter, and watch out for scripts trying to execute. Clearly, all remaining scripts have not been encoded before hand, and not expected, benign scripts, but injected, malicious ones. Third, after verifying that there is indeed no (malicious) script in the page, we decode all previously generated script IDs to restore the original code, and deliver the page to the client.

This paper is summarized as following contributions:

1. We introduce XSS filter, a solution for mitigating XSS attacks, by utilizing a Web browser in order to detect malicious JavaScript content.
2. We introduce HTML Input filter for analyses incoming HTTP request and outgoing HTTP response without any harmful java script.
3. In contrast to previously proposed solutions, our server side solution does not require client-side modifications. Thus, each Web site can be protected from XSS flaws transparently for its visitors.
4. We describe our implementation of "A Server Side Solution for Protection of Web applications from Cross site scripting attacks", and demonstrate its efficiency in successfully detecting and preventing authentic attacks on two popular Web application's XSS vulnerabilities.

II. RELATED WORK

A. Server Side Solution:

The cross site scripting vulnerabilities in Web applications, a number of testing tools has been proposed earlier. There are two types of testing tool such as, Black-box [4] Web application testing tools as well as white-box [6] vulnerability scanners have been suggested in previous research, and are successfully used in real time practice. This kind of tools can generally help in identifying cross site scripting vulnerabilities, it is likely that some remain undetected, which clearly recommends additional safeguards for web application. Such kind of tool also, the owner of a Web site running a third party Web application to fix the identified bugs, requires the commitment of the developers of the Web application, which often have other priorities that seem more economically rewarding. In [1], an application-level firewall is suggested, which is located on a security gateway between server and client, and which applies all security relevant checks and transformations. By separating the security relevant part of the code from the rest of the application, as well as providing a specialized Security Policy Description Language to design it, the system helps Web developers to apply measures against XSS in a less error prone fashion. Comparably to this work, we also use a reverse Web proxy to implement XSS mitigation strategies. However, while the security gateway operates on the incoming requests, our reverse proxy inspects the server's replies.

This is preferable because it protects visitors of the page even if an attacker found a way to inject his malicious content in spite of the security gateway's checks. Additionally, by using an actual Web browser in order to identify scripts instead of a complex policy that targets various kinds of sanitization , our approach asks less from Web masters who wish to deploy it, and leaves less room for mistakes. Scott and Sharp [1] describe a web proxy that is located between the users and the web application, and that makes sure that a web application adheres to pre written security policies. The main categories of such policy based approaches are that the creation and management of security policies is a tedious and error-prone task. Similar to [1], there exists a commercial product called AppShield, which is a web application firewall proxy that apparently does not need security policies. Furthermore,[1] reports that AppShield is a plug and play application that can only do simple checks and thus, can only provide limited protection because of the lack of any security policies.

B. Client-side solution:

Complementary to mitigating XSS on the server-side, there are several client-side solutions. In [5], a strictly client-side mechanism for detecting malicious Java Scripts is proposed. The system consists of a browser-embedded script auditing component, and IDS that processes the audit logs and compares them to signatures of known malicious behavior or attacks.

With this system, it is possible to detect various kinds of malicious scripts, not only XSS attacks. However, for each type of attack a signature must be crafted, meaning that the system is defeated by original attacks not anticipated by the signature authors. Client side cross site scripting protection (Noxes Tool) [3] is a client-side Web-proxy that relays all Web traffic and serves as an application-level firewall. The approach works without attack-specific signatures. However, as opposed to SWAP [2], Noxes requires user-specific configuration (firewall rules), as well as user interaction when a suspicious event occurs.

The main difference of our approach with respect to existing solutions [2] is that it is a Server-side solution. The solutions presented server-side that aim to protect specific web applications. Huang [7] describe the use of a number of software-testing techniques and suggest mechanisms for applying these techniques to web applications. The main aim is to cover and fix web vulnerabilities such as XSS. The researches Engin Kirda et al [8] and O.Ismail et al [9] 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.

Another client-side approach is presented in [14], which aims to identify information leakage using tainting of input data in the browser. All client-side solutions share one *drawback:* The necessity to install updates or additional components on each user's workstation. While this might be a realistic precondition for skilled, security-aware computer users, it is perceived as an obstacle or is not even considered by the vast majority of users. Thus, the level of protection such a system can offer is severely limited in practice. Pixy [11] performs tainted data flow analysis using flow-sensitive, inter procedural, context-sensitive data flow analysis and checks if user input is used at a target statement without any input validation. Web Static Approximation [20] uses a static string analysis technique to approximate possible string output for variables in a web application and checks if the approximated string output is disjoint with unsafe strings defined in a specification file. If the approximate string output is disjoint with the unsafe strings, Web Static Approximation reports that the application is not vulnerable.

C. Hybrid mitigation approaches:

Some solutions apply hybrid approaches, which also involve the Web browser. The server annotates the delivered content and provides information on the legitimacy or level of privileges of scripts. The Web browser is then responsible for checking and enforcing these annotations. BEEP (Browser-Enforced Embedded Policies) [13] proposes to use a modified browser that hooks all script execution attempts, and checks them against a policy, which must be provided by the server. Two kinds of policies are suggested. First, using a white list of the hashes of all allowed scripts, which the browser can check against. Second, labeling those nodes in the HTML source, which are supposed to contain user-provided content, so the browser can determine whether a script's position in the DOM tree is within user-provided content. The modified browser verifies each script with respect to the policy and prohibits scripts from execution that do not comply. Wes Masri and Andy Podgurski have stated [16] 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 [17] and scanners proposed to prevent XSS vulnerabilities [18]. Some software engineering approaches are also proposed such as WAVES for security assessment. However none of the solutions are not built for the latest developments and would fail if tags are permitted in the web applications. Jayamsakthi et al. [18] provided solutions based on financial and non financial applications but this does not cater for the XSS attacks emerge from various interfaces.

In Nonce spaces [12], the authors propose to use randomized XML namespaces in order to partition the content into different trust classes. The client is responsible for interpreting the namespaces and restricting the content's rights according to a policy that is provided alongside the Web site. The owner of the site can assign the desired trust levels via XPath expressions, and thus, disallow JavaScript code in HTML sub trees that are supposed to contain user contributed content. The mentioned hybrid mitigation techniques offer the most powerful features and the best ratio between parameterization costs and level of protection. However, they share the same *drawback* as the strictly client-based solutions: The requirement to being deployed on user's machines.

Our solution is similar to BEEP [13] and Nonce spaces in that we use a server-provided specification of legitimate JavaScript content and detect when a script has been injected. However, our solution performs *a*ll XSS mitigation functionality on the server-side. It therefore does not require any client-side modifications, and can be applied transparently, without the user even being aware of it. We focus in this paper on the specific case of Cross-Site Scripting attacks against the security of web applications in Server side. This attack relays on the injection of a malicious code into a web application, in order to compromise the trust relationship between a user and the web application's site. If the vulnerability is successfully exploited, the malicious user who injected the code may then bypass, for instance, those controls that guarantee the privacy of its users, or even the integrity of the application itself.

Our contribution of this paper on the specific case of Cross-Site Scripting attacks against the security of web applications in browser side. This attack relays on the injection of a malicious code into a web application, in order to compromise the trust relationship between a user and the web application's site. If the vulnerability is successfully exploited, the malicious user who injected the code may then bypass, for instance, those controls that guarantee the privacy of its users, or even the integrity of the application itself. The main contribution of this paper is that it is the Server-Side Solution that provides Cross Site Scripting protection effectively without relying on web application providers. Server side solution supports a Cross Site Scripting mitigation mode that significantly reduces the number of connection alert prompts while, at the same time, it provides protection against Cross Site Scripting attacks where the attackers may target sensitive information such as cookies and session IDs. We propose a mechanism that limits the amount of information that can be stolen by any single Cross Site Scripting attack. This paper describes the possibilities to filter JavaScript in Web applications in server side protection.Server side solution effectively protects against information

leakage from the user's environment. Also it is possible to consider the fact that the web applications are built for various purposes. For instance we have researchers web application, social networking web application, e-mail application, e-commerce application etc. Each web application is built with different requirements for performance, security mechanisms, internationalization and scalability to serve its customers.

This paper proposes a Cross-site Scripting Protection System in server side which is based on passive HTTP traffic monitoring and relies upon the following observations:

1. There is a strong correlation between incoming parameters and reflected XSS issues.
2. The set of all legitimate JavaScript's in a given web application is bounded.

III. PROPOSED ARCHITECTURE

This proposed architecture describes each module in detail and derives test plan for the paper entitled "A Server Side Solution for Protection of Web applications from Cross site scripting attacks". In this proposed Architecture (see Figure 1) present Server Side Solution to mitigate Cross Site Scripting attacks. The main purpose of Server side solution is that it is effectively reduces Cross Site Scripting attacks. The Server-Side Solution that provides Cross Site Scripting protection without relying on web application providers.

This architecture describes the overall problem and elaborates on the possibilities to filter JavaScript in Web applications in server side protection. The cross site scripting (XSS) attack is based on the possibility to insert malicious JavaScript code into pages shown to other users. Due to that reason XSS filtering malicious JavaScript code is necessary for any Web application. This paper describes the possibilities to filter JavaScript in Web applications, and also a filtering XSS architecture is presented that allows Web application developers to filter JavaScript depending on the application need to reduce the danger of successful Cross-Site Scripting attacks. The Server-Side Solution capability to analyze all web pages for embedded links. That is, every time Server-Side Solution fetches a web page on behalf of the user, it analyzes the page and extracts all external links embedded in that page. Because each link can be followed without receiving a connection alert; the impact of Server-Side Solution on the user is significantly reduced. Static links that are extracted from the web page include HTML elements with the HREF and SRC attributes and the URL identifier in Cascading Style Sheet (CSS) files.

The Server-Side Solution allow easy integration into Java based applications the filter provides a simple interface encapsulated in the class JavaScriptFilter.The empty constructor that is, public JavaScriptFilter (String filterConfigFile) that is called with a configuration file, only two methods are provided allowing two different access ways to the filter: public void XSS filter (Reader filterInput, Writer filteredOutput), public String XSS filter (String filterInputString).

A Web Application may then create XSS filter classes which as much configurations as needed, to perform appropriate input or output filtering. Based on these concepts, we extended our system with the capability to analyze all web pages for embedded links. That is, every time client side solution fetches a web page on behalf of the user, it analyzes the page and extracts all external links embedded in that page. When client side solution receives a request to fetch a page; it goes through several steps to decide if the request should be allowed. We have used a technique to determine if a request for a resource is a local link. It is achieved by checking the Referrer HTTP header and comparing the domain in the header to the domain of the requested web page.

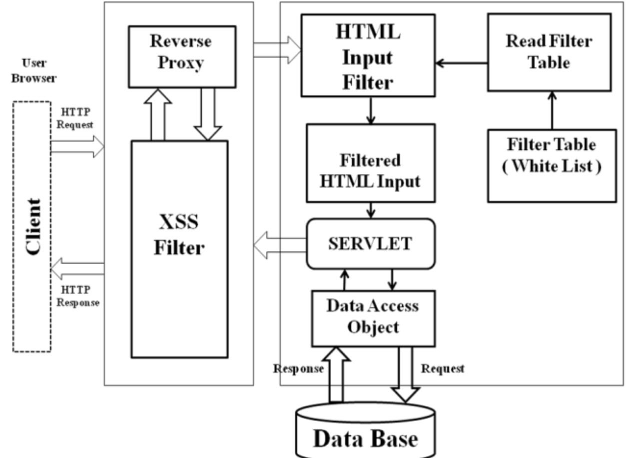

Figure 1: Architecture for Cross-Site Scripting in Server Side

The main components of our proposed architecture are:

1. A JavaScript detection component, which, given the Web server's response and request, is capable of determining whether script content is present or not.
2. A reverse proxy installed in front of the Web server, which is used to getting incoming HTTP request parameter from the user and outgoing HTTP response parameter from the server and subjects them to analysis by the JavaScript detection component.
3. A XSS filter component, which is used to clean harmful script from the HTTP request and HTTP response.
4. A HTML Input filter component is located in front of servelet component ,which is used to inspect escape comments, balance HTML tags, Remove blanks space, protocol attributes from the incoming HTTP request and encode this parameter.
5. A Data Access Object (DAO) component, by using data access objects instead of accessing the data source directly, the type and implementation of the actual data source is decoupled from its usage. This allows moving from one data source to a different data source without having to change the business logic.

IV. IMPLEMENTATION AND DISCUSSION

For implementation of JavaScript filters languages that internally using a Unicode representation of strings are suited best, since they automatically transform national character set characters to the Unicode representation. The Java programming language is used to implement these web applications. The JavaScript filter described in the following section is implemented in Java. This following section describes the main component and implementation of our proposed server side solution.

1. XSS Filter

From a connectional point of view filtering JavaScript to prevent Cross-Site Scripting attacks can be performed on any data sent to an application as input, or can be performed on the output sent by the application to Web browsers, or both .A web security mechanism point of view the whole HTTP request sent to a Web application must be considered input and not only the parameter values that are fed by users into HTML input fields. Cross-Site Scripting filtering for JavaScript means scanning a data stream for specific string patterns considered dangerous and then take appropriate actions like transformation or deletion. There are many character encodings schemes available that are used to represent foreign language characters. The character encoding schemes of the input data for a Web application is normally indicated in the request header generated by the client browser.

The first step in XSS filter is to normalize the input data to specific character encoding. Since standard encodings are not suited to provide a uniform encoding base Unicode should be used instead. Most commonly used encoding is UTF-8. Because the UTF-8 is using a variable length encoding schema additional actions must be taken to avoid the problem of illegal UTF-8 character encodings. The case if a UTF-8 character for which the encoding is one byte long is encoded using two or more bytes which the additional bytes set to zero. A simple XSS filter would not match dangerous characters since the lengths of the character encodings differ. A JavaScript XSS filter must honor the character encoding and make sure that only valid encodings are accepted. JavaScript filters implementing languages that internally using a Unicode representation of strings are suited best, since they automatically transform national character set characters to the Unicode representation. This XSS filter is used to develop by Java programming language that is also often used to implement Web applications.

An important concept is that all links that are statically embedded in a web page can be considered safe with respect to Cross Site Scripting attacks. The attacker does not directly use static links to encode sensitive user data. The reason is that all statically embedded links are composed by the server before any malicious code at the client can be executed. A Cross Site Scripting attack, on the other side, can only succeed after the page has been completely retrieved by the browser and the script interpreter is invoked to execute malicious code on that page. All local links can implicitly be considered safe as well, after all, cannot use a local link to transfer sensitive information to another domain external links have to be used to leak information to other domains. The contribution of our dynamically enhanced XSS protection mechanism, we analyzed the web pages recursively.

2. Reverse Proxy

A reverse proxy is used to get the incoming HTTP request parameter from the web browser and send to the java script detector component. Again the script detector send to the appropriate java script to the reverse proxy then it passes to HTML input filter. The main aim is client connects to the proxy server and also requesting some web services, such as a file, connection, web page, or other resources available from a different server. A reverse proxy is used to passes requests and replies unmodified are usually called a gateway or sometimes tunneling proxy. A proxy can be placed in the user's local computer or at various points between the user and the destination servers on the Internet. A reverse proxy is a Internet-facing proxy used as a front-end to control and protect access to a server on a private network, commonly also performing tasks such as load-balancing, authentication, decryption or caching.

3. Html Input Filter

Html Input Filter is using an own HTML implementation of a HTML parser that is based on pattern matching. Filtering mechanism was necessary since standard HTML parser libraries cannot cope with malformed HTML input. The input parameter first is analyzed using the HTML parser to build up the HTML object tree. Filtering for JavaScript code means scanning a data stream for specific string patterns considered dangerous and then take appropriate actions like transformation or deletion. The removing of harmful script and character encoding of input data for a Web application is normally indicated in the request header generated by the client browser. If it is untrusted data, which can be used by attackers to mislead the application or the JavaScript filter used. A simple java filter would not match dangerous characters since the lengths of the character encodings differ. A Java Script XSS filter must honor the character encoding and make sure that only valid encodings are accepted.

3.1 Steps to Perform Filtering

The HTML input filter process consists of four modules. Such as, escape comments, balance HTML tags, remove blanks and check tags are shown in fig.2. The XSS filtering is done by a filter class that is using the filter table generated by the configuration file reader from the XML configuration file. Finally harmless filtered HTML input is returned to the application for further processing.

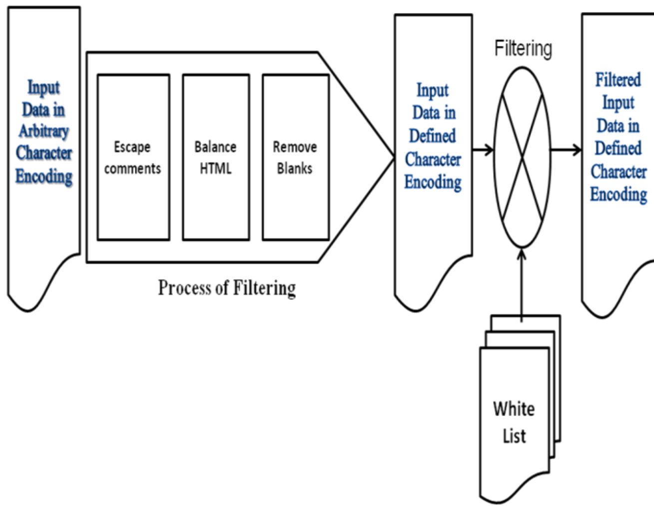

Figure 2: Steps to Perform XSS Filtering

3.1.1 Escape Comments

This method takes the user given string and does a pattern matching for HTML comments [<! -- -- >]. If it matches then it removes the HTML comments from the given input string. The pattern matching is using *java.util.regex.matcher* and *java.util.regex.pattern* class. This method returns a string after filtering out the HTML comments.

It also replaces the following HTML special characters with corresponding equivalents shown in table1.

Table 1: HTML Special Character

Original Character	After Replacement Character
&	&
\	"
<	<
>	>

The below prototype shows the pattern matching regular expressions for this escape comments method is given below in fig.3.

```
protected String escapecomments( String s ) {
   Pattern p = Pattern.compile( "<-!--(.*?)-->",
Pattern.DOTALL );
   Matcher m = p.matcher( s );
   StringBuffer buf = new StringBuffer();
   if (m.find()) {
         m.appendReplacement( buf, "<!--" +
htmlspecialchars( match ) + "-->" );    }
   m.appendTail( buf );
   return buf.toString();  }
```

Figure 3: Code for escape comments in HTML

3.1.2 Balance HTML

This method checks if there are any unbalanced HTML or Java script tags in the given input string. If it founds any, then it checks whether the tags are balanced with proper open and close tags. If the tags are not balanced then, it removed the unbalanced tag from the given string.

3.1.3 Remove Blank Space

This method checks if there is any HTML or Java script tag in the given input string. If it founds any, then it pattern matches it against a set of empty tags. If it matches, then removes the empty tags from the input string. The ability of filter to correctly detect XSS attacks strongly depends on how precisely the JavaScript detection component works in locating JavaScript content within HTML code. In order to verify that our implementation works satisfactorily also in non-traditional ways of embedding script code, we evaluated it on the XSS Cheat Sheet a collection of various XSS attack code snippets, that cover a broad range of nuances regarding filter evasion. All tested examples that work in an unmodified Firefox browser have been successfully detected by our JavaScript detection component.

3.1.4 Check Tags

This method checks whether the given input string has any java script or HTML tags are incomplete. If it has any tags then it compare with "white - list", which is a predefined set of allowable and non-allowable tags. It then removes the harmful tags from the input string. It also does a protocol check and removes from the given input string shown in fig.4.

```
Protected String check tags (String s )  {
 Pattern p = Pattern.compile( "<(.*?)>", Pattern.DOTALL );
   Matcher m = p.matcher( s );
   StringBuffer buf = new StringBuffer();
   while (m.find()) {
     String replaceStr = m.group( 1 );
   m.appendReplacement(buf, replaceStr); return s; } }
```

Figure 4: Checking open and close tag in HTML

5. White List Table

White-List table filtering the pre-defined patterns is specified, since new attack patterns are used they most probably do not match any of the allowed patterns if these patterns where constructed carefully. This white-list provides good protection from attacks they can be hard to specify depending on the application use case. Most of the Web applications are simple with respect to the input expected, so that white list definition is simple in many cases also. A useful mechanism of White-Lists is that if new tags are available they explicitly must be included into the list, thus a sound decision can be made before the new tag is processed by the application. This white list also increases the overall security, because of its security shortcomings Black-List filtering will not be considered in the following. There are two steps must be performed for HTML filtering are: Identifying the HTML elements, checking whether the HTML elements are on the list of allowed entities.

For identifying HTML elements or markup appropriate parsing of the input is necessary, which technically is performed by pattern matching. The HTML parsing techniques can also cope with malformed HTML input, since attackers may intentionally send malformed requests to the application. In general most of the HTML parser must be constructed to fail into a safe state that allows filtering out the malformed parts, so that the filter may not be subverted by a fooled parser. Most of publicly available HTML parser lacks this property and refuse to operate on malformed HTML input rendering them unusable for implementing a JavaScript filter.

6. Data Access Object (DAO)

Data access objects provide the portability for applications from one data source to another data source. Many modern applications require a persistent database for their objects. There are currently several common types of databases: Flat files, object-oriented databases and relational databases, with relational databases being the most widely used. Unfortunately these types of databases are accessed in a very different way. Even databases of the same type, such as relational databases behave very similar but not exactly identical. By using data access objects instead of accessing the data source directly, the type and implementation of the actual data source is decoupled from its usage. This allows moving from one data source to a different data source without having to change the business logic. The use of the data access object may also encourage the use of additional functionality with proxies, such as caching. This may improve performance, depending on the use of the data.

V. CONCLUSION

Cross Site Scripting vulnerabilities are being discovered and disclosed at an alarming rate. Cross Site Scripting attacks are generally simple, but difficult to prevent because of the high flexibility that HTML encoding schemes provide to the attacker for circumventing server-side input filters. Several approaches have been proposed to mitigate Cross Site Scripting attacks.

The main advantage of these solutions is that they rely on service providers to be aware of the Cross Site Scripting problem and to take the appropriate actions to mitigate the threat. In this paper, we present Server Side Solution to mitigate Cross Site Scripting attacks. The main contribution of server side solution is that it is effectively reduces Cross Site Scripting attacks. The Server-Side Solution that provides Cross Site Scripting protection without relying on web application providers. Server Side Solution supports a Cross Site Scripting mitigation mode that significantly reduces the number of connection alert prompts while, at the same time, it provides protection against Cross Site Scripting attacks where the attackers may target sensitive information such as cookies and session IDs. It acts as a web proxy to protect Cross Site Scripting attacks in the server side.

REFERENCES

1. Richard Sharp and David Scott," Abstracting Application Level Web Security," In Proceedings of the 11th ACM International World Wide Web Conference (WWW 2002), May 7-11, 2002.
2. Peter wurzinger, Christian Platzer, Christian Ludl, and Christopher Kruegel,"SWAP:Mitigating XSS Attacks using a Reverse Proxy," In proceedings of the 2009 ICSE Workshop on Software Engineering for secure systems,pp.33-39,2009.
3. Engin Kirda, Nenad Jovanovic, Christopher Kruegel and Giovanni Vigna,"Client-Side Cross-Site Scripting Protection," ScienceDirect Trans.computer and security ,pp.184-197,2009.
4. Acunetix. Acunetix Web Vulnerability Scanner. http://www.acunetix.com/, 2008.
5. Nao Ikemiya and Noriko Hanakawa, "A New Web Browser Including A Transferable Function to Ajax Codes", In Proceedings of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06), Tokyo, Japan, pp. 351-352, September 2006.
6. V. Felmetsger, N. Jovanovic, D.Balzarotti, M. Cova, E. Kirda, and C. Kruegel,"Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," In IEEE Security and Privacy Symposium, 2008.
7. Ravi Sandhu and Joon S. Park, "Secure Cookies on the Web", IEEE internet computing, Volume 4, pp. 36-44, July/August 2000.
8. Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad JovanovicNoxes, "A Client Side Solution for Mitigating Cross-Site Scripting Attacks", In Proceedings of the 2006 ACM Symposium On Applied Computing (SAC'06), Dijon, France, pp. 330-337, April 2006.
9. O. Ismaill, M.E. Youki, K. Adobayashi, S. Yamaguch, "A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability", In Proceedings of the 18th International Conference On Advanced Information Networking And Application (AINA'04), Fukuoka, Japan, Volume 1, pp.145-151, March 2004.
10. G.Vignaand, Christopher Krugel, and William Robertson, "A Multi-Model Approach to the Detection of Web Based Attacks", Computer Networks, Volume 48, Issue 5, pp. 717-738, August 2005.
11. E. Kirda, C. Kruegel, and N. Jovanovic, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities", In Proceedings of the 2006 IEEE Symposium on Security and Privacy(S&P'06), California, U.S.A, pp. 27-36, May 2006.
12. Hao Chen and Matthew Van Gundy,"Using randomization to enforce information flow tracking and thwart crosssite scripting attacks,"In Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS), 2009.
13. T. Jim and N. Swamy and M. Hicks. ,"BEEP: Browser- Enforced Embedded Policies," In 16th International World Wide Web Conference (WWW2007), Banff, 2007.
14. C. Kruegel, E. Kirda, G. Vigna P. Vogt, F. Nentwich, and N. Jovanovic, "Cross site scripting prevention with dynamic data tainting and static analysis," In 14th Annual Network and Distributed System Security Symposium (NDSS), 2007.
15. CERT "Advisory CA-2000-2002- Malicious HTML Tags Embedded in Client Web Requests" http://www.cert. org/advisories/CA-2000-02.html, 2000.
16. Sy-Yen Kuo, Yao-Wen Huang, Chung-Hung Tsai, and D. T. Lee,"Non Detrimental Web Application Security Scanning", In Proceedings of 15th International Symposium on Software Reliability Engineering (ISSRE'04), France, pp. 219-230, November 2004.
17. Chung-Hung, Tsai Yao-Wen Huang, Shih-Kun Huang, and Tsung-Po Lin, "Web Application Security Assessment By Fault Injection and Behavior Monitoring", In Proceedings of the 12th international conference on World Wide Web, Budapest, Hungary, pp. 148 – 159, May 2003.
18. Dr.M.Ponnavaikko and Jayamsakthi Shanmugam, "A Solution to Block Cross Site Scripting Vulnerabilities Based on Service Oriented Architecture", In Proceedings of 6th IEEE international conference on computer and information science (ICIS 07) published by IEEE Computer Society in IEEE Xplore, Australia, pp. 861-866, July 11-13, 2007.
19. Martin Johns, Bjorn Engelmann, and Joachim Posegga,"XSSDS: Server-side Detection of Cross-Site Scripting Attacks," proc. IEEE Computer Security Applications Conference, pp. 335–343, October 2008.
20. Yasuhiko Minamide, "Static Approximation of Dynamically Generated Web Pages," In Proceedings of the 14th International Conference on World Wide Web, Chiba, Japan, pp. 432–441,May 2005.
21. WhiteHat Security. Website Security Statistics Report. http://www.whitehatsec.com/home/ resource/stats.html, 2008.
22. B. (BK) Rios. Google XSS. http://xs-sniper.com/ blog/2008/04/14/google-xss/, 2008.
23. Phishmarkt :: de. http://baseportal.com/ baseportal/phishmarkt/de, 2006.
24. Phishmarkt::at. http://baseportal.com/baseportal/phishmarkt/at, 2007.
25. S.-Y. Kuo, Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, and D. Lee," Securing Web Application Code by Static Analysis and Runtime Protection," In Proceedings of the 13th International World Wide Web Conference (WWW 2004), May 2004.

AUTHORS PROFILE

Mr.A.Duraisamy received his M.E degree in Computer Science and Engineering from College of Engineering Guindy, Anna University Main Campus Chennai, India in 2010 and B.E degree in Computer Science and Engineering from Anna University, Chennai, India in 2006. He is currently working as a Teaching Fellow in University College of Engineering, Tindivanam, Tamilnadu, India.
His areas of interest are: Web Application Security, Image Processing, Networking, Bio-Metrics, Cloud Computing and Data Base Management System. He has published Three Papers in International Journals, Three National Conferences and Life Member in Indian Society for Technical Education.

Mr.M.Sathiyamoorthy received his M.E degree in Computer Science and Engineering from College of Engineering Guindy, Anna University Main Campus Chennai, India in 2007 and B.E degree in Computer Science and Engineering from Madras University, Chennai, India in 2003. He is currently working as a Teaching Fellow in University College of Engineering, Tindivanam, Tamilnadu, India.
His areas of interest are: Web Services, Web Application Security, Cryptography and Network Security, peer to peer Networking and Data Base Management System. He has published One Paper in International Journal. He worked in Tata Consultancy Services Ltd, Chennai as Assistant Systems Engineer for 2.5 years.

Mr. S.Chandrasekar received his B.E. Degree (CSE) from Thiruvalluvar College of Engineering and Technology in 2005. He obtained his M.E degree (CSE) from Rajalakshmi Engineering College, Chennai, India in 2009.
He is currently working as a Teaching Fellow with University College of Engineering, Tindivanam, A constituent College of Anna University, Chennai, India.
His current research interests include Network and security, Mobile Computing, Image Processing, Web Applications Security and Database Management System. He has presented a Paper in National Conference.