A NOVEL FRAMEWORK TO ALLEVIATE DISSEMINATION OF XSS WORMS IN ONLINE SOCIAL NETWORK (OSN) USING VIEW SEGREGATION

P. Chaudhary, B.B. Gupta

Abstract: In this paper, we propose a client-server based framework that alleviates the dissemination of XSS worms from the OSN. The framework initially creates the views corresponding to retrieved request on the server-side. Such views indicate that which part of the generated web page on the server can be accessed by user depending on the generated Access Control List (ACL). Secondly, JavaScript attack vectors are retrieved from the HTTP response by referring the blacklist repository of attack vectors. Finally, injection of sanitization primitives will be done on the client-side in place of extracted JavaScript attack vectors. The framework will perform the sanitization on such attack vectors strictly in a context-aware manner. The experimental testing of our framework has performed on the two platforms of open source OSN-based web applications. The observed detection rate of JavaScript attack vectors was effective and acceptable as compared to other existing XSS defensive methodologies. The proposed framework has optimized the method of auto-context-aware sanitization in contrast to other existing approaches and hence incurs a low and acceptable performance overhead.

Key words: *Cross-Site Scripting (XSS) worms, online social network security, URI links*

Received: April 11, 2016 **DOI:** 10.14311/NNW.2017.27.001
Revised and accepted: October 3, 2016

1. Introduction

In the recent years, Online Social Network (OSN) have gained significant popularity as these have become intertwined into the daily life of people as the digital virtual place that empower communication. It is a virtual place, where people create their own profile, find new friends and re-establish the lost connections based on the common attributes and behavior. The popular OSN utilized nowadays is Facebook [7,24] with over more than 1 billion active users. Other famous OSNs are Google+ [11,12] with more than 235 million active users; Twitter [9] has over 200

Pooja Chaudhary; B.B. Gupta – Corresponding author; National Institute of Technology, Kurukshetra (pin code: 136119), Haryana, India, E-mail: pooja.ch04@gmail.com bbgupta.nitkkr@gmail.com

million active users and LinkedIn [22] with more than 160 million active users. As user share every type of information on OSN platform ranging from personal to professional; therefore, such networks suffer from various categories of cyber-attacks. XSS worms have turned out to be a plague for the OSN based web applications like Facebook, Twitter, LinkedIn, etc. Cross Site Scripting (XSS) [13–15] attack has become one of the dangerous threats to the OSN. XSS is a type of code injection attack in which attacker injects judiciously crafted malicious JavaScript code through the input parameters at the client-side. It is done in order to cause harmful actions of the web applications and accomplish the attacker's objectives like cookie stealing, session token theft or to launch other attacks. It can be categorized into 3 types [16, 17]: 1) Persistent XSS attack in which attacker permanently inserts malicious scripts into the server. After that, when web pages is loaded at browser then malicious scripts get executed and results into XSS attack; 2) Non-persistent attack in which attacker lures the victim to click on illicitly crafted URL which leads to the execution of malicious script included in this URL; 3) DOM-based XSS attack [18,19] is caused because client-side scripts dynamically alter the DOM structure of web page in order to run malicious scripts.

1.1 Various security challenges on OSN platform

As OSN usage has been tremendously increased in the digital world, its users unknowingly become exposed to the many threats to their security and privacy. Users are unaware of the security risks which exist in these types of communications, including privacy risks, identity theft, malware, fake profiles (also in some cases referred to as sybils or Socialbots), and sexual harassment, among others. As the use of OSNs becomes progressively more embedded in users' daily lives, personal information becomes easily exposed and abused. This information may include relationship status, DOB, school name, email address, phone number, and even home address. This information if put into wrong hands, can be used to harm users both in the virtual world and real world. Various security issues exist on the OSN platform [10]. Some of them are shown in Fig 1. As reported by the White Hat Security (2015) [33]. XSS secure 3rd position among all dangerous web application vulnerabilities. According to the survey done by the OWASP [25] in 2013, the problem causing greatest infection is the Cross-Site Scripting (XSS) attack and is accordingly the third top most weakness among the top ten susceptible threats.

1.1.1 XSS attack is very challenging to detect

The three main factors that affect the XSS vulnerability are 1) users' behaviour means the probability with which user visits a friend's profile than a stranger; 2) highly concentrated architecture of OSN groups; 3) group size. Since the probability with which a user visits friend's profile is higher therefore, its propagation speed is faster than the other worms. Amplitude of damage from XSS depends on the sensitivity of data breached by the attacker. Unfortunately, many OSN users are unaware of the security risks which exist in communications online like XSS. Hence, information comprising personal data becomes easily accessible and abused by the adversary. It can be used to harm user in virtual world and real world also. In 2005, Samy worm was the first worm which exploits the XSS vulnerability [8].

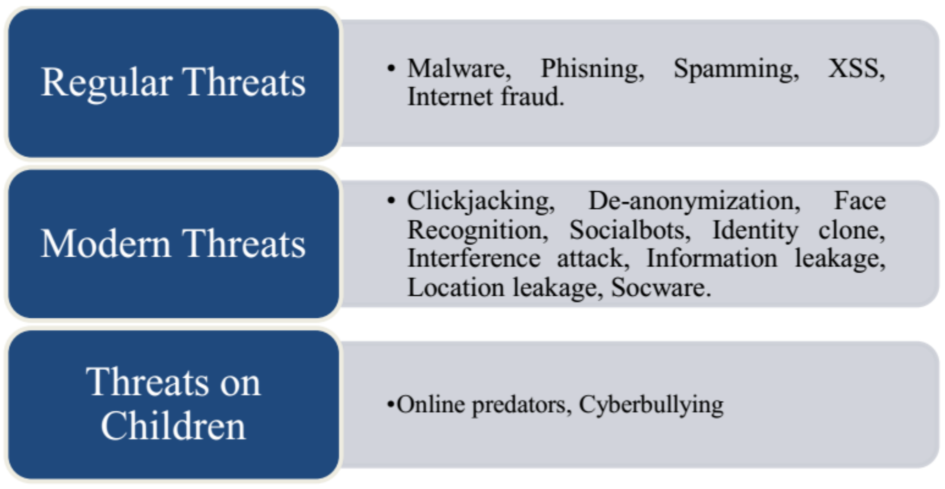
Regular Threats
• Malware, Phisning, Spamming, XSS, Internet fraud.
Modern Threats
• Clickjacking, De-anonymization, Face Recognition, Socialbots, Identity clone, Interference attack, Information leakage, Location leakage, Socware.
Threats on Children
•Online predators, Cyberbullying

Fig. 1 *Security and Privacy issues in OSN platform.*

Over a time period of 20 hours, Samy worm has infected one million users on MySpace social networking site as shown in Fig. 2. Since it triggers by the attacker

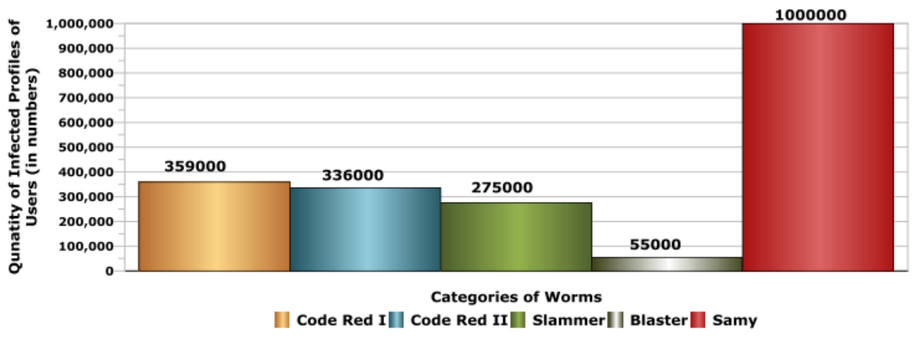

Fig. 2 *Total number of infections after 20 hours by Samy worm.*

through the injection of JavaScript code that looks similar to the legitimate code, therefore, it is a tedious task to make a differentiation between the two code snippets. Consequently, XSS attack is more difficult to catch on highly clustered OSN platform.

1.2 State-of-art techniques

Numerous XSS defensive solutions were proposed by the researchers for detecting and alleviating the effect of XSS vulnerabilities from the different platforms of Web applications. Blueprint [31] provides robust protection from the XSS attacks in spite of having irregular browser parsing behavior. However, it requires modifications at both the sides, i.e. client and server side. It does not protect against the non-scripting attacks. It is difficult to apply as it changes the way application generates the HTML content for browser. Li et al. [32] uses machine learning algorithm to classify malicious Web pages from benign Web pages in OSN-based web applications. However, its effectiveness depends on the training phase to build up the feature database. Saner [2] make use of both static and dynamic analysis for detecting the presence of malicious scripting code in the source code of web applications. However, its deployment cost is high. Nandji et al. [23] proposed a client-server based architecture to enforce document structure integrity. It combines runtime tracking and randomization to thwart XSS attack. However, it is not effective to detect the DOM-based XSS attack. Pelizzi [26] proposed a client-side XSS filter, XSSFilt, which could discover non-persistent XSS vulnerabilities. This filter identifies and thwarts portions of address URL from giving an appearance in web page. This filter could also discover partial script insertions. XSS Auditor [3] is a filter that realizes equally extraordinary performance as well as high accuracy via jamming scripts following the HTML parsing and prior to execution. The filter can simply spot the components of the response which are considered as a script. It also scans the Document Object Model (DOM) tree generated by the HTML parser for the clear interpretation of the semantics of the bytes. Cao et al. [4] proposed a technique (i.e. Path Cutter) that works by restricting the two main steps in the proliferation of the XSS worm: 1) access to DOM, at client side, to different views. 2) Illegitimate HTTP request to the OSN server. Path Cutter is a server side approach which attempts to reduce the impact of the XSS worm by constraining its dissemination path. However, it is not able to protect each and every view from being infected from the XSS attack vector. Reis and Gribble [27] have designed browser architecture based on the web applications instances to improve the reliability of multi-process browser architecture. To achieve this, they have identified program abstractions in the web browser by preserving their semantic structure. These instances can be used to improve browser robustness and performance. Tang et al. [29] designed a client-side architecture named as Alhambra, which enforces security policies to automatically defend browser from wide range of attacks like XSS. It uses policy enhanced components to enforce policies for the restriction of execution of malicious content in the web page. Then, it uses testing component to test for the presence of the XSS attack as reply system. However, it cannot handle re-ordering of non-deterministic function calls like random() within the same script. Cox et al. [5] have designed a web browsing system named Tahoma, to improve the safety and security of the web users. In this system, each web application is allowed to execute in its own sandboxed process to remove the need to trust on web browser and services that they retrieve. For this, it inserts new security layer browser operating system (BOS) which manages web applications and its sandboxed process. For security of web application, BOS restricts the resources and sites to which web application can communicate. Its evaluation revealed that its can prevent web application from 87% of vulnerabilities that has infected Mozilla web browser. However, the existing techniques pay their attention towards minimizing the effect an XSS worm can cause, by restraining it to a particular view and blocking its proliferation path. However, an attacker may capture the control of a specific view and successfully steal the important information about the user, to achieve his objectives. These techniques also incur high rate of false positive and false negative.

1.3 Our contribution

To overcome such performance issues, this paper presents the view segregation based defensive framework that mitigates XSS attack on the OSN platform. It is a client-server based framework. Firstly, it learns about the Access Control List (ACL) in offline mode. Secondly, at the server side, it performs view generation and isolation process. Then, at the client side, it implements context-sensitive sanitization of extracted malicious JavaScript code after the successful request authentication. The rest of the paper is organized as follows: In Section 2, we describe the detailed information of our proposed technique. In Section 3, we provide the implementation details. Finally, Section 4 concludes the paper.

2. Proposed framework

In this section, we provide a brief description of our proposed framework which is a based on the view segregation approach. It is a novel framework which aims at protecting each part of the OSN based web application from XSS attack by performing 2 main steps: first, through the extraction of malicious JavaScript code from each part of web application, second, performing sanitization of malicious scripts extracted in first step. Next, we outline the key concepts used by our framework and the strategy used by our method to mitigate XSS worms. Finally, we describe how the implementation of our framework on OSN platform can help in protecting the users from XSS attack.

2.1 Abstract design overview

Our proposed framework aims to provide protection to each view of the Web application from XSS attack. It restricts attacker to gain access to any view and become capable of stealing sensitive information corresponding to that view like session token, cookie information and any other personal information of the user. Our framework executes in two phases: Learning phase and Identification phase. In the first phase, we distribute the OSN web application into different views and Access Control List (ACL) is rehearsed to apprentice all the privileges/rights a particular view can secure. In the second phase, firstly it validates the action originated by the particular view to ensure that the corresponding view possess the capability to perform that action or not. If action is validated then it discovers the malicious JavaScript nodes in the parsed HTML document. Secondly, it applies context-sensitive sanitization method on these extracted JavaScript nodes and finally returns the sanitized document to the user. By doing this we ensure that each view is protected against the execution of malicious JavaScript code that will trigger XSS attack. Otherwise, request is denied. Fig. 3 illustrates the abstract view of our proposed model. Now let us discuss the prime concepts used in our model. These are defined below:

- *Views*: It may be defined as sandboxed process used to implement the part of the Web application. At the client-side, it will appear as the Web page or a part of it. For example, when user access example.com/options and example.com/update, it can be considered as those two to be from different views.

- *Actions*: Actions may be considered as the tasks executed by the View. For example there may be a request from view X to write a comment on X's comment area. Actions may be considered as the rights given to a view to perform accordingly. To validate the actions initiated from the view Access Control List (ACL) is used.

- *Malicious JavaScript nodes*: After the construction of HTML parse tree, malicious JavaScript nodes represent script node that take input from user. These nodes help in implanting of untrusted code in the legitimate scripts as string or literals. This is a common practice of web application to allow user input in JavaScript variables for modifying performance of scripts. However, this leads to a successful embedding of malicious user input which outcomes into XSS attack. For example, <script> alert(document.cookie);</script> or the use of functions such as eval() or innerHTML() to invoke dynamic content that initiate dangerous parsing behavior.

- *Context Sensitive Sanitization*: Sanitization is method for removing malicious scripts/data from the used supplied input. It is done to ensure that input data are in the correct syntactic and semantic format as acceptable by the web application. Context sensitive sanitization apply sanitizers' functions on each untrusted variable (i.e dynamic content like JavaScript) according to the context they are used. For example, filter_xss() is a function used to filtered out HTML string to prevent from XSS attack. There may be different context present in an HTML document like Element tag, attribute

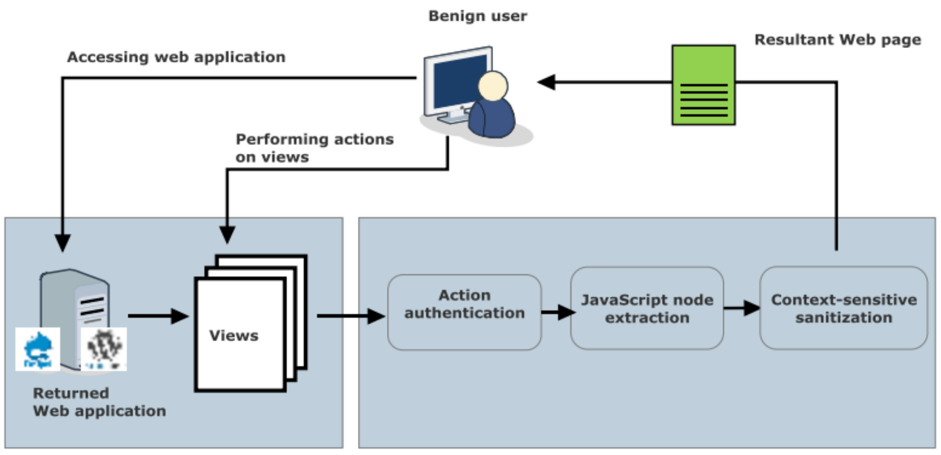
Benign user
Accessing web application
Resultant Web page
Performing actions on views
Views
Returned Web application
Action authentication
JavaScript node extraction
Context-sensitive sanitization

Fig. 3 *Abstract design view of the Proposed Model.*

value, Style Sheet, Script, anchors, href, etc. These all context can help the attacker to launch XSS attack by embedding malicious code into the trusted code. Table I highlights HTML features used to inject the XSS vector into the Web application.

HTML Tag	Attribute Type	Expected Value	Attack Request Instance
Link	Href	Text/CSS	<link type = "text/css" href = "attack_malicious URL" />
Body	Background	Image	<body background: url('attack_malicious URL')>
Img	Src	Image	<img src = "attack_malicious URL" />
Script	Src	text/JavaScript	<script type = "text/javascript" src = "attack_malicious URL" > </script>
Style	Type	text/JavaScript	<STYLE TYPE="text/javascript ">alert('XSS');</STYLE>
Input	Src	Image	<input type = "image" src = "attack_malicious URL" alt = "Submit"/>

Tab. I *HTML elements and their attributes used to initiate XSS attack.*

Fig. 4 highlights the step-wise working process of our proposed framework in the form of a flow chart. It provides deep insight into the implementation details of the model.

2.2 Detailed design overview

In this sub-section, we provide detailed description of each component present in the abstract view to deliver deep understanding of our proposed model. Fig. 5 shows the detailed architecture of our proposed framework. It defines how different modules interact with each other. Our framework executes in two phases: Learning phase and Identification phase.

- *Learning phase*: Learning phase plays a vital part in determining the efficiency of our proposed technique. This phase is implemented offline. In this phase, ACL list is lay down. Moreover, system learns about all the actions that a particular view can originate. Access Control List is maintained to carefully authenticate each action during identification phase. Basically, it provides a mapping between each view and its corresponding actions. It should be constructed carefully so that system will be more powerful in defining that the view has the privileges to perform actions or not. ACL contains the entry in the form of ¡User ID, privileges¿, User ID denotes the user's cookie information and privileges denotes the actions a user with that User

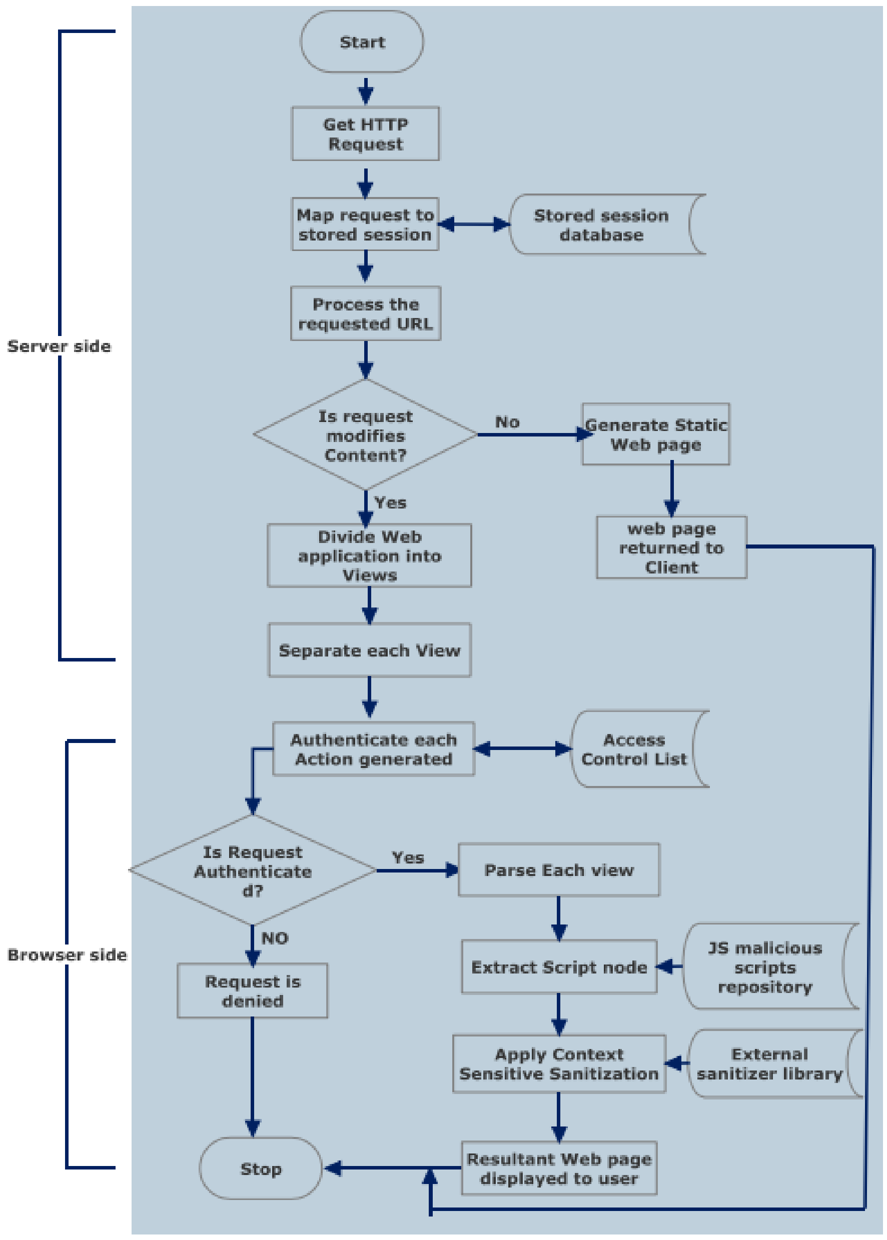
Server side
Browser side
Start
Get HTTP Request
Map request to stored session
Stored session database
Process the requested URL
Is request modifies Content?
No
Generate Static Web page
web page returned to Client
Yes
Divide Web application into Views
Separate each View
Authenticate each Action generated
Access Control List
Is Request Authenticate d?
Yes
Parse Each view
NO
Request is denied
Extract Script node
JS malicious scripts repository
Apply Context Sensitive Sanitization
External sanitizer library
Resultant Web page displayed to user
Stop

Fig. 4 *Flow chart depicting working process of the proposed model.*

ID can execute. It is maintained at the client-side for validating each action. For example, consider some of the pseudo entries in ACL list shown in Tab. II. First column indicates the type of the user (i.e. developer, anonymous user, authenticated user, client, etc.) second column represents path of specific part of the web application and third column denotes all the privileges secured by that type of user.

User type	URL path	Privileges
Developer	www.drupal.com/dashboard	All privileges
Client	www.drupal.com/dashboard	All, except code modifications
Authenticated user	www.drupal.com/dashboard	Read, write, settings configuration
Anonymous user	www.drupal.com/dashboard	Read

Tab. II *An illustration of entries in ACL list.*

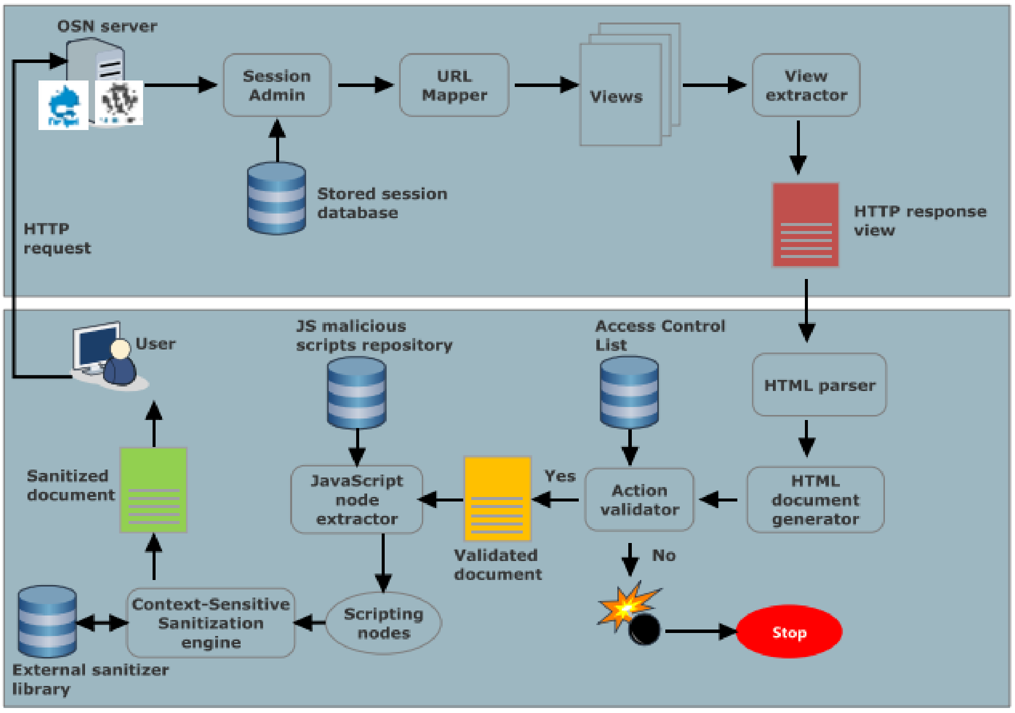

Fig. 5 *Detailed Architecture of the proposed model.*

- *Identification Phase*: During this phase, system works in the real environment. It is the most important phase as XSS attack detection is accomplished here with the help of ACL list prepared in the training phase. This phase is executed at both side client and server. The following steps are performed at server side to achieve view segregation:

  1. When OSN server receives the HTTP request from the Client side then it is firstly dispatched to the session admin where it is mapped with

stored session depending on the user's cookie information (i.e. user's login credentials).

2. After this, Request is checked to ensure whether it alters the server content or not. For example, a request arises to post a comment. If it does not alter the web content then OSN server produces the static Web page and returns it to the browser.
3. Otherwise, multiple views are generated of the Web application and request if forwarded to view which can process it. It is achieved with the help of URL mapper which dispatch HTTP requests to their corresponding view for generation of HTTP response web page.
4. Depending upon HTTP request, identified view is taken out via view extractor and returned to the client as HTTP response. However, steps which are given below are implemented at the client side.
5. When browser gets the HTTP response firstly it is parsed by the HTML Parser to generate the DOM (Document Object Model) tree so that browser can deduce the web content signified by the HTML document. Thus, parsed document is passed to the HTML document generator process the web content.
6. HTML document is forwarded to the action validator to check that the view which initiated the action is capable to perform that action or not by using ACL list prepared during Training period. If it is not authenticated then it means attacker is trying to breach the view privileges and trying to launch the XSS attack. Therefore, access is denied.
7. Otherwise, it discovers the malicious JavaScript nodes in the generated DOM tree with the help of the JS malicious scripts repository. Finally, extracted JavaScript nodes are then dispatched to the Context Sensitive Sanitization engine where it is sanitized according to the context in which each script node is embedded. After sanitized each node, again they are injected into the document and lastly, sanitized HTML Document is displayed to the user.

Next, we provide the comprehensive description of each of the major components involved in the abovementioned process.

2.2.1 Major components

This sub-section provides the brief introduction to the main components used in our proposed framework.

Server-side components: These components are implemented at the server side to accomplish view segregation process. These are described below:

- *Session admin*: It is responsible for implementing 2 basic functions: 1) It creates the session for the OSN user when user registers itself for the first time, by generating cookie corresponding to user's login details. Moreover, it provides storage space for these sessions and responsible for controlling and monitoring these stored sessions; 2) it performs the mapping of the stored session with the user's login credentials. Basically it stores and maps information given to OSN server at the time of registration or at the time of login.

- *URL mapper*: Web application can be isolated into distinct views on the basis of its content by using two strategies: first, we can isolate web application into different views on the basis of content semantics. For example, blogging social site like WordPress can be divided into views via blog names. Second, isolation can be achieved on the basis of requested URL. For example, www.wordpress.com/blog1/options and www.wordpress.com/blog2/updates can be considered as two different views of web application WordPress.
  In our proposed framework, we use second strategy to divide web application into different views. Therefore, URL mapper is responsible to map requested URL to the part of the web application that is capable to process the request. It is responsible for generation of the view that is requested by the user, on the basis of requested URL. For example, the requested URL www.wordpress.com/blog1/post maps to the post part of the blog1 web page of WordPress web application.
- *View extractor*: It is responsible for the withdrawing the requested view and returns it to the client as the HTTP response. It extracts the view according to the HTTP request. To ensure correct extraction of requested view, it performs isolation of the content of particular view from the content of another view of same web application. It is achieved to prevent unauthorized access from one view to another. To implement this, our framework encapsulates each view into virtual domain. For example, each view from blog1.com can be encapsulated into virtual domain as segblog1.com. It is embedded into main page using iframe feature provided in HTML5. It ensures that a user having access to blog1.com cannot gain access to segblog1.com illicitly. This will enforce Same Origin Policy (SOP) on view segregation of web application.

Algorithm 1 describes the algorithm implemented at the server side to perform the view generation operations by using session admin and URL mapper. This algorithm performs the view generation and extraction operation by performing the following steps:

1. Session variable (Str_S) is an array which is used to hold the session related to user cookie information. For each HTTP request (H_I), Str_S will store the session (S_I) and then server checks if request modifies the Web application content, if not, it generates the static web page (W_I) and returns it to the client as HTTP response (H_1). Otherwise, V_I holds the view generated for severing the HTTP request.
2. URL mapper maps the requested URL $\in (U_1, U_2, U_3, \ldots, U_N)$ to the respective view and finally, that view is returned to the client as HTTP response.

Client-side Components: In addition to the server-side components, our proposed framework also includes some of the major components which are positioned at the client-side. These components are illustrated below:

Algorithm 1 Server-side View Generation.

```
**Input**: Set of HTTP Request $(H_1, H_2, H_3, \dots, H_N)$.
**Output**: Set of Views $(V_1, V_2, V_3, \dots, V_N)$ or Web Pages
**Start**
Str_S ⇐ NULL; /*list for storing session for users.
**for Each HTTP request as $H_I$ do**
  Generate session $S_I$;
  Str_S⇐ $S_I \cup$ Str_S;
  **if** $H_I$ modifies OSN server content **then**
    Generate View $V_I$
    Map each $V_I$ to their corresponding URL $\in (U_1, U_2, U_3, \dots, U_N)$
    **Return** $H_1 \Leftarrow V_I$
  **else**
    Generate static Web page $W_I$
    **Return** $H_1 \Leftarrow W_I$
  **end if**
**end for**
**End**
```

- *HTML parser*: At the client side, HTTP response is first received by the HTML parser which constructs the Document Object Model (DOM) tree. It is the method which is used by the browser to interpret the HTML documents. During parsing, text code are separated from the executable code and represented in the DOM tree as data node and script node respectively. Browser interprets these nodes to display the content in the resultant HTML document to the user. For instance, consider the following code snippet as shown in Algorithm 2.

Algorithm 2 Example shows vulnerable HTML code produced by vulnerable server.

```
<html>
<body>
<div name= “val” onClick= “my()”> Click Me!!! </div>
function my(){
document.getElementByName(“val”).innerhtml= “hello” + “$_GET(‘name’)” +
“you are” + “$_GET(‘age’)” + “years old”;}
</script>
</body></html>
```

In the above example, untrusted user input is applied at $_GET(‘name’) and $_GET(‘age’). The parse tree generated for the above code snippet is shown in Fig. 6. Each node of the tree represents HTML tags or text. This tree will be processed to determine script node embedded in to the web page.

- HTML document generator: This component is responsible for storing and interpreting the web content represented by parsed document. Its main ob-

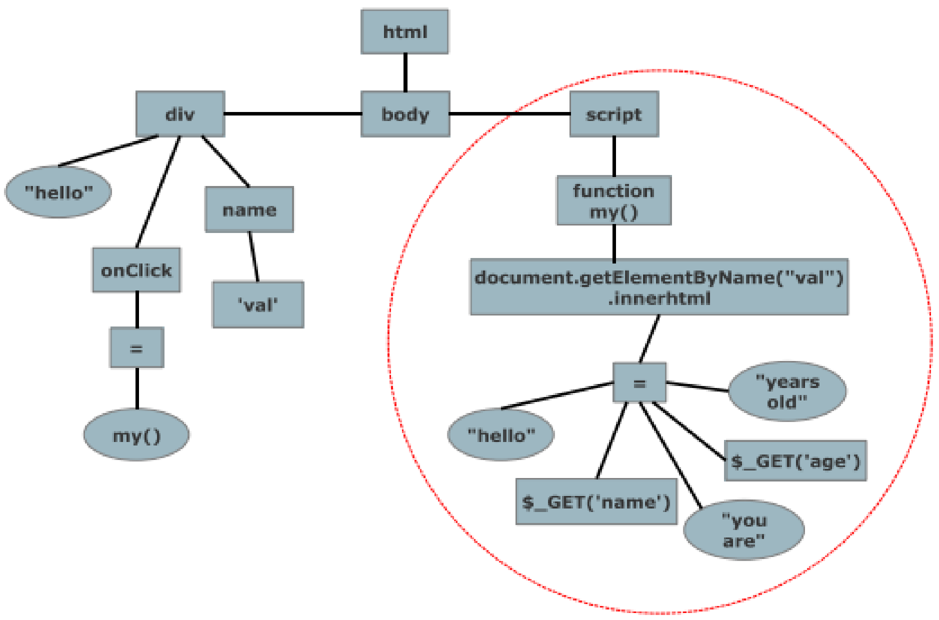

Fig. 6 *Parse tree generated for the code shown in listing 1.*

jective is to separate visual elements and scripting elements and pass them to other parts of the browser. For example, style elements are passed to the CSS parser and scripts are forwarded to the JavaScript interpreter.

- Action validator: After parsing, HTML document is validated by the action validator which checks the authenticity of the action initiated by the view. It is to ensure that view which has originated the action is able to perform this action or not i.e. it has the capability to execute action or not. It is achieved with the help of the ACL (Access Control List) prepared during learning phase. To examine the action authenticity, first, we retrieve the URL link specified in the referer header of HTTP request as String referer= request.getheader(referer);.secondly, we utilize this extracted link along with type of user to search in the ACL list. If match is found then it checks the corresponding actions/privileges. If initiated action matches with the stored actions in ACL then action is authenticated and request is forwarded JavaScript node extractor. Otherwise, it means that some adversary is trying to breach the security of the view by injecting some XSS attack vectors into the view and finally it is rejected.

- JavaScript node Extractor: On the successful validation of the Parsed HTML document, it is forwarded to the JavaScript node extractor which determines JavaScript node from the DOM tree. These nodes are extracted because trusted scripts may be used by the attacker to inject the malicious code into the web page. For example, in the parse tree shown in Fig. 5, it extract the node with value "¡script¿" (encircled with red color) along with its descendants.

- Context Sensitive Sanitization engine: Sanitization is a method to substitute the untrusted user variable with the sanitized variable. Extracted JavaScript nodes are sanitized according to the context in which they are used in the HTML document. In order to defend against XSS attack, we need to ensure that the all the characters used in the attack vectors must be replace by their escaping codes. Tab. III shows some of the escaping codes corresponding to malicious characters. For instance, attacker inject <script>alert("XSS); </script> attack vector into any input field of the web application then, ' <′ and ' >′ would be replaced with the &#60 and &#62. Algorithm 3 shows a code snippet that will replace these characters.

Characters	Hexadecimal code	Numerical code
"	&#x22	&#34
#	&#x23	&#35
&	&#x26	&#38
'	&#x27	&#39
(	&#x28	&#40
)	&#x29	&#41
/	&#x 2F	&#47
,	&#x 3B	&#59
¡	&#x 3C	&#60
¿	&#x 3E	&#62

Tab. III *Character escaping code.*

Algorithm 4 describes the algorithm used by the context sensitive sanitization engine to sanitize the script nodes. The context sensitive sanitization engine procedure algorithm works as follows:

1. San is a repository which includes the externally available sanitizer's vector, used for sanitization. V_{UT} is an array used to hold the untrusted variables.
2. After extracting JavaScript nodes (J_S), we search for the untrusted variable (X) and store it in the V_{UT}. Now context type identifier is used to determine the context (C_I) of the each X and then apply the sanitizer (S_I) according to the context in which V_I is used.
3. Sanitized variable is store in V_{SI} and then it is merged with the San for more effective results.
4. All the sanitized variables are then injected to the HTML document at their respective extracted locations and modified HTML document is displayed to the user.

Algorithm 5 describes the algorithm executed at the client side to complete all the processes. It uses various variables and ACL to complete its working process. The complete details of the algorithm are given below:

Algorithm 3 Example shows vulnerable HTML code produced by vulnerable server.

```
Public function filter_char_esc($string)

$str = str_replace
(
array(_<‘, _>‘, |‘||, _||‘, _)‘, _(_ ),
array(_&#x3C;‘, _&#x3E;‘, _&#x27;‘, _&#x22;‘, _&#x29;‘, _&#x28;‘)
$string
);
$str = str_ireplace( _%3Cscript‘, __, $str );
return $str;
```

Algorithm 4 Context Sensitive Sanitization engine procedure.

```
Input: Set of JavaScript nodes $(J_{S1}, J_{S2}, J_{S3}, \dots, J_{SN})$.
Output: Sanitized HTML document.
Start
San$\Leftarrow$ list of sanitizers $(S_1, S_2, S_3, \dots, S_N)$;
X$\Leftarrow \varphi$; /*list to hold untrusted variable
for Each extracted node $J_{SI}$ do
  X$\Leftarrow$Untrusted variable;
  $V_{UT} \Leftarrow$ X$\cup V_{UT}$ ;
  for Each $V_{UT}$ do
    $C_I \Leftarrow$Context($V_{UT}$);
    $S_I \Leftarrow (S \in$ San $) \cap (S$ matches $C_I)$;
    $V_{SI} \Leftarrow S_I(V_{UT})$;
    San$\Leftarrow V_{SI}\cup$ San;
  end for
  Insert all sanitized variables in document
  Return document
end for
End
```

1. Each entry in the ACL (A) is denoted as $(A_1, A_2, A_3, \dots, A_N)$ and J_S is an array which is used to store the extracted JavaScript nodes.

2. For each responded view (V_I) performing action AI, it first parses the document as DP and it is passed to the HTML document generator to process the document in D1. After this, it checks whether the VI has the capability to execute that A_I or not (i.e. check the condition $((V_I$ executes $A_I) \cap (A_I \in$ A)). If it is true then JavaScript nodes are extracted and context-sensitive sanitization is applied on these extracted nodes. This is achieved by applying the algorithm as shown in Algorithm 4.

Algorithm 5 Client side sanitization and action authentication.

```
Input: Set of extracted views $(V_1, V_2, V_3, \ldots, V_N)$ or web page $W_I$.
Output: Modified HTTP response $(H_{RM1}, H_{RM2}, H_{RM3}, \ldots, H_{RMN})$
Start
A⇐ Access Control List $(A_1, A_2, A_3, \ldots, A_N)$
$J_S \Leftarrow \varphi$; /*list to store extracted JS nodes
for Each View as $V_I$ do
  $D_P$ Parser($V_1$);
  $D_1$ HTML Document-generator($D_P$);
  if (($V_I$ executes $A_I$) ($A_I \in$ A)) then
    J_S⇐ Extract JavaScript node($D_P$)∪ J_S ;
    $H_{RMI} \Leftarrow$ Context-Sensitive-Sanitization-engine(J_S);
    Return $H_{RMI}$
  end if
  if $A_I \notin$ A then
    Request is denied.
    Return Access Denied
  end if
end for
End
```

2.3 Key strengths of our proposed framework

The proposed framework is a client-server and view segregation based defensive solution to thwart XSS attack via view isolation and sanitization of malicious JavaScript code. Our framework executes by intercepting two critical ways in the proliferation path of XSS worm: 1) illegitimate request to the OSN server; 2) access to the views at client-side. Therefore, instead of restrict attacker to a particular view of the web application, it ensures that attacker is not able to gain access to any view of the web application. In addition to the detection of traditional XSS attack vector payload, our framework is capable to detect attack vector payload constructed using new tags and attributes introduced in HTML5 like audio, video, src, etc. Moreover, we have performed the sanitization of attack vector after the successful determination of its context so that we can implant correct sanitization routine. Also, less work is done in the direction of protecting OSN platform from the plague of XSS attack. Therefore, our framework aims to mitigate XSS attack from OSN platform. The next section will elaborate the implementation details of our framework along with the experimental analysis.

3. Experimental evaluation and analysis

This section discusses the implementation and experimental evaluation outcomes of our client-server side framework. We have implemented our framework in java using Apache Tomcat server [1] as the backend, for mitigating the effect of XSS vulnerabilities from the tested suite of real world web applications. The experiment background is simulated with the help of a normal desktop system, comprising

1.6 GHz AMD processor, 2 GB DDR RAM and Windows 7 operating system. Initially, we manually verified the performance of our framework against five open source available XSS attack repositories [20,28,30,35,36], which includes the list of old and new XSS attack vectors. Very few XSS attack vectors were able to bypass our client-server resident framework. We utilize HtmlUnit [21] HTML parser to generate the parse tree for the extraction of JavaScript nodes. We have tested our proposed system on two open source real world OSN platforms i.e. WordPress [34] and Drupal [6]. This has been done for evaluating the XSS attack vector mitigation capability by the deployed mechanisms on these open source OSN web applications. In terms of accuracy, we estimate what percent of XSS attack payloads are alleviated by our system. We have incorporated the infrastructure of our proposed framework into the OSN application. Tab. IV highlights the configuration and XSS known vulnerability on the mentioned OSN platforms. We have tested and evaluated the detection capability of our proposed solution on two real world OSN web applications namely WordPress and Drupal. We select these applications for accessing the forms to potentially supply modified pages and access the HTML forms. We have also calculated the XSS attack payload detection rate of OXSSD on the utilized two individual OSN-based web applications. This is done by dividing the number of XSS attack payload detected to the number of malicious script exploited for each category of attack vectors. Tab. V highlights the detection rate of two OSN web applications w.r.t. individual category of attack vectors.

Application	Version	XSS vulnerability	Lines of Code
WordPress	3.6.1	CVE-2013-5738	135540
Drupal	7.23	CVE-2012-0826	43835

Tab. IV *Observed experimental results on Drupal.*

The observed results of JS-SAN on two real world web applications corresponding to the chosen categories of JS attack vectors has been shown in the Figs. 7 and 8.

Scripts category⇒ / Application⇓	JS malicious events	Embedded character tags	Obfuscated JS URL	Malicious JS variables	Character encoding script
WordPress	82	91	93	90	94
Drupal	94	91	89	85	92

Tab. V *Detection rate (in % age) of OSN-based web applications.*

3.1 Performance analysis using F-measure

In this section, we discuss the performance evaluation of our framework during XSS worm detection testing on open source OSN platform. We have presented detailed performance analysis of our framework by conducting a statistical analysis method

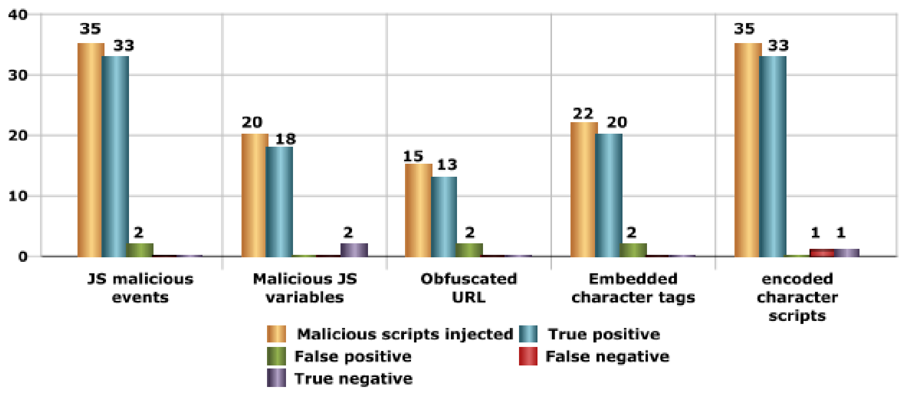

Fig. 7 *Observed experimental results on WordPress.*

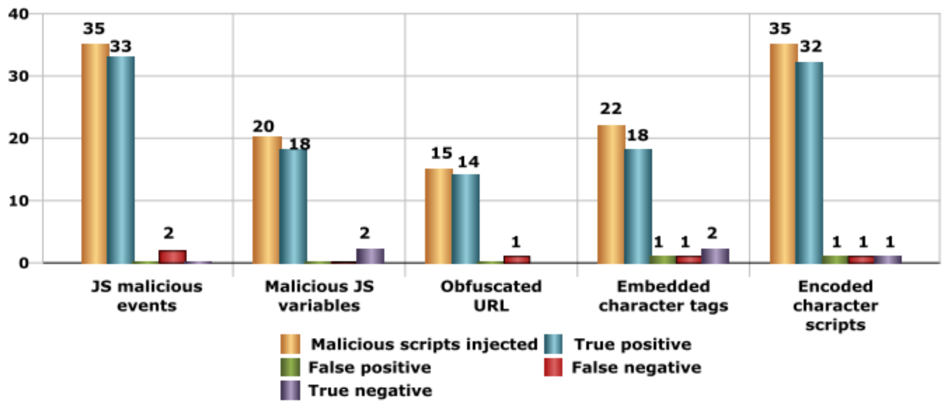

Fig. 8 *Observed experimental results on WordPress.*

(i.e. F-measure). F-measure generally analyses the performance of system by calculating the harmonic mean of precision and recall. The analysis conducted reveals that our framework exhibits high performance as the observed value of F-measure in all the platforms of web applications is 0.9. Therefore, the proposed framework exhibits 90% success rate in all the five HTML5 web applications. Tab. VI highlights the detailed performance analysis of our work.

$$\text{Precision} = \frac{\text{True Positive (TP)}}{\text{True positive (TP)} + \text{False Positive (FP)}}$$

$$\text{Recall} = \frac{\text{True Positive (TP)}}{\text{True positive (TP)} + \text{False Negative (FN)}}$$

$$\text{F-measure} = \frac{2(\text{TP})}{2(\text{TP}) + \text{FN+FP}}$$

Web Application	Total	# of TP	# of TN	# of FP	# of FN	Precision	Recall	F-measure
Drupal	127	117	3	6	1	0.9512	0.9915	0.9709
WordPress	127	115	5	2	5	0.9829	0.9583	0.9704

Tab. VI *Performance analysis by calculating F-measure.*

4. Conclusion and future work

This article discusses a client-server based XSS defensive methodology that scans and mitigates the effect of JavaScript vectors from the platforms of OSN. The proposed framework generates the views by performing segmentation on the generated HTTP response. These views define the accessibility region of user to check which view is accessible by user or not. Based on the extracted views, the extraction of JavaScript attack vectors will be performed by referring the blacklisted repository of such attack vector. Finally, the automated process of context-aware sanitization will be executed on these extracted attack vectors for their safe interpretation on the client-side web browser. We will try to integrate the capabilities of our proposed framework on the cloud platforms and will evaluate its XSS attack detection capability on some more platforms of OSN-based web applications.

References

[1] APACHE SOFTWARE FOUNDATION. Apache tomcat 5.5 [software]. 2012-10-10 [accessed 2016-01-30]. Available from: https://tomcat.apache.org/download-60.cgi

[2] BALZAROTTI D., COVA M., FELMETSGER V., JOVANOVIC N., KIRDA E., KRUEGEL C., SANER V.G.: Composing static and dynamic analysis to validate sanitization in web applications. *IEEE Symposium on Security and Privacy.* 2008, pp. 387–401, doi: 10.1109/SP.2009.33

[3] BATES D., BARTH A., JACKSON C. Regular expressions considered harmful in client-side XSS filters. In: *Proceedings of the 19th international conference on World wide web*, Raleigh, North Carolina, USA, ACM, 2012, pp. 91-—100.

[4] cao y., yegneswaran v., porras p.a., chen y. PathCutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks. In: *NDSS Symposium*, San Diengo, California, 2012. Available from: https://pdfs.semanticscholar.org/5ee0/0f29d07faaa19f6bb3a4c671e9fb26034d9d.pdf

[5] COX R.S., HANSEN J.G., GRIBBLE S.D., LEVY H.M. A safety-oriented platform for web applications. *In 2006 IEEE Symposium on Security and Privacy (S&P'06).* 2006, pp. 15, doi: 10.1109/SP.2006.4

[6] DRUPAL ORG. Drupal 7.23 [software]. 2013-04-03 [accessed 2016-01-30]. Available from: https://www.drupal.org/

[7] Facebook. [viewed 2014-01-14]. Available from: http://www.facebook.com/

[8] FAGHANI M.R., SAIDI H. Social networks' XSS worms. In: *International Conference on Computational Science and Engineering (CSE'09)*, IEEE, 2009, pp. 1137-—1141, doi: 10.1109/CSE.2009.424

[9] FIEGERMAN S. *Twitter now has more than 200 million monthly active Users* [online]. [viewed 2012-18-12]. Available from: https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/

[10] FIRE M., GOLDSCHMIDT R., ELOVICI Y. Online social networks: threats and solutions. *IEEE Communications Surveys & Tutorials*. 2014, 16(4), pp. 2019–2036, doi: 10.1109/COMST.2014.2321628

[11] Google+. [viewed 2014-01-14]. Available from: https://plus.google.com/

[12] GUNDOTRA V. *Google+: Communities and Photos* [online]. [viewed 2012-12-06]. Research report. Available from: http://googleblog.blogspot.coil/2012/12/google-communities-and-photos.html

[13] GUPTA B.B., GUPTA S., GANGWAR S., KUMAR M., MEENA P.K. Cross-site scripting (XSS) abuse and defense: exploitation on several testing bed environments and its defense. *Journal of Information Privacy and Security*. 2015, 11(2), pp. 118–136, doi: 10.1080/15536548.2015.1044865

[14] GUPTA S., GUPTA B.B. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*. 2015, pp. 1–19, doi: 10.1007/s13198-015-0376-0

[15] GUPTA S., GUPTA B.B. BDS: browser dependent XSS sanitizer. *Book on Cloud-Based Databases with Biometric Applications, IGI-Global's Advances in Information Security, Privacy, and Ethics (AISPE) series*, 2014, pp. 174–191, doi: 10.4018/978-1-4666-6559-0.ch008

[16] GUPTA S., GUPTA B.B. JS-SAN: defense mechanism for HTML5-based web applications against javascript code injection vulnerabilities. *Security and Communication Networks*. 2016, 9(11), pp. 1477–1495, doi: 10.1002/sec.1433

[17] GUPTA S., GUPTA B.B. PHP-sensor: a prototype method to discover workflow violation and XSS vulnerabilities in PHP web applications. In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*, Ischia, Italy, ACM, 2015, pp. 59.

[18] GUPTA S., GUPTA B.B. XSS-secure as a service for the platforms of online social network-based multimedia web applications in cloud. *Multimedia Tools and Applications*. 2016, pp. 1–33, doi: 10.1007/s11042-016-3735-1

[19] GUPTA S., GUPTA B.B. XSS-SAFE: a server-side approach to detect and mitigate cross-site scripting (XSS) attacks in JavaScript code. *Arabian Journal for Science and Engineering*. 2016, 41(3), pp. 897–920, doi: 10.1007/s13369-015-1891-7

[20] HANSEN R. Rsnake. XSS Cheat Sheet [online]. [viewed 2016-02-14]. Available from: http://ha.ckers.org/xss.html,2008

[21] HTMLUNIT PARSER [software]. [accessed 2016-01-30]. Available from: https://sourceforge.net/projects/htmlunit/files/htmlunit/

[22] Linkedin, about Linkedin, 2013. [viewed 2016-01-16]. Available from: https://press.linkedin.com/about-linkedin

[23] NADJI Y., SAXENA P., SONG D. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. *NDSS*. 2009, pp. 20.

[24] NEWSWIRE P.R. *Facebook reports fourth quarter and full year 2013 results.* [online]. MENLO PARK, Calif, 2014 [viewed 2014-01-29]. Research report. Available from: http://files.shareholder.com/downloads/AMDA-NJ5DZ/3073793635x0x721811/f028299e-a5b9-4ed5-9a2d-e3f0923ef261/FacebookReportsFourthQuarterAndFullYear2013Results.pdf

[25] OWASP top 10 vulnerabilities 2013. [online]. Available from: https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf

[26] PELIZZI R., SEKAR R. Protection, usability and improvements in reflected XSS filters. In: *ASIACCS*, Seoul, Korea, ACM, 2012, pp. 5.

[27] REIS C., GRIBBLE S.D. Isolating web programs in modern browser architectures. In: *Proceedings of the 4th ACM European conference on Computer systems (EuroSys '09)*, Nuremberg, Germany, ACM, 2009, pp. 219–232.

[28] ROXBERRY M., KOTOWICH K., STRANATHAN W., SHAH S., LARA J. *HTML5 Security Cheat Sheet.* [online]. [viewed 2016-02-14]. Available from: http://html5sec.org/

[29] TANG S., GRIER C., ACIICMEZ O., KING S.T.A.: a system for creating, enforcing, and testing browser security policies. In: *Proceedings of the 19th international conference on World wide web*, Raleigh, North Carolina, USA, ACM, 2010, pp. 241–250.

[30] Technical Attack Sheet for Cross Site Penetration Tests. [online]. [viewed 2016-02-14]. Available from: http://www.vulnerability-lab.com/resources/documents/531.txt

[31] TER LOUW M., Venkatakrishnan V.N. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. *30th IEEE Symposium on Security and Privacy.* 2009, pp. 331–346, doi: 10.1109/SP.2009.33

[32] WANG R., JIA X., LI Q., ZHANG S. Machine Learning Based Cross-Site Scripting Detection in Online Social Network. *High Performance Computing and Communications, 2014 IEEE 6th Int Symp on Cyberspace Safety and Security, 2014 IEEE 11th Int Conf on Embedded Software and Syst (HPCC, CSS, ICESS).* 2014, pp. 823–826, doi: 10.1109/HPCC.2014.137

[33] White Hat Security report. [online]. Available from: https://info.whitehatsec.com/rs/whitehatsecurity/images/2015-Stats-Report.pdf

[34] WORDPRESS ORG., WordPress 3.6.1 [software]. 2013-08-13 [accessed 2016-01-30]. Available from: http://wordpress.org/

[35] @XSS Vector Twitter Account. [online]. [viewed 2016-02-14]. Available from: https://twitter.com/XSSVector

[36] 523 XSS vectors [online]. [viewed 2016-02-14]. Available from: http://xss2.technomancie.net/vectors/