Hunting for DOM-Based XSS vulnerabilities in mobile cloud-based online social network

Article *in* Future Generation Computer Systems · June 2017

DOI: 10.1016/j.future.2017.05.038

CITATIONS
17

READS
1,476

3 authors, including:

Shashank Gupta
Birla Institute of Technology and Science Pilani
56 PUBLICATIONS **670** CITATIONS
SEE PROFILE

Pooja Chaudhary
25 PUBLICATIONS **103** CITATIONS
SEE PROFILE

Future Generation Computer Systems 79 (2018) 319–336

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Hunting for DOM-Based XSS vulnerabilities in mobile cloud-based online social network

Shashank Gupta, B.B. Gupta *, Pooja Chaudhary

Department of Computer Engineering, National Institute of Technology, Kurukshetra, Haryana, India

HIGHLIGHTS

- Define a DOM tree generation and context-sensitive sanitization based framework.
- Works offline to extract all the modules of the OSN-based web application.
- Detects the injection of malicious scripts in the DOM tree online.
- Performs a matching between white list and these extracted scripts.

ARTICLE INFO

Article history:
Received 17 September 2016
Received in revised form 26 February 2017
Accepted 28 May 2017
Available online 12 June 2017

Keywords:
Mobile cloud computing
Cross-site scripting (XSS) worms
JavaScript code injection attacks
DOM-Based XSS attacks
HTML5
Document Object Model (DOM) tree

ABSTRACT

This article presents a runtime Document Object Model (DOM) tree generator and nested context-aware sanitization based framework that alleviates the DOM-based XSS vulnerabilities from the mobile cloud-based OSN. The frameworks executes in dual mode: offline and online. The offline mode captures all the traces of modules of web applications and transformed such traces into static DOM tree for the extraction of benign script nodes. The legitimate script content embedded in such nodes will be marked in the whitelist of scripts. The online mode detects the injection of untrusted script content in the DOM tree generated at runtime. This was done by usually matching the script content embedded in this DOM tree with the whitelist of script code generated at offline mode. Any deviation observed in the script content will be marked as the injection of malicious script content in the dynamically generated DOM tree. This mode also identifies the different context of malicious variables embedded in such scripts and consequently executes the process of nested context-sensitive sanitization on them. The prototype of our mobile cloud-based framework was developed in Java and integrated the functionality of its components on iCanCloud simulator by creating different virtual machines with their proper link-to-link connectivity. The experimental testing and performance evaluation of our work was carried out on the open source OSN websites that are integrated in the virtual cloud servers. Evaluation results revealed that our framework is capable enough to detect the injection of untrusted/malicious script in the dynamically generated DOM tree with very low rate of false positives, false negatives and suffer from acceptable performance overhead.

1. Introduction

Due to proliferation of resource restraint mobile devices, pervasive wireless infrastructure, geographical-based services on cloud computing platform triggers a new paradigm called Mobile Cloud Computing (MCC). It is continuously gaining popularity among mobile users. MCC is a new upcoming technology that provides cloud based services to low processing power, low storage and less security devices.

Fig. 1 describes an overview of the architecture of MCC. In recent years, there is large number of applications targeted at mobile devices, including games, social networking, health, business, news, multimedia, etc. because MCC provide provides Internet-based services to user, regardless of geographical location. Currently, the most popular used application by mobile users is the social networking. The smart phone users access the facilities of Online Social Network (OSN) through the infrastructures of mobile cloud. Users retrieve multimedia data comprising text, audio, video, images, etc. on the platform of OSN. While multimedia provides smart services on OSN to its customers but it also attracts various types of cyber-attacks, namely, injection, XSS, spamming, phishing and so on. Now-a-days, to make mobile apps more interactive and dynamic, these are developed using advanced technologies (AJAX,

* Corresponding author.
E-mail addresses: shashank.csit@gmail.com (S. Gupta), gupta.brij@gmail.com (B.B. Gupta), pooja.ch04@gmail.com (P. Chaudhary).

http://dx.doi.org/10.1016/j.future.2017.05.038
0167-739X/

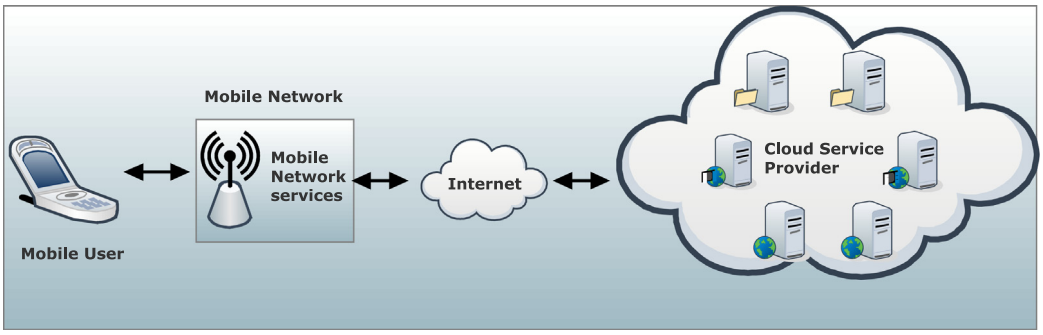

Fig. 1. Architecture of mobile cloud computing.

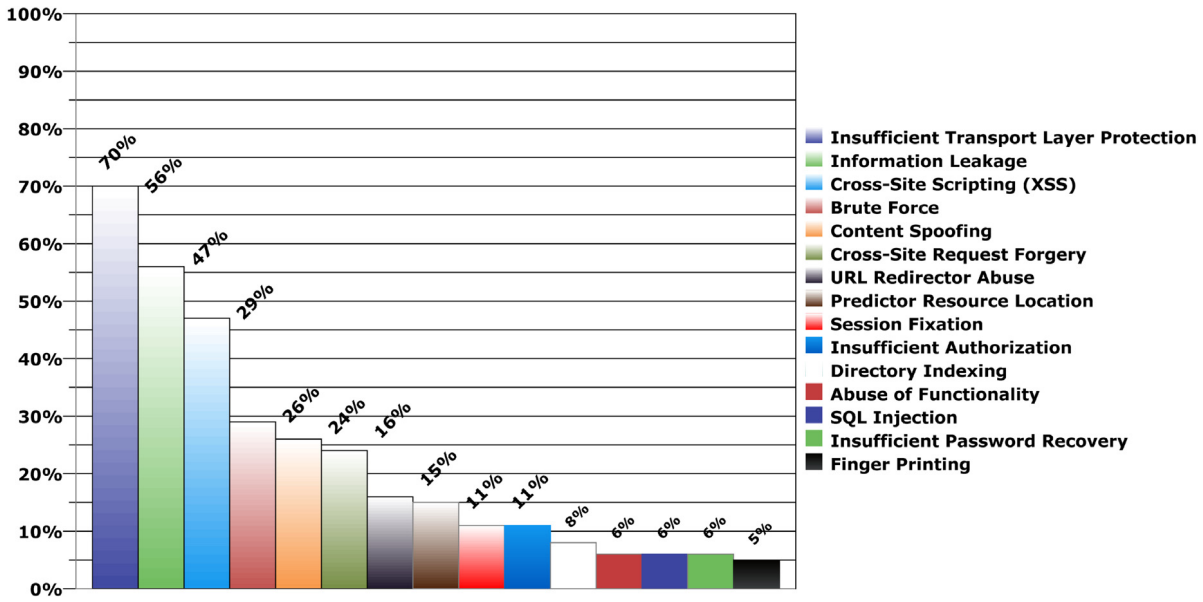

Fig. 2. White hat security statistics 2015 [2].

JavaScript, etc.). On one hand, it provide dynamic and processing intensive applications to mobile users, however, on the other hand, it has raised numerous issues related to the privacy and security of the smart phone users. This is generally due to the fact that the sensitive credentials of such users are kept in the domain of public cloud that is managed by untrustworthy commercial service providers. In the modern era of mobile cloud computing, mobile cloud security has turned out to be an utmost key challenge that has appealed numerous efforts related to research and development in the recent times. The most prominent attack found on OSN sites is the Cross Site Scripting (XSS) attack [1]. It is revealed by the report presented by the White Hat Security (2015) [2] that XSS occupies 47% among all other threats. Fig. 2 shows the statistics provided by the White Hat Security.

Fig. 3 illustrates the simple scenario of the exploitation of XSS vulnerability on the web browser of smart phone user. XSS worms have turned out to be a plague for the mobile cloud-based OSN accessing the multimedia data. Such worms steal the sensitive credentials of the active users by injecting the malicious JavaScript worms in the form of some posts on such web applications [3]. XSS attack falls under 3 categories: Reflected, Stored and DOM-based XSS attack. Numerous XSS defensive methodologies have been designed for detecting and mitigating the effect of reflected and stored XSS worms [3–6]. Input validation and malicious script sanitization are the most effective mechanism for alleviating and mitigating the effect of stored and reflected type XSS worms from the mobile cloud-based OSN platforms. Numerous defensive methodologies [1,7–16] had been proposed for thwarting the effect of XSS worms or XSS attacks from such platforms. However, till now, less attention is paid towards mitigating the impact of the DOM based XSS worm on mobile platforms. In this attack, the web application's client-side scripts write the user provided data to the DOM. Finally, this data is subsequently read by the OSN Web application and display results on browser. If the data is not validated, then an attacker may inculcate the malicious scripts which is stored as a part of DOM and gets executed when data is read from DOM and triggers adverse effects. Fig. 4 illustrates the pattern example of DOM-Based XSS attack.

Following are the key steps of exploiting the vulnerabilities of DOM-Based XSS attacks:

- Initially, attacker crafts a malicious URL incorporating a malicious JavaScript string and sends it to mobile user to lure victim.
- Then, victim is tricked to click on the URL and compels to make a request for the multimedia resource from OSN server on mobile cloud network.
- An error message is generated by the OSN server including the malicious script, as the requested resource is not stored into the OSN server. It is returned to the mobile user as HTTP response.

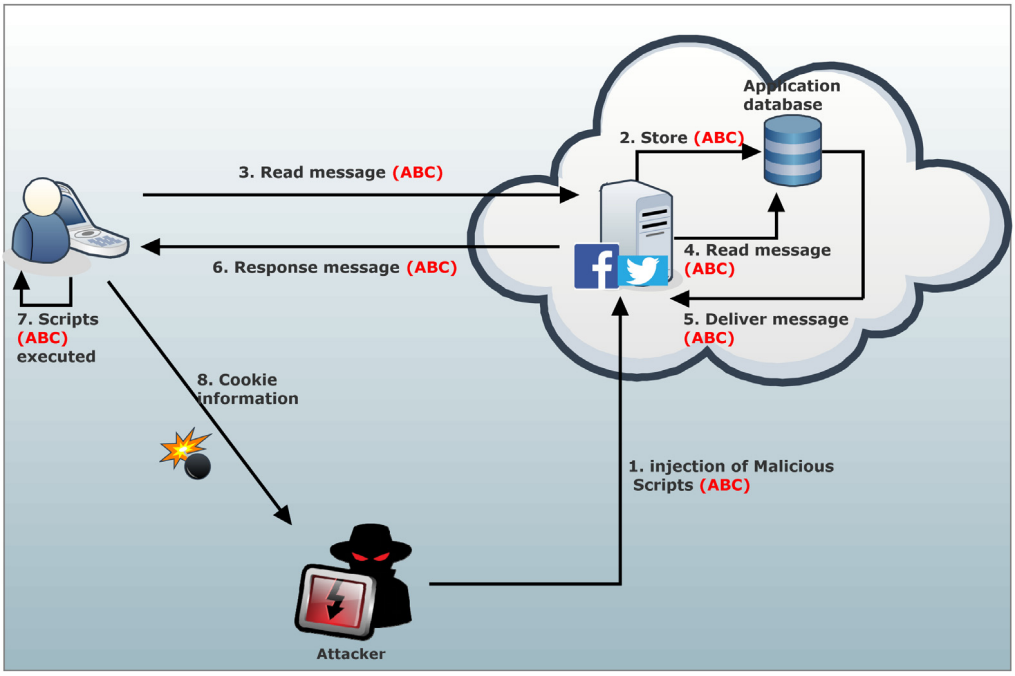

Fig. 3. XSS attack exploitation.

- At the client side, browser renders the HTTP response and executes the returned malicious script which triggers the modification in the DOM structure.
- When browser processes the DOM tree to display the result then script performs maliciously and sends sensitive information to the attacker's domain.

1.1. Existing work

Lekies [17] designed a technique that exploits a taint-aware JavaScript engine with DOM implementation and context-sensitive method for the generation of exploits. The technique provides complete analysis of all JavaScript features and DOM APIs by altering the JavaScript engine. Finally, attack vectors are injected depending on the context information to detect the XSS worm. Parameshwaran [18] evaluates and identifies DOM-based XSS susceptibilities in web applications. The technique rewrites JavaScript of the requested website to achieve character-precise taint tracing. It has two core modules: Instrumentation engine and Exploit generation. The former is responsible for character-precise taint tracing. It intercepts HTTP request, recognizes malicious JavaScript in the HTTP response. The later module analyzes the tainted flows, identified by the first module, and constructs context-based test payloads, which can be simply tested on the vulnerable web site. Bates et al. [4] broadly investigate the condition of the ability of three client side XSS filters (IE8 [19], noXSS [20] and NoScript [21]) and designed XSS Auditor, that is currently facilitated by default in Google Chrome. It scans the Document Object Model (DOM) tree generated by the HTML parser for the clear interpretation of the semantics of the HTTP response. In addition, it also provides the provision for discovering the untrusted external scripts that utilize the document.write functions.

Sun [6] proposed an entirely client-side solution, employed as a Firefox plug-in, which utilizes a string similarity methodology to identify the instances where downloaded scripts strictly look likes an outgoing HTTP requests. The key idea of this technique is to discover alike strings involving the category of parameter values in departing HTTP requests and extracted external files, and the category of DOM scripts. This technique discovers self-replicating XSS worms by calculating a string similarity distance between DOM nodes and HTTP requests executed at run-time by the client-side web browser. They capture every HTTP request on the browser side and match them with implanted scripts in the currently loaded web page. Cao [3] proposed a tool known as Path Cutter, which utilizes the dynamic analysis by jamming the transmission path of XSS worms by restricting the DOM access to several different views at the web browser and hampers the illicit HTTP web requests to the web server. Their technique partition the web application into several views. Path-Cutter separates these views on the client side by simply permitting those HTTP requests approaching from a view with the authorized privileges. Balzarotti [5] integrates static and dynamic analysis to model the sanitization method. Their analysis is utilized to detect the defective sanitization routines that can be evaded by a malicious attacker. Such routines are detected by recreating the source code used by the web application and then run this recreated code with the malicious input values in order to determine the defective sanitization routines.

1.2. Existing performance issues

In order to exploit the vulnerabilities of the two categories of XSS worms (i.e., stored and reflected [3,5,6]), attacker usually has to inject the malicious script code on the server-side. The existing literature comprises of numerous existing XSS defensive solutions for these two categories of XSS worms. However, very few solutions [1,16] had been proposed for detecting and mitigating the effect of client-side malicious script injection on the DOM tree. DOM-based XSS vulnerability is initiated due to the insecure data flows from attackers' domain sources such as innerHTML, document.location() attributes to the safe HTML APIs. As with the development of Web 2.0, most of the server-side functionality has been moved to the client side by incorporating JavaScript at the client side. Additionally, its capabilities are increasing due to the introduction of the new HTML5 APIs like web worker, history and properties including *<video>*, *<source>*, *<autofocus>*, etc. which can be utilized for creating new XSS attack vectors.

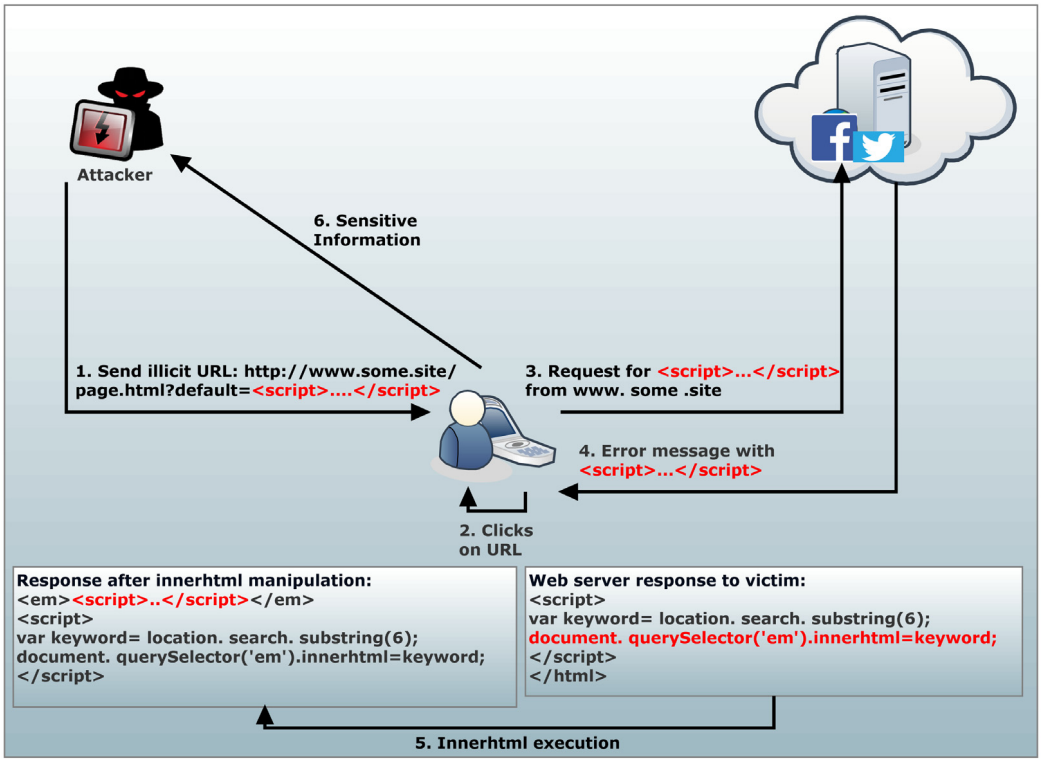

Fig. 4. DOM-XSS attack exploitation.

<video><source onerror = "<script>alert(document.cookie) </script>"></video>.

The modern web browsers or existing XSS filters does not check for this HTML5 attack vector. The parser on the web browser interprets the HTML document (embedded with scripts) and generates a DOM tree for the better and enhanced representation of the HTML document. The injected script code in this DOM tree will alter the content of the document and this alteration will give rise to DOM-based injection XSS attack on the client-side. The key advantage of adopting this feature is that it can be easily integrated among other platforms of web browsers. Furthermore, unlike other languages, most of the portion of JavaScript string is interpreted at runtime by using function like *eval()*. Moreover, most of the web application allows third-party JavaScript through *<script>* tag linked to cross-domain sources. Thus, this code is retrieved directly from third-party source and processed immediately. So, this code is not controlled by the application at client-side not even at the server-side. Therefore, a robust DOM-based XSS defensive solution is the need of the hour that will detect and introduce an effective mechanism of sanitizing/filtering the HTML5 XSS attack vectors. Although, no XSS defensive solutions has been designed for the platforms of mobile cloud-based OSN. Hence, the infrastructure settings of framework solution must be capable enough to integrate in the settings of mobile cloud-based environment and should evaluate the XSS worm recognition capability on mobile cloud-based OSN platforms.

1.3. Key contributions

Based on the above mentioned performance issues, this paper has the following contributions:

- Document Object Model (DOM) generator-based and nested context-aware sanitization-based technique is proposed for Online Social Network (OSN) against XSS vulnerabilities on mobile cloud-based platforms.
- Conventional context-aware sanitization is transformed into nested context-sensitive sanitization that reduces the runtime overhead caused due to the excessive sanitization of variables of JavaScript code.
- DOM tree generation of the offline and online HTTP response (by utilizing the capabilities of JSoup parser) enhances the detection of hidden scripts as well as the detection of injection of scripts from the remote servers.
- To the best of our knowledge, the proposed work is the first framework that evaluates the XSS worm detection capability on mobile cloud-based OSN by integrating the functionalities of proposed modules of our framework on the virtual machines of iCanCloud simulator.
- The accuracy and performance of our work in contrast to the existing client-side XSS filters is very high, since we have estimated the both of our framework (by using *F*-Measure, *F*-test, etc.) and is very high and acceptable.

The rest of this paper is organized as follows: Section 2 elaborates our proposed framework in a detailed manner. Section 3 provides the implementation details with the experimental results and provides a performance analysis of our work followed by comparison-based analysis. Lastly, Section 4 concludes our work.

2. Proposed design

This article presents a mobile cloud-based framework that detects and alleviates the propagation of DOM-based XSS vulnerabilities from the OSN. The novelty of our client-side framework lies in the fact that it generates the DOM tree of the HTTP response at runtime and obstructs the execution of untrusted injected JavaScript code. In addition, it identifies those script nodes in which such malicious injection occurs and recognizes the context of embedded variables in such malicious JavaScript strings. Such different contexts are determined for automatically executing the process of context-sensitive sanitization on such malicious variables. The

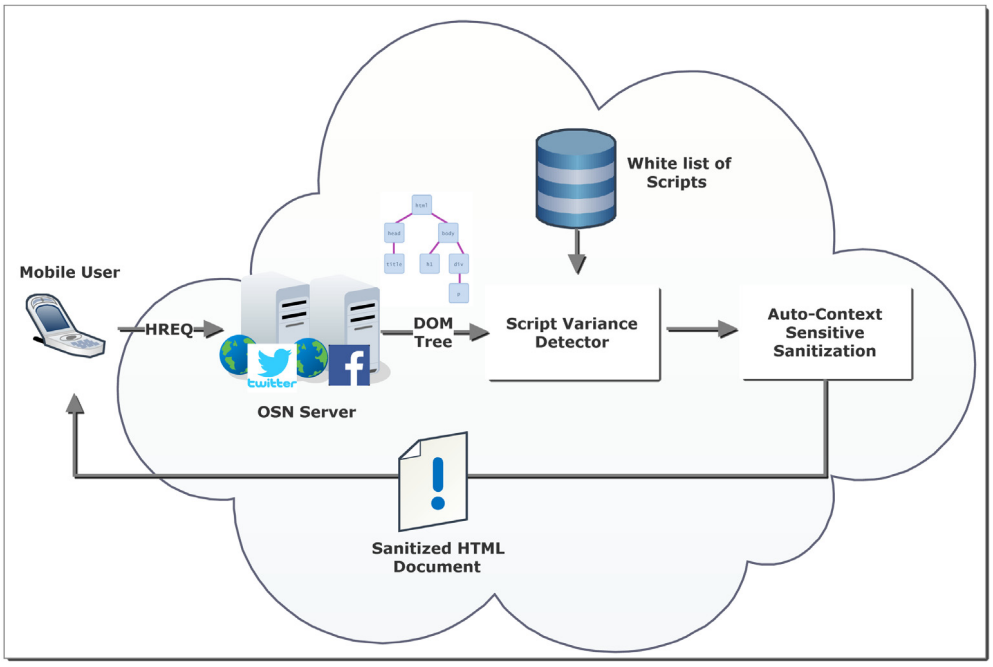

Fig. 5. Abstract view of the proposed framework.

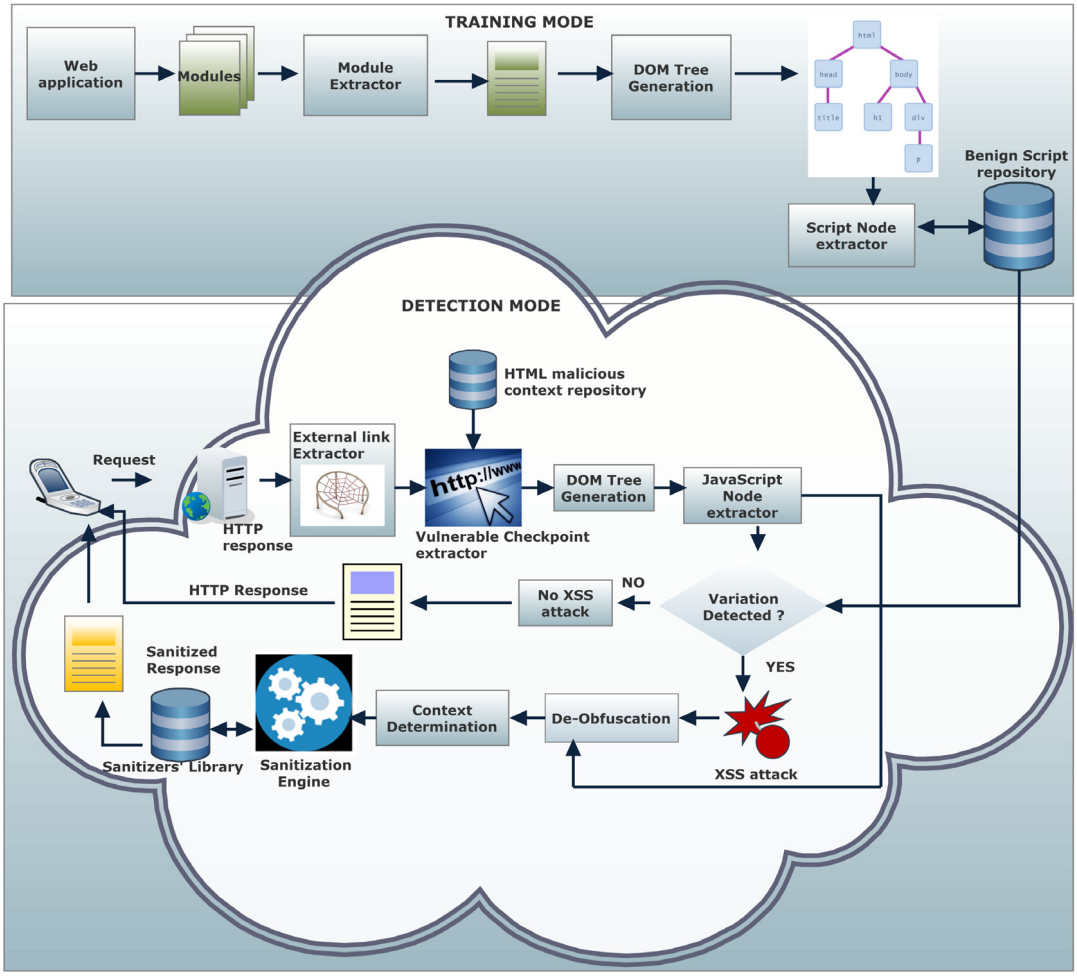

Fig. 6. Detailed view of proposed framework.

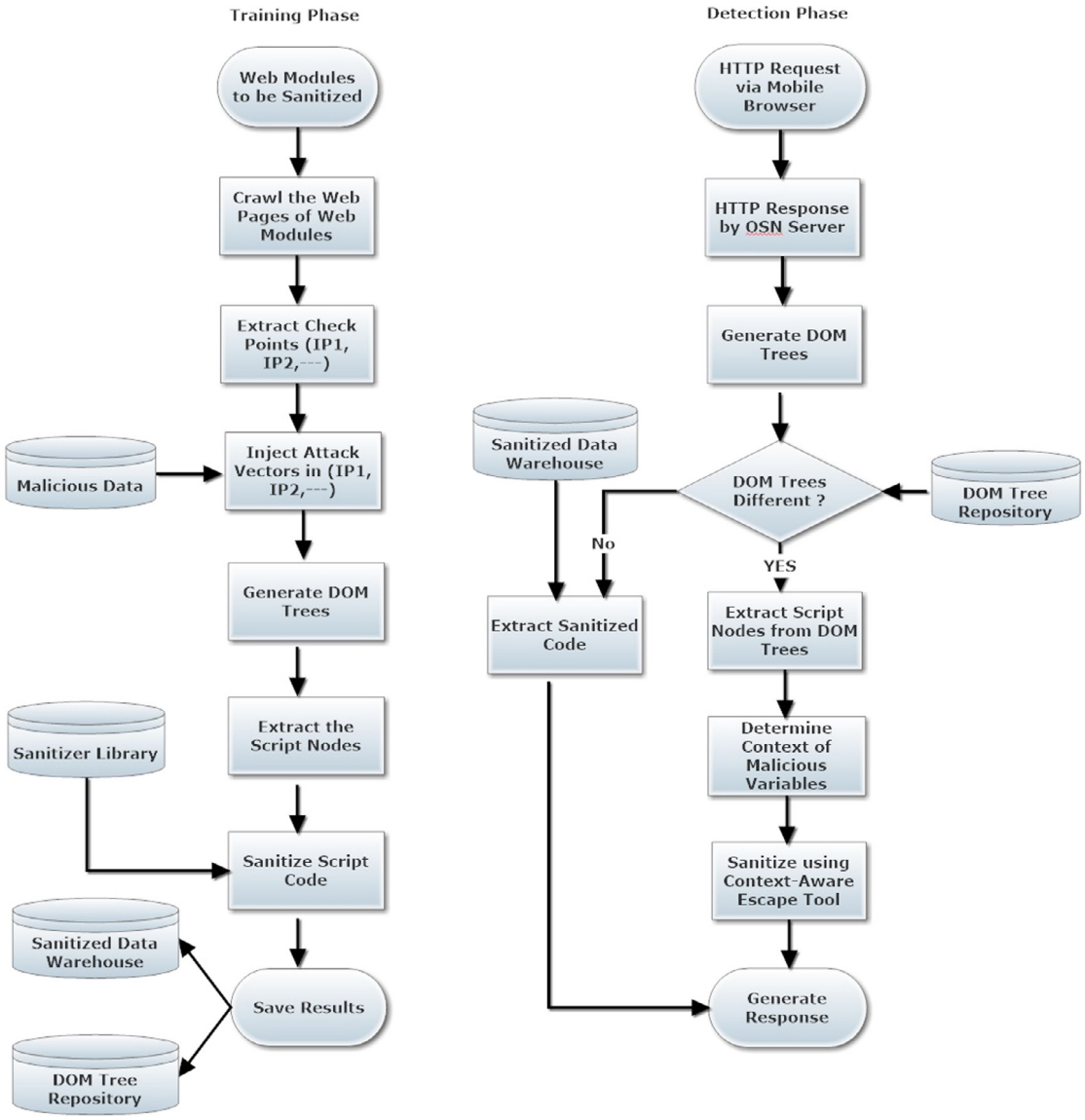

Fig. 7. Flowcharts of training and detection modes.

next subsection describes the brief overview of our proposed framework.

2.1. Abstract view

The proposed framework is a runtime DOM tree generator and context-aware sanitization based framework that scans for the DOM-based XSS vulnerabilities in the mobile cloud-based OSN. Fig. 5 highlights the abstract design view of our proposed mobile cloud-based framework. The OSN server deployed in the virtual machine of cloud platform initially extracts the HTTP request (HREQ) from the smartphone user. The OSN server generates the DOM tree at runtime corresponding to the generated HTTP response message. This tree is traversed and explored for the extraction of script nodes. The script variance detector will match the script content against the white list of scripts which is generated offline in the training mode by referring the existing modules of web applications. Any variation observed in the script content will simply indicate the injection of XSS worms in the DOM tree.

The injected script content, if any, will be transmitted to the sanitization engine that determines the context of untrusted variables present in the script string. Based on the determined context of such variables, this component will perform the context-sensitive sanitization by applying sanitization routines with the corresponding context on such untrusted variables and finally generates and transmits a sanitized HTML document to the smart phone user. Henceforth, we have described our proposed framework comprehensively. We have elaborated each phase and the key components involved in each phase.

2.2. Detailed illustration of framework

The mobile cloud-based framework executes in two modes: Training and Detection mode. Fig. 6 highlights the detailed design view of our mobile cloud-based framework. The training mode extracts all the benign JavaScript code by referring the script nodes of generated DOM tree and generates a white list of such script code offline. This whitelist will be further utilized for the detection of injected scripts in the DOM tree in the detection mode. The detection mode performs the context-sensitive sanitization on the injected obfuscated JavaScript code embedded in the DOM tree generated during runtime. The working flow of both the training

Algorithm: Training Mode

Input: Requested Web application

Output: Set of Extracted legal JavaScript code.

```
Start
JS_rep ⟵ NULL;
Mod_URL ⟵ NULL;
For each module MI ∈ web application
    UMI ⟵ ExtractURL ≙ MI;
    Mod_URL ⟵ UMI ∪ Mod_URL;
End For each
For each UMI ∈ Mod_URL
    WI ⟵ webpage ≙ UMI;
    DP ⟵ DOM-tree-generation(WI);
    For each node NI ∈ DP
        If (NI == <script>) then
            While (NI != </script>)
                JS_rep ⟵ NI;
        End if
    End For each
End For each
Return JS_rep
End
```

Fig. 8. Algorithm implemented for training mode.

and detection modes of our proposed framework is illustrated in Fig. 7. The following sub-section discusses the detailed illustration of both these modes.

2.2.1. Training mode

This mode extracts and scans all the modules of existing web applications by utilizing the web spider component. This mode extract all the script content embedded in such modules by generating the static DOM tree. All the legitimate script content is saved in the whitelist of scripts. Fig. 8 highlights the detailed illustration of training mode. The algorithm is implemented offline to perform all the operations for the completion of training mode. The working of the algorithm is as follows: JS_rep and Mod_URL are two logs maintained to store extracted JavaScript code and URL of each module of the Web application respectively. For each module $M_I \in$ Web application, we store the corresponding URL into the variable UM_I. Then, for each web page corresponds to URL UM_I (i.e. webpage $\triangleq U_{MI}$), it constructs the Document Object Model (DOM) tree as D_P. Then, it store all the content between the pair <script>…</script>in JS_rep. And it generates JS_rep as output. This log stores all the legitimate JavaScript code that may be present in the webpage and browser is allowed to execute at the client side.

2.2.1.1. Key components. This sub-section discusses the detailed illustration of components deployed in the training mode.

Module Extractor: It is defined as the sandboxed process used to run the part of the Web application. At the browser end, it will appear as the Web page or a part of it. For example, at an OSN site, comment post may be considered as a different module from the remaining Web page. It is responsible for the extraction of requested module and returns it to the browser as the HTTP response. It also extracts the module corresponding to each extracted HTTP request.

Script Node Extractor: The key goal of this module is to traverse the DOM tree and explore for the script nodes. The component scans each node (starting from the root) and searches for the opening script tags. The content embedded between such opening and closing tags (<script>…</script>) will be marked as a benign JavaScript code. Such code is saved in the benign script repository that will be further utilized for the detection of untrusted JavaScript code in the detection phase.

Algorithm: DOM Tree Generation

Input: Set of HTTP response

Output: Generated DOM tree of each requested web page

```
Start
S ⟵ -1;                    /* stack, initially empty*/
Root ⟵ NULL:               /* root node of the DOM tree*/
HTag_rep ⟵ NULL;
For Each HTTP Response HRI
    WI ⟵ web page ≙ HRI;
    TI ⟵ extract_tag(WI);
    S.push(TI);
    Root.addd_node(TI, NULL);
    HTag_rep ⟵ TI ∪ HTag_rep;
    While (S!= " ")
        Loc ⟵ extract_tag(WI);
        If ( opening_tag(Loc)) then
          X ⟵ S.size();
          Y ⟵ DFS(Root, X);
          Root.add_node(Loc, Y);
          S.push(Loc);
          HTag_rep ⟵ Loc ∪ HTag_rep;
        Else If ( closing_tag(Loc)) then
          S.pop();
        End If
    End While
End For Each
Return Root
End
```

Fig. 9. DOM tree generation algorithm.

DOM Tree Generation: Each extracted module of the web application is forwarded to this component. It is responsible for the construction of DOM tree for the received module by implementing the algorithm illustrated in Fig. 9.

The working process of the algorithm is described as follows: Initially, it extracts HTTP response web page as WI. Firstly, it discovers first HTML tag and stores it in TI and also pushes it into the stack S. Then, TI is added to the empty DOM tree as its root node. All the identified tags are stored into the repository HTag_rep as well as extended to the Stack. Until stack is empty, it extracts each HTML tag as Loc and check if it is an opening tag or closing tag. If opening tag then, it finds out the size if stack in X and traverse the partially generated DOM tree up to depth level X. When it reaches the last traversed node Y, then it adds the extracted HTML tag. Additionally, it add it to the stack and in to the HTag_rep. Otherwise, it pops out from the stack and returns the root node of the generated DOM tree for further processing to other component.

2.2.2. Detection mode

This mode detects the injection of XSS attack vectors in the DOM tree. The different context of malicious variables will be determined in this mode and accordingly performs the context-sensitive sanitization on such variables for alleviating the effect of DOM-based XSS vulnerabilities. Fig. 10 describes the algorithm for online mode. The algorithm shown in Fig. 10 runs online to detect DOM-based XSS attack. The working procedure of above algorithm is described as follows: JS_rep is the log containing set of all extracted legal JavaScript code produced by the training mode. S_rep is the repository maintained to hold JavaScript code present in the requested web page. For each generated HTTP response as HRI, we extract all external link embedded into HRI in Link_rep. And

Algorithm: Detection Mode

Input: HTTP request
Output: Sanitized HTTP Response

```
Start
JS_rep ⟵ Set of legal JavaScript code;
S_rep ⟵ NULL;
For each HTTP request H_I
    Generate HTTP response HR_I
    For each HR_I
        Link_rep ⟵ External link extractor(HR_I);
        CK_PT ⟵ Vulnerable Checkpoint Extractor(Link_rep);
        D_P ⟵ DOM-tree-generation(HR_I);
        For each node N_I ∈ D_P
            If (N_I == <script>) then
                While (N_I != </script>)
                    S_rep ⟵ NI;
            End if
        End For each
        Z ⟵ Compare(S_rep, JS_rep);
        If (Z== TRUE) then
            No XSS;
            Return HR_I
        Else
            S_I' ⟵ Decode(S_I ∈ S_rep);
            Con_rep ⟵ Context determination(S_I');
            HR_I' ⟵ Sanitization engine(Con_rep);
            Return HR_I'
        End If
    End For each
End For each
End
```

Fig. 10. Algorithm implemented for detection mode.

then, discover all vulnerable checkpoints present in the web page in CK_PT. Additionally, it generates DOM tree as DP. It retrieves all content present between the pair <script>…</script> into S_rep. To detect the XSS attack, it compares both the extracted JavaScript code present in JS_rep and S_rep. If there is a no variance, then no XSS attack. Otherwise, malicious script is decoded as S_I' and its context is determined to correctly identify the sanitizer's function. Finally, the identified sanitizer is applied on the untrusted variable present in the malicious JavaScript. Lastly, it produces sanitized HTTP response for user free from the malicious content causing XSS attack.

2.2.2.1. Key components. This sub-section discusses the detailed illustration of components deployed in the detection mode.

External Link Extractor: This module is responsible for the extraction of external JavaScript links embedded in the Uniform Resource Identifier (URI) links. Initially, each HTTP response will be checked for embedded parameter values. Such extracted parameter values will be explored for the embedded URI links.

The existence of such URI links indicates the remote location of external JavaScript links. It can be usually extracted by transmitting Asynchronous JavaScript XML (AJAX) HTTP request and responses from the remote location of servers. Fig. 11 highlights the detailed algorithm of external JavaScript link extractor.

This algorithm restricts the level of crawling of different web pages of the web application, by using a randomly generated threshold (β). It maintains Link_rep repository to store all crawled URLs. Initially, it retrieves the web page W_I corresponding to the requested URL HRI. Then, it extracts out all the external links embedded in the received web page. Finally, it stores the extracted URLs in the Link_rep and produces Link_Rep as its output.

Vulnerable Check Point Extractor: This component is responsible for the identification of injection points in the response web page

Algorithm: External link extractor

Input: HTTP response
Output: Set of crawled URLs.

```
threshold (β):- 0
Start
Link_rep ⟵ NULL;
β ⟵ randomGenerator();
For I ⟵ 1 to β
    For each HTTP response HR_I
        W_I ⟵ webpage ≙ HR_I
        U_IN ⟵ Crawler(W_I);          /* traverse internal links*/
        Link_rep ⟵ U_IN ∪ Link_rep;
    End For each
End For
Return Link_rep
End
```

Fig. 11. Algorithm for remote JavaScript link extractor.

Algorithm: Vulnerable Checkpoint Extractor

Input: List of crawled URLs (U_1, U_2, …, U_N)
Output: Set of Extracted Vulnerable Checkpoint.

```
Start
Link_rep ⟵ Set of crawled URLs;
P_rep ⟵ NULL;
HTML_Malcon ⟵ List of HTML malicious context;
CK_PT ⟵ NULL;
For each U_I ∈ Link_rep
    P_I ⟵ getparametervalue(U_I);
    P_rep ⟵ P_I ∪ P_rep;
End For each
For each P_I ∈ P_rep
    If (P_I.matches(URL)) then
        U_I ⟵ extracted URL;
        Send AJAX request to server;
        W_S ⟵ Extracted page;
        C_K ⟵ explore-malicious-context(W_S) ⊆ HTML_Malcon;
        CK_PT ⟵ C_K ∪ CK_PT;
    End if
End For each
Return CK_PT
End
```

Fig. 12. Algorithm for vulnerable checkpoint extractor.

where an attacker may inject malicious code to launch XSS attack. This is achieved by monitoring all malicious contexts in the HTML document with the help of HTML malicious context directory. Fig. 12 describes the algorithm implemented for accomplishing the functionality of this component. The algorithm works in three key steps: Initially, the input to the algorithm is the set of crawled URLs produced by an external link extractor algorithm, as Link_rep. P_rep and CK_PT are maintained to hold parameter values extracted from the crawled URLs and vulnerable injection points present in the web page respectively.

HTML_Malcon is the externally available repository of all HTML malicious contexts which may be present in the web page. Secondly, for each URL ∈ Link_rep as U_I, we store the parameter value in P_I using getParametervalue() function and store it in the P_rep. Then, for each $P_I \in$ P_rep, we extract the URL link in U_I by matching the parameter value with URL syntax. Lastly, we send AJAX request to remote server to receive the requested web page and discover all the vulnerable checkpoints present in the received web page with the help of HTML_Malcon repository. Finally, it stores all the

Table 1
List of escape codes.

Display	Hexadecimal code	Numerical code
“	"	"
#	#	#
&	&	&
‘	'	'
(	(	(
)	)	)
/	/	/
;	;	;
<	<	<
>	>	>

identified injection points into the CK_PT repository and generates it as its output.

DOM Tree Generation: This component works similar as in training mode. It receives the HTTP response web page with the identified vulnerable check points and it generates the DOM tree of the related web page by using the algorithm illustrated in Fig. 9.

JavaScript Node Extractor: This extraction component works in the similar way as script node extractor. However, the script node extractor component scans for the benign JavaScript code and extracts all the transmitted JavaScript code. This component scans for the <script> tag in DOM tree and stores all data after that until if finds </script> tag in the DOM tree.

De-obfuscation: This component performs the decoding of the obfuscated JavaScript code. In order to bypass the XSS filters applied at server and/or client side, attackers, generally, performs the obfuscation on the simple malicious JavaScript code. Therefore, it is necessary to de-obfuscate the injected JavaScript code to clearly identify the malicious elements present in it. Table 1 highlights some of the list of escape codes [22]. Suppose malicious attacker inserts <script>alert (“XSS”) </script> inside the variable area of victim’s web application, then the special character like ‘<’, ‘>’ would be replaced with &#60 and &#62. Now, the web browser will show this script of a portion of a web page, however the web browser could not run the script.

Nested Context Determination: The goal of this component is to identify different context of each type of untrusted variable included in the JavaScript string entered as untrusted input by user. It will reveal the context in which each untrusted variable in JavaScript input will be safely rendered and sanitizers’ functions are selected accordingly. Fig. 13 illustrates the algorithm processed to determine the different possible contexts of JavaScript code. Here, input to the above algorithm is the set of the extracted JavaScript code as JS_rep. Con_rep is a log maintained to store context of each untrusted variable present in the JavaScript code. For each untrusted JavaScript variable X_I, we attach a context identifier CI in the form as $CR_I \leftarrow (CI)X_I$. The generated output is the internal representation of the extracted JavaScript code embedded with the context identifier CI corresponds to each untrusted variable present in it. After this, it is merged with the Con_rep as $Con\_rep \leftarrow CR_I \cup Con\_rep$. For each $CR_I \in$ Con_rep, we generate and solve the type constraints. Here, Γ represents the type environment that performs the mapping of the JavaScript variable to the Context identifier CI. In the path sensitive system, variable’s context changes from one point to other point. Thus, to handle this issue, untrusted variables are represented through the typing judgments as $\Gamma \mapsto$ e: CI. It indicates that at any program location, e has context identifier CI in the type environment Γ. Finally, all CR_I variables have been assigned the context dynamically (string, regular expression, numeric etc.) and produce the modified log Con_rep as output. This step outputs untrusted variable, present in the extracted JavaScript code, with their context in which browser interprets it.

Algorithm: Nested Context Determination

Input: Set of decoded JavaScript code
Output: Context of each untrusted variable in JavaScript.

Start
Context identifier: $CI_1|\ CI_2|\ldots|CI_N$;
JS_rep $\leftarrow$ List containing extracted JavaScript code.
Con_rep $\leftarrow$ NULL; /* list for type qualifier variables*/
For Each extracted JavaScript S_I
$X_I \leftarrow$ untrusted variable $\widehat{=} S_I$;
$CR_I \leftarrow CI(X_I)$;
Con_rep $\leftarrow CR_I \cup$ Con_rep ;
End For Each
For Each $CR_I \in$ Con_rep
Parse(S_I);
If ($X_I \in$ String) **then**
$\Gamma \mapsto CR_I$: String;
Else if ($X_I \in$ Numeric) **then**
$\Gamma \mapsto CR_I$: Numeric;
Else if ($X_I \in$ Regular expression) **then**
$\Gamma \mapsto CR_I$: Regular expression;
Else if ($X_I \in$ Literal) **then**
$\Gamma \mapsto CR_I$: Literal;
Else if ($X_I \in$ Constant) **then**
$\Gamma \mapsto CR_I$: Constant;
End If
End For Each
Con_rep $\leftarrow$ newvalue(CR_I)
Return Con_rep
End

Fig. 13. Algorithm for nested context determination of malicious variables.

Sanitization Engine: This component is responsible for validating the untrusted user input to ensure that they are in correct format as perceived by the Web application. Auto Context-sensitive sanitization applies sanitizer on each untrusted variable in an automated manner (i.e. dynamic content like JavaScript) according to the context in which they are used.

There may be different contexts present in an HTML document like element tag, attribute value, style sheet, script, anchors, href, etc. These all contexts may be used by the attacker to launch the XSS attack. Fig. 14 highlights the algorithm for executing the process of sanitization on the injected untrusted JavaScript code in the DOM. The above algorithm takes Con_rep as its input which stores identified context of all untrusted JavaScript variable. XSS-San_lib is the externally available sanitizers’ library which stores sanitizers’ functions corresponding to each malicious context. For each extracted Jav bvaScript S_I, it extracts the untrusted variable X_I from the Con_rep and store the corresponding context identifier CR_I attached to the X_I in C_I. Then, it identifies for the corresponding sanitizer’s function, from the XSSSan_lib, with matching context and store it in SZ_I. It applies the identified sanitizer’s function on the untrusted variable and store result into the Y_I. It then merges Y_I with XSSSan_lib as $XSSSan\_lib \leftarrow Y_I \cup XSSSan\_lib$. Finally, it embeds all sanitized variable into the HTTP response and produce HTTP response for the user.

2.3. Security analysis

In addition to this, we have referred the list of taint sources and sinks for recognizing the sources of untrusted data. We have

Algorithm: Sanitization engine

Input: List of context identified JavaScript (S_1, S_2, S_3… S_N).

Output: Sanitized HTTP response.

Start

XSSSan_lib ⟵ Externally available sanitizers' libraray (S_1, S_2, S_3… S_N);

Con_rep ⟵ List of context identified variables;

For Each extracted node S_I

X_I ⟵ Con_rep – (Untrusted variable ≙ S_I);

C_I ⟵ CR_I;

SZ_I ⟵ ($SZ_I \in$ XSSSan_lib) ∩ (SZ_I matches C_I);

Y_I ⟵ $SZ_I(X_I)$;

XSSSan_lib ⟵ $Y_I \cup$ XSSSan_lib;

Embed all sanitized variables in response;

End For Each

Return HTTP response

End

Fig. 14. Algorithm for sanitization of untrusted JavaScript code.

Table 2
Sources of untrusted data.

Dangerous flow sinks	Resultant attack exploitation
document.forms[0].action, document.body.* document.writeln(…), window.attachEvent(), document.write(…), document.create() document.attachEvent() document.body.innerHTML document.execCommand() document.forms[*].action,	HTML Code Insertion
XMLHttpRequest.open(,url,), document.forms[*].action,	Parameter Insertion
window.setInterval(), eval(), window.setTimeout()	JavaScript Insertion
document.cookie	Session Hijacking

also explored for the critical sinks as well as equivalent exploits that may occur if data is utilized deprived of any sort of context-sensitive sanitization. Table 2 highlights the sources of untrusted data, possible dangerous flow sinks and their resultant attack exploitation.

While extracting the set of scripts from the DOM tree in the online and offline mode, we have carefully access the DOM properties of the JavaScript document. The objects retrieved by the DOM tree could be any attribute or keyword that can also result in stealing of cookie file and history related to that web page, that can be further useful for tracking the surfing behavior of cloud user. In order to avoid this, we have maintained a blacklist of DOM properties that are used for the exploitation of XSS worms. Table 3 highlights the list of such DOM properties. The script HTML element could encompass dangerous code as well as the link and object tags that could automatically copy and run dangerous code. HTML attributes like object, frameset, and body elements could include event handlers that again could run dangerous code in an automated manner. The offline mode has identified the list of these event attributes after the successful detection of XSS worms and finally sanitizes them in a context-sensitive manner. Table 4 highlights the blacklist of malicious event attributes utilized for the exploitation of XSS worms. These can be utilized to run malicious script files. Hence, our Cloud-Sensor recognizes such HTML attributes as well as elements as possibly malicious.

2.4. Key strengths

The main advantage of our framework is that it provides defend against DOM-based XSS attack as less attention is paid towards this XSS vulnerability by the existing system. It does not require any kind of modifications at the server and client side as it is integrated as Google Chrome extension. Its novelty lies in the fact that it does not require to trace the code dynamically, rather, it generates the DOM tree dynamically and extracts the scripts node to ensure that whether these scripts are allowed by the web application in its web page or not. This guarantee that no malicious scripts injected by the attacker get executed by the browser as it is not included in the white list of the scripts generated offline in training phase. Moreover, it does not perform sanitization of all scripts included in the web page, instead, it only sanitized those scripts which shows deviation from the legitimate scripts allowed by the web application. So, it also perform sanitization in an optimized manner and by applying only sanitization functions whose context matches with the context of the untrusted variable in the malicious script. The next section will elaborate the implementation details of the framework with experimental results as Chrome extension along with performance analysis of our work.

3. Implementation and experimental evaluation

This section discusses the implementation and experimental evaluation outcomes of our mobile cloud-based framework.

3.1. Implementation

We have implemented our framework as an extension of Google Chrome for mitigating the effect of XSS vulnerabilities from OSN-based web applications. We have developed a prototype of our framework in Java via introducing the Java Development Kit and integrated its settings on the virtual machines of iCanCloud simulator. The iCanCloud simulator is deployed at a 1.63 GHz Dual Core Processor with 4 GB RAM running Lollipop android OS versions and Google Chrome. Initially, we manually verified the performance of our proposed work against five open source available XSS attack repositories [23–27], which includes the list of old and new XSS attack vectors. Very few XSS attack vectors were able to bypass our client-side design.

3.2. Experimental evaluation

This section discusses the evaluation outcomes of our proposed framework on different platforms of OSN-based web applications and evaluates the performance and accuracy of our system. We have tested our proposed system on five open source real world OSN platforms i.e. Elgg [28], WordPress [29], Humhub [30], Joomla [31] and Drupal [32]. 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 percentage of XSS attack payloads are alleviated by our system. In terms of performance, we evaluate the performance-related issues of executing the proposed framework on a variety of web page-loading and HTML malicious context standards. To provide a deep insight on how to exploit XSS vulnerability present in the web application, we have illustrated the process through Fig. 15. We have integrated the snapshots of the web application to provide clear understanding.

This entire process works as follows:

- Initially, attacker impersonate as valid user by providing valid user's login credentials.
- Attacker tries to perform maliciously by finding out the vulnerable points like comment box and injects some malicious string.
- Web application processes it as comment from the valid user and stores it in database without validation.

Table 3
Malicious DOM property.

DOM property	Attack description
document.body.innerHTML document.write document.writeln	These properties can be utilized to modify the content of web page.
location.reload location.href window.location.reload window.location document.location	Such properties can be utilized to change the position of document.
document.getElementsByName document.getElementById document.getElementsByTagName	These can be utilized to adjust the values of attributes and tags in the web page.
document.cookie	This property is utilized to retrieve the values of cookies of a particular session.
History.forward Window.history History.go History.back	Such properties can be accessed for traversing and accessing the history of web browser tabs.

Table 4
List of blacklisted event attributes.

HTML entity	Event attributes
Keyboard	onkeyup, onkeydown
Media	onerror, onplayong, oncanplay, onemptied, onreadystatechange, oncanplaythrough, onprogress
Window	onmessage, onunload, onbefore, onhaschange, onpagehide, onafterprint, onrisize
Form	onfocus, oninput, onsubmit, onblur, onformchange, onforminput, oninvalid, onselect

Table 5
Configuration of different platforms of OSN-based web applications.

Application	Version	XSS vulnerability	Source language	Lines of code
Drupal [32]	7.23	CVE-2012-0826	PHP	43 835
Humhub [30]	0.10.0	CVE-2014-9528	PHP and jquery	129 512
Elgg [28]	1.8.16	CVE-2012-6561	PHP	114 735
WordPress [29]	3.6.1	CVE-2013-5738	PHP	135 540
Joomla [31]	3.2.0	CVE-2013-5738	PHP	227 351

- Henceforth, a legitimate user login into the system and starts reading the comments posted by other user.
- As victim reads the malicious comment posted by attacker, the injected malicious script get executed by the browser and victim's login credentials are send back to the attacker.

Fig. 16 illustrates the process of exploiting the XSS vulnerability on real OSN based web application i.e. Humhub. It shows what are the steps involved to initiate an XSS attack by the attacker. We have incorporated the infrastructure of our proposed framework into the OSN application. Table 5 highlights the configuration and XSS known vulnerability on the mentioned OSN platforms. In order to include the capabilities of our framework in these web applications, developers of these websites have to do less quantity of effort. The motivation to select these web applications is that, we simply need to mark an argument that web sites can utilize the capabilities of proposed framework, irrespective of the input testing. This will aid in alleviating XSS malicious code injection vulnerability concerns and would include supplementary security layer.

The simplicity with which we are capable to combine our framework in these popular OSN web applications determines the flexible compatibility of our proposed framework. We deploy these OSN web applications on an XAMPP web server with MySQL server as the backend database. We have tested and evaluated the detection capability of our proposed solution on five real world OSN web applications namely Elgg, WordPress, Humhub, Joomla, and Drupal. We select such applications for accessing the forms to potentially supply modified pages and access the HTML forms. Table 6 highlights the five different categories of the malicious HTML context, including HTML malicious tags, JavaScript attack vectors; CSS attack vectors, URL attack vectors and HTML malicious event-handler. These attack vectors also include the HTML5 attack vectors also. The observed results of our framework on five real world OSN web applications corresponding to the chosen categories of XSS attack vectors has been shown in the Fig. 17.

We have injected the five different contexts of XSS worms on the injection points of these web applications. We have analyzed the experimental results of our mobile cloud-based framework based on five parameters (# of malicious script injected, # of True Positives (TP), # of False Positives (FP), # of True Negatives (TN) and # of False Negatives (FN)). It can be clearly observed from the Fig. 17 that the highest numbers of TPs are observed in Drupal, Elgg and Joomla. In addition, the observed rate of false positives and false negatives is acceptable in all the five platforms of web applications. We have also calculated the XSS attack payload detection rate of our framework on all five platforms of OSN-based web applications. This is done by dividing the number (#) of TPs to the number of malicious script injected for each category of context of attack vectors.

Fig. 18 highlights the XSS worm detection rate of our framework on all the five platforms of OSN-based web applications w.r.t. individual category of XSS attack vectors. It is clearly reflected from the Fig. 18 that the highest overall DOM-based XSS worm detection

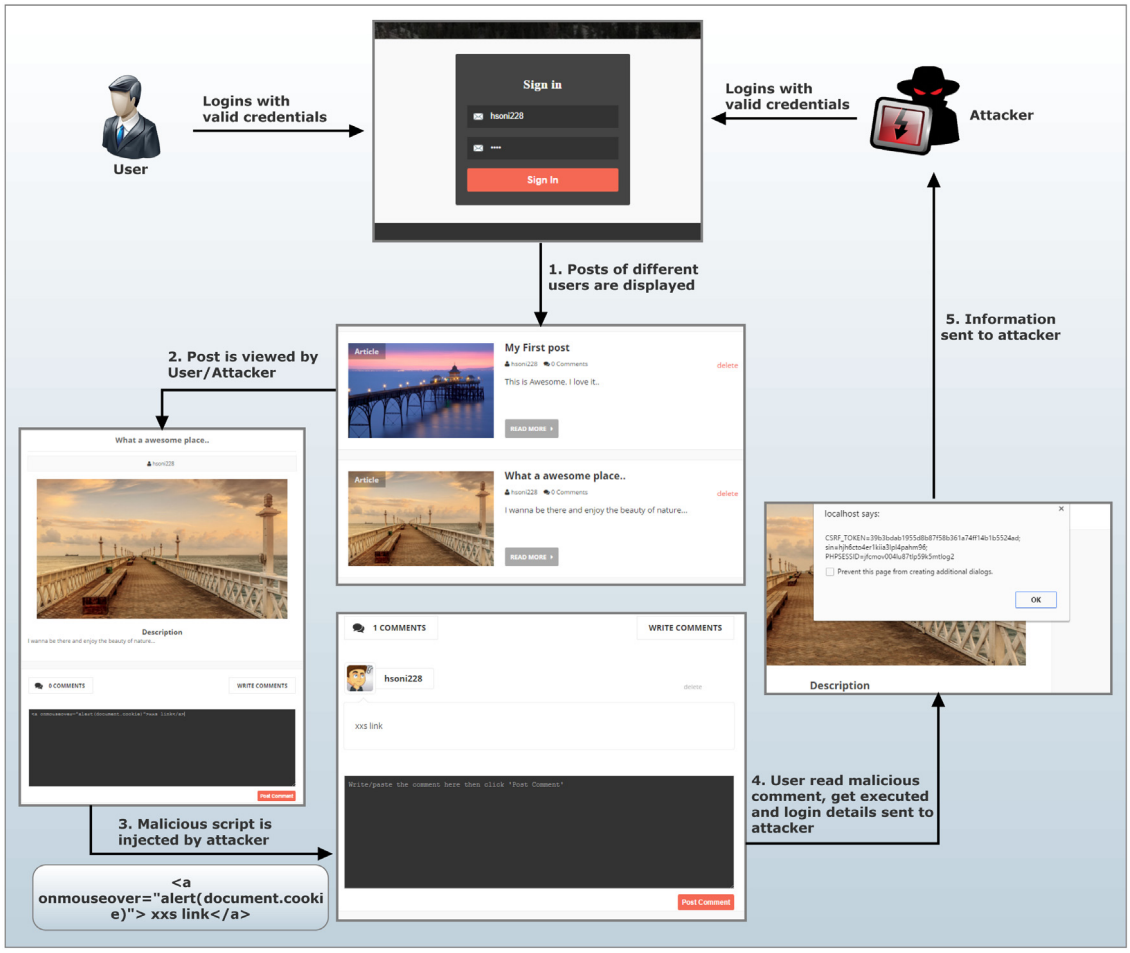

Fig. 15. Exploitation of XSS vulnerability.

rate is observed in Drupal, Elgg and Joomla. In addition, the overall XSS worm detection rate falls in the range of 95%–98% for all the platforms of HTML5web applications. Fig. 19 illustrates how we have integrated our proposed framework as Google chrome extension. It shows the steps of how it is defending against DOM-based XSS attack. Following steps are executed by our framework to mitigate the effect of malicious scripts injected by the attacker.

- Initially, user login into the system and posted a comment.
- When user click on the post button, then the data is forwarded to the Chrome Extension where we have integrated our framework for filtering the malicious string.
- Then, this data is sanitized by the framework and result is stored in the extension.
- Sanitized data is returned back to the input field in the application and get posted.
- Therefore, sanitized comment is posted on the web application, preventing the execution of malicious scripts.

Figs. 20 and 21 illustrates the integration of our proposed framework for mitigating the effect of the XSS on real-world OSN platform i.e. Humhub and Drupal, respectively. It shows the steps involved in the mitigation procedure.

3.3. Performance analysis

This sub-section discusses a detailed validation and performance analysis of our mobile cloud-based framework by conducting two statistical analysis methods (i.e., *F*-Measure and *F*-test Hypothesis). We have also compared the DOM-Based XSS detection capability of our proposed framework with other recent related XSS defensive methodologies based on some useful metrics. The analysis conducted reveal that our framework produces better results as compared to existing state-of-art techniques.

3.3.1. Performance analysis using F-Measure

For the binary classification, precision and recall are the values used for evaluations. And *F*-Measure is a harmonic mean of precision and recall.

$$FalsePositiveRate\,(FPR) = \frac{False\ Positives\,(FP)}{False\ Positives\,(FP) + True\ Negatives\,(TN)}$$

$$FalseNegativeRate\,(FNR) = \frac{False\ Negatives\,(FN)}{False\ Negatives\,(FN) + True\ Positives\,(TP)}$$

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

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

$$F - Measure = \frac{2\,(TP)}{2\,(TP) + FP + FN}.$$

Here, we calculate the precision, recall and finally *F*-Measure of observed experimental results of our framework on five different platforms of web applications. *F*-Measure generally analyzes the performance of system by calculating the harmonic mean of precision and recall. The analysis conducted reveals that our mobile

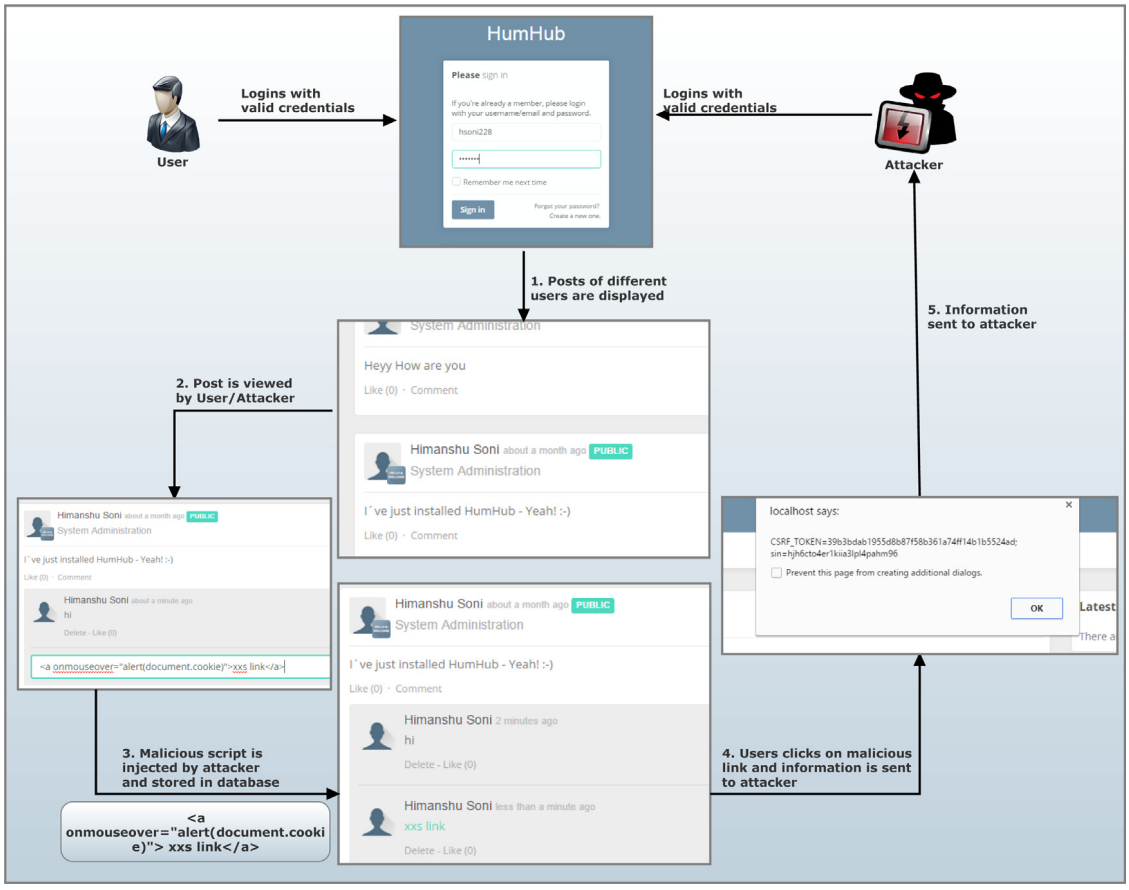

Fig. 16. Exploitation of XSS vulnerability on Humhub.

Table 6
Category of XSS attack vectors and their example patterns.

XSS attack vector category	Description	Example pattern
External source script vectors	Such scripts are loaded when page is loaded by browser and all the malicious content is inserted in webpage.	<SCRIPT SRC=http://ha.ckers.org/xss.js></SCRIPT> < *SCRIPT*/*XSSSRC* =”http://ha.ckers.org/xss.js”></SCRIPT> <SCRIPT SRC=http://ha.ckers.org/xss.js?<B >
HTML malicious attributes	Such scripts are executed when the attributes of tags in scripts are loaded by browser.	<BODY BACKGROUND=”javascript:alert(‘XSS’)”> <IMG SRC=“javascript:alert(’XSS’);”> <IMG SRC=javascript:alert(String.fromCharCode(88,83,83))> <IMG SRC=/ onerror=“alert(String.fromCharCode(88,83,83))”></img>
Event triggered scripts	Such code is executed when events like mouse click is performed on attack vector.	xxs link <IMG SRC=# onmouseover=“alert(’xxs’)”> <IFRAME SRC=# onmouseover=“alert(document.cookie)”></IFRAME>
Encoded attack vectors	JavaScript code encoded by the attacker to bypass it from filters.	<IMG SRC=javascri& #112;t:alert('XSS')>
Explicate Attack vectors	Such code is injected in the injection points of Web application in its clear form	<BR SIZE=“&{alert(’XSS’)}”> <BASE HREF=“javascript:alert(’XSS’);//”>
Unclear attack vectors	This type of attack vectors are not in clear form so can be bypassed by filters.	<aonmouseover=”alert(document.cookie)”>xxs link </a&# x3E; <? echo(’<SCR)’; echo(’IPT>alert(“XSS”)</SCRIPT>’); ?>

cloud-based framework exhibits high performance as the observed value of *F*-Measure in the two platforms of web applications in greater than 0.9. Table highlights the detailed performance analysis of our mobile cloud-based XSS defensive framework on five different web applications. It is clearly reflected from the Table 7 that the performance of our XSS defensive framework on both the platforms of web applications is almost 99% as the highest value of *F*-Measure is 0.992.

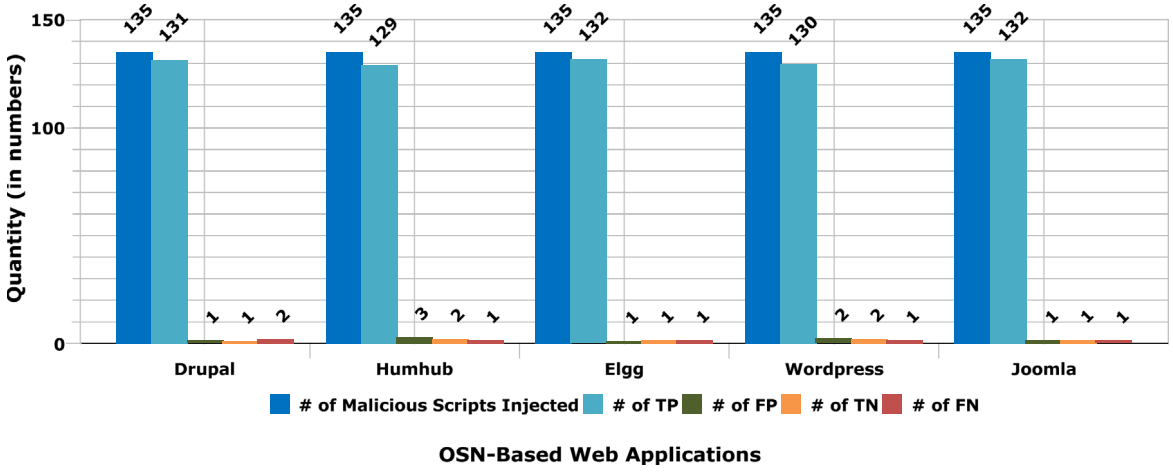

Fig. 17. Observed results on different platforms of web applications.

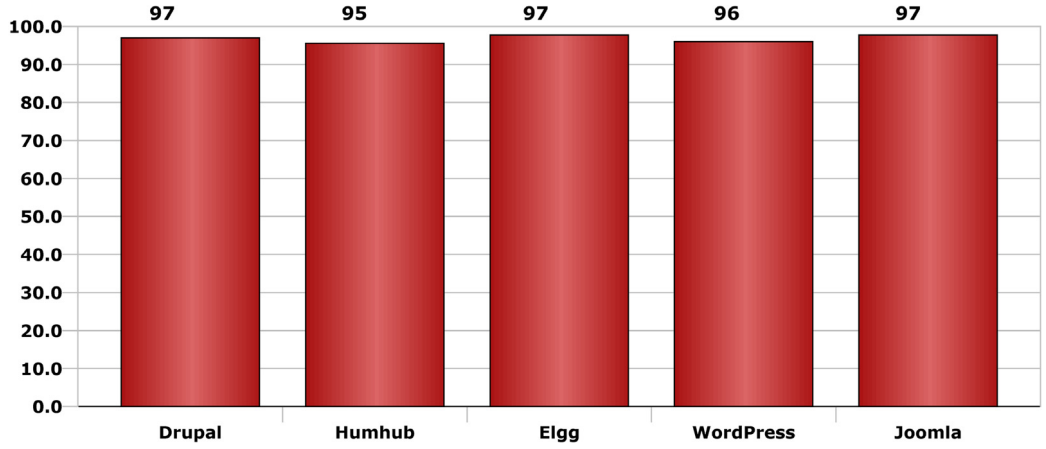

Fig. 18. DOM-Based XSS detection rate (in %age) in all HTML5 Web Applications.

3.3.2. Performance analysis using F-test

In order to prove that the number of malicious scripts detected (i.e. number of true positives) is less than to the number of malicious scripts injected; we use the *F*-test hypothesis, which is defined as:

Null Hypothesis: H0 = Number of malicious scripts detected is less than the number of malicious scripts injected. ($S1^2 = S2^2$).

Alternate Hypothesis: H1 = Number of malicious scripts injected is greater than number of malicious scripts detected ($S1^2 < S2^2$)

The level of significance is ($\alpha = 0.05$). The detailed analyses of statistics of XSS attack vectors applied and detected are illustrated in the Tables 8 and 9. In our work, we utilized and injected total of 135 XSS attacks vectors from the freely available XSS attack repositories [23–27] in all the five HTML5 web applications. But here note that, for evaluating the performance of our framework by using *F*-test, we injected different number of XSS attack vectors in all the five web applications.

# of Malicious Scripts Injected

# of Observation (N_1) = 5

Degree of Freedom dof (df_1) = $N_1 - 1 = 4$.

# of Malicious Scripts Detected

# of Observation (N_2) = 5

Degree of Freedom dof (df_2) = $N_2 - 1 = 4$.

$F_{CALC} = S_1^2/S_2^2 = 5.745609/4.999696 = 1.149$

The tabulated value of *F*-test at $df_1 = 4$, $df_2 = 4$ and $\alpha = 0.05$ is

$F_{(df1,df2,1-\alpha)} = F_{(4,4,0.95)} = 6.3882$

We know that the hypothesis that the two variances are equal (Null Hypothesis) is rejected if

$F_{CALC} < F_{(df1,df2,1-\alpha)}$.

Since $F_{CALC} < F_{(4,4,0.95)}$ therefore, we accept the alternate hypothesis (H1) that the first standard deviation (S_1) is less than the second standard deviation (S_2). Hence it is clear that the number of XSS worms detected is less than number of XSS attack vectors injected and we are 95% confident that any difference in the sample standard deviation is due to random error.

3.3.3. Performance analysis using response time

For better analysis of performance analysis, we have also calculated the response time of our cloud-based framework in validation phase mode for the detection and sanitization of injected attack vectors in different frameworks and cloud platforms. Nowadays, several IT organizations install the setup of their web applications in the infrastructures of cloud for better response time efficiency. The same optimized response time was observed in both the modes of proposed framework deployed in the virtual machines of cloud environment. Table 10 highlights the response time of our proposed framework in different environment (i.e. without cloud infrastructure) and cloud platform.

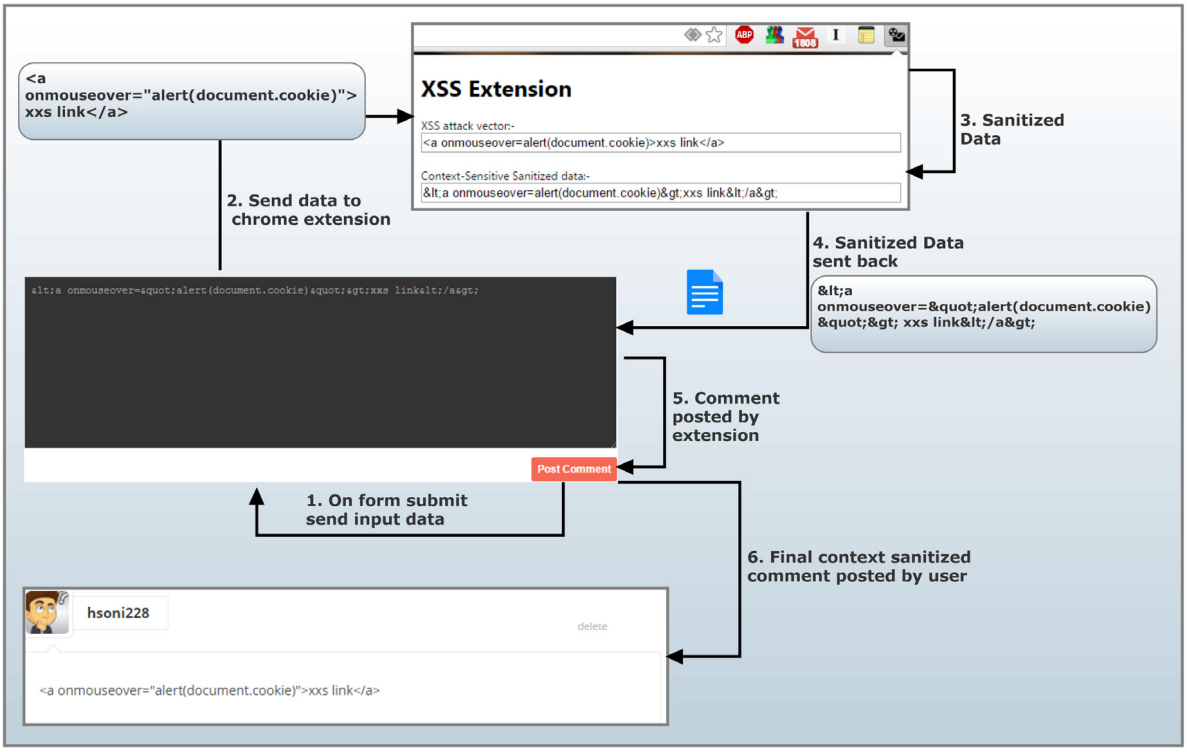

Fig. 19. Chrome Extension for general web application.

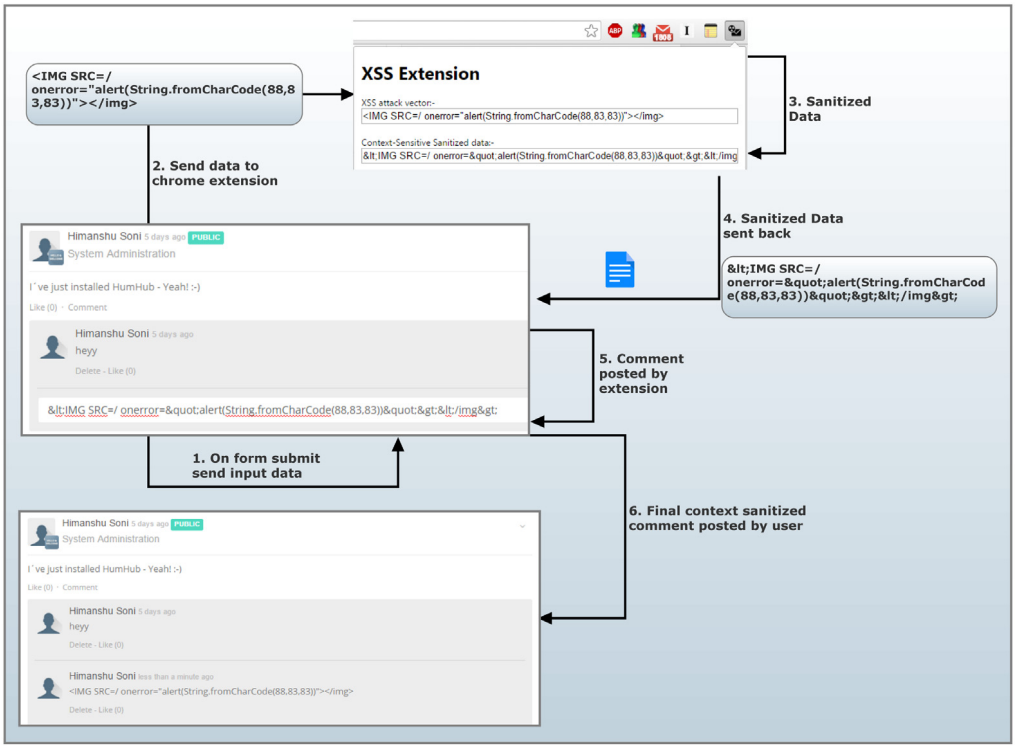

Fig. 20. Chrome extension for Humhub OSN-based web application.

3.4. Comparison-based analysis

This sub-section discusses the comparison of our proposed mobile cloud-based framework with the other recent existing XSS defensive methodologies. Table 11 compares the existing sanitization-based state-of-art techniques with our work based on nine identified metrics: DOM-Based XSS Attack Detection Proficiency (DXDP), Taint Tracking (Ttrac), Code Rewriting (CRW), Automated Pre-processing Required (APR), XSS attack Detection Proficiency (XDP), Source Code Monitoring (SCMon), Source Code Modification (SCMod), Scrutinizing Mechanism (SCMech) and Context-Aware Sanitization (CAS).

Table 7
Performance analysis of our mobile cloud-based framework by calculating F-measure.

Web applications	Total	# of TP	# of FP	# of TN	# of FN	Precision	FPR	FNR	Recall	F-Measure
Drupal [32]	135	131	1	1	2	0.992	0.5	0.015	0.984	0.988
Joomla [31]	135	129	3	2	1	0.977	0.6	0.007	0.992	0.984
Elgg [28]	135	132	1	1	1	0.992	0.5	0.007	0.992	0.992
Humhub [30]	135	129	3	2	1	0.977	0.6	0.076	0.992	0.984
Wordpress [29]	135	130	2	2	1	0.984	0.4	0.007	0.992	0.988

Table 8
Statistics of XSS attack vectors applied.

# of malicious scripts injected (X_i)	$(X_i - \mu)$	$(X_i - \mu)^2$	Standard deviation $S_1 = \sqrt{\sum_{i=1}^{N1}(Xi - \mu)^2/(N1 - 1)}$
130	0	0	
133	3	9	
129	−1	1	2.397
127	−3	9	
132	2	4	
Mean $(\mu) = \sum Xi/N1 = 130$		$\sum_{i=1}^{N1}(Xi - \mu)^2 = 23$	

Table 9
Statistics of XSS attack vectors detected.

# of JS malicious scripts detected (X_j)	$(X_i - \mu)$	$(X_i - \mu)^2$	Standard Deviation $S_2 = \sqrt{\sum_{i=1}^{N2}(Xi - \mu)^2/(N2 - 1)}$
127	0	0	
130	3	9	2.236
126	−1	1	
124	−3	9	
128	1	1	
Mean $(\mu) = \sum Xi/N2 = 127$		$\sum_{i=1}^{N1}(Xi - \mu)^2 = 20$	

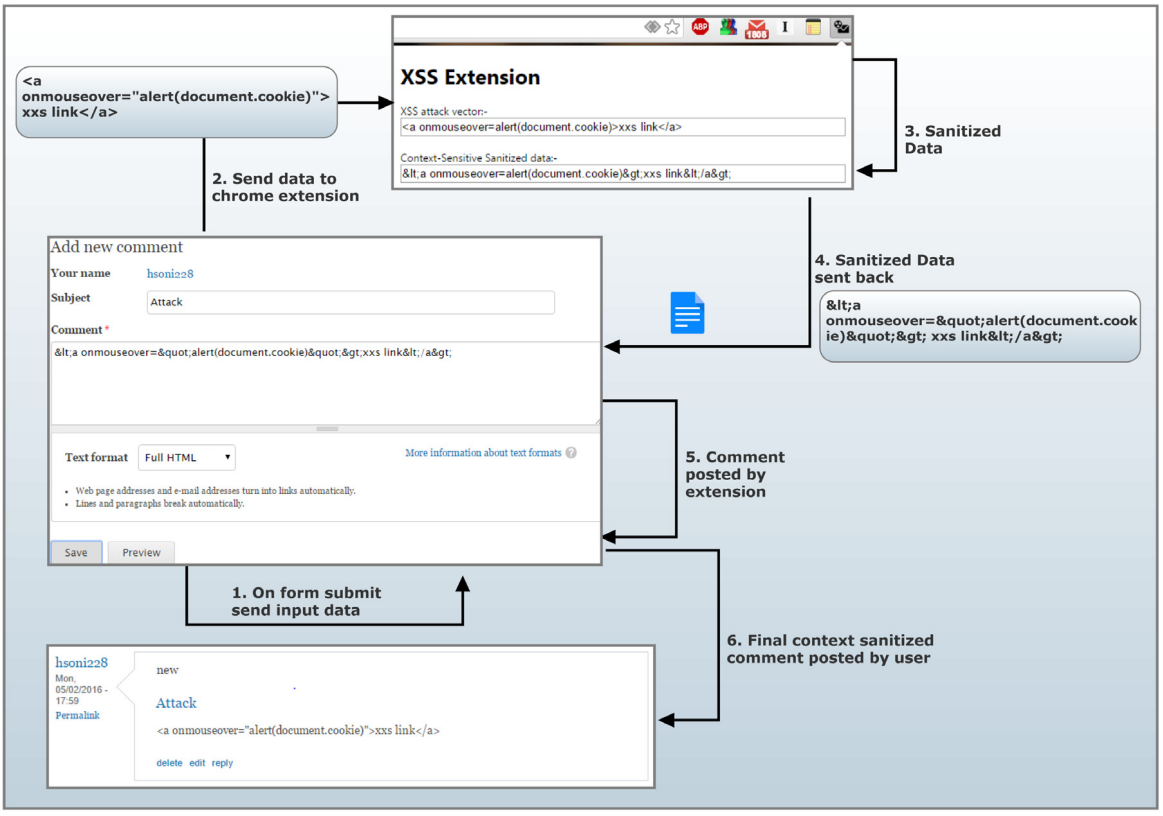

Fig. 21. Chrome extension for Drupal OSN-based Web application.

The false positive and false negative rate of our proposed work is very low and highly acceptable as the value of *F*-Measure in all the platforms of web applications is at least 90%. Therefore, the DOM-Based XSS Attack Detection Proficiency (DXDP) is acceptable in comparison to other related work. Proper isolation of untrusted code and data by Code Re-Writing (CRW) is still bypassed in the existing related work. In order to execute the process of sanitization of untrusted JavaScript code, the frameworks must initially track

Table 10
Performance analysis using response time calculation.

Web applications	Response time (in ms)	
	Without mobile cloud platform	On mobile-based cloud platform
Drupal	2096	2018
Joomla	2312	2216
Elgg	2789	2676
Humhub	3126	2986
Wordpress	2753	2697

Table 11
Summary of comparison of existing XSS defensive methodologies with our work.

Techniques	Metrics							
	DXDP	Ttrac	CRW	APR	SCMon	SCMod	SCMech	CAS
Lekies et al. [17]	Medium	No	No	Yes	Yes	Yes	Passive	No
Parameshwaran et al. [18]	Low	No	No	Yes	Yes	No	Passive	Yes
Bates et al. [4]	Medium	Yes	No	No	No	Yes	Active	No
Sun et al. [6]	Low	No	Yes	Yes	Yes	Yes	Passive	No
Cao et al. [3]	Medium	No	No	No	No	Yes	Passive	No
Balzarotti et al. [5]	Medium	No	No	Yes	Yes	No	Active	No
(Our Work)	Acceptable	Yes	Yes	No	No	No	Active	Yes

Table 12
Performance comparison of XSS-Auditor with our work.

Web Application	# of TP		# of FP		# of TN		# of FN		Precision		Recall		F-Measure	
	XA	OW	XA	OW	XA	OW	XA	OW	XA	OW	XA	OW	XA	OW
Drupal	102	131	8	1	11	1	5	2	0.772	0.992	0.739	0.984	0.726	0.988
Joomla	109	129	11	3	17	2	6	1	0.723	0.977	0.762	0.992	0.746	0.984
Elgg	105	132	13	1	15	1	7	1	0.710	0.992	0.772	0.992	0.782	0.992
Humhub	97	130	8	2	16	2	8	1	0.720	0.984	0.752	0.992	0.734	0.988
Wordpress	101	132	14	1	17	1	11	1	0.748	0.992	0.771	0.992	0.745	0.992

XA: XSS-Auditor, OW: Our Work.

the tainted flow of data. The existing techniques do not perform the Taint Tracking (Ttrac) of the untrusted variables of JavaScript attack vectors Nevertheless, the existing work [4] tracks the flow of tainted data in a static manner that is completely ineffective form of taint tracking. Moreover, lot of pre-processing is required in the existing framework of web applications for their successful execution on different platforms of web browsers as well as web applications. Context-aware sanitization is simply evaded by most of these existing sanitization-based techniques. Although, they perform the sanitization on the XSS attack vectors in a context-insensitive manner. Such sort of conventional sanitization methods are easily bypassed by the attackers.

On the other hand, we have also compared the performance analysis of existing client-side XSS filter (i.e. XSS-Auditor) with our mobile cloud-based framework. XSS-Auditor is installed as an extension of Google Chrome web browser. Here also, we have verified the XSS worm detection capability of XSS-Auditor by injecting 135 XSS worms on the five OSN-based web applications. Table 12 highlights the performance comparison of our work with existing client-side XSS filter (XSS-Auditor). It can be clearly observed from the Table XI that the value of F-Measure is decreasing in all the platforms of web applications for XSS-Auditor (in comparison to our work).

Although, XSS-Auditor performs the sanitization on the untrusted variables of JavaScript in a context-aware manner, yet this filter is not capable enough to determine all possible contexts of such untrusted variables. Therefore, the sanitization of such variables in such malicious variables becomes ineffective for the XSS-Auditor. However, our work is capable enough to determine the probable different nested contexts of malicious variables of JavaScript prior to the execution of context-sensitive sanitization procedure.

4. Conclusion and future work

In this article, we present a DOM generator and nested context-aware sanitization-based framework that detects and mitigates the effect of DOM-based XSS vulnerabilities from mobile cloud-based OSN. The framework operates in two modes: training and detection mode. The former mode retrieves all the benign script code embedded in the existing modules of web application by performing the deep traversal on the statically generated DOM tree. The detection mode scans the dynamically generated DOM tree for the injection of untrusted/malicious script from the remote servers by comparing the white list of scripts with the script code embedded in this tree. In addition, the injection of context-sensitive sanitization primitives will be executed on such malicious code for its safe interpretation on the web browser of mobile user. Experimental evaluation revealed that our framework detects the injection of untrusted JavaScript attack vectors with very low and acceptable false positive and false negatives. In addition, the response time of runtime nested auto-context-sensitive sanitization was optimized and incurs less performance overhead in the detection mode as compared to existing DOM-based XSS defensive methodologies. We will try to evaluate the DOM-based XSS attack detection capability of our framework on some more contemporary platforms of mobile cloud-based OSN web applications as a part of our further work.

Acknowledgments

We would like to thank the anonymous reviewer(s), who have facilitated us with their valuable comments and feedback throughout the review process of this manuscript. The authors would also like to thank the support of Information and Cyber Security Research Group working in the National Institute of Technology Kurukshetra, India for their valuable feedbacks and worthwhile

discussions. This work was financially supported by TEQIP-II and MHRD with the economic aid provided by the World Bank.

References

[1] S.C.V, S. Selvakumar, Bixsan: Browser independent XSS sanitizer for prevention of XSS attacks, ACM SIGSOFT Softw. Eng. Notes 36 (5) (2011) 1.
[2] 2015 Website Security Statistics Report – WhiteHat Security. Retrieved from: http://info.whitehatsec.com/rs/whitehatsecurity/images/statsreport2015.pdf.
[3] Yinzhi Cao, et al., PathCutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks, NDSS, 2012.
[4] D. Bates, A. Barth, C. Jackson, Regular expressions considered harmful in client-side XSS filters, in: Proceedings of the Conference on the World Wide Web, 2010, pp. 91–100.
[5] Davide Balzarotti, et al., Saner: Composing static and dynamic analysis to validate sanitization in web applications, in: Security and Privacy, 2008 SP 2008 IEEE Symposium on, IEEE, 2008.
[6] Fangqi Sun, Xu Liang, Su Zhendong, Client-side detection of XSS worms by monitoring payload propagation, in: Computer Security–ESORICS 2009, Springer, Berlin, Heidelberg, 2009, pp. 539–554.
[7] Shashank Gupta, B.B. Gupta, JS-SAN: defense mechanism for HTML5-based web applications against javascript code injection vulnerabilities, Secur. Commun. Netw. (2016).
[8] Gupta Shashank, B.B. Gupta, XSS-SAFE: A server-side approach to detect and mitigate cross-site scripting (XSS) attacks in javascript code, Arab. J. Sci. Eng. (2015) 1–24.
[9] Shashank Gupta, B.B. Gupta, 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, vol. 59, ACM, 2015.
[10] B.B. Gupta, Shashank Gupta, S. Gangwar, M. Kumar, P.K. Meena, Cross-site scripting (XSS) abuse and defense: exploitation on several testing bed environments and its defense, J. Inf. Privacy Secur. 11 (2) (2015) 118–136.
[11] Shashank Gupta, B.B. Gupta, BDS: browser dependent XSS sanitizer, in: Book on Cloud-Based Databases with Biometric Applications, in: IGI-Global's Advances in Information Security, Privacy, and Ethics (AISPE) Series, IGI-Global, Hershey, 2014, pp. 174–191.
[12] S. Gupta, B.B. Gupta, Cross-site scripting (XSS) attacks and defense mechanisms: classification and state-of-Art, in: International Journal of System Assurance Engineering and Management, Springer, 2015.
[13] Shashank Gupta, B.B. Gupta, XSS-secure as a service for the platforms of online social network-based multimedia web applications in cloud, Multimedia Tools Appl. (2016) 1–33.
[14] S. Gupta, B.B. Gupta, An infrastructure-based framework for the alleviation of javascript worms from OSN in mobile cloud platforms, in: International Conference on Network and System Security, Springer International Publishing, 2016, pp. 98–109.
[15] R. Pelizzi, R. Sekar, Protection, usability and improvements in reflected XSS filters, in: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, 2012.
[16] Saxena Prateek, David Molnar, Benjamin Livshits, SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, ACM, 2011, pp. 601–614.
[17] Lekies Sebastian, Ben Stock, Martin Johns, 25 million flows later: large-scale detection of DOM-based XSS, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, ACM, 2013.
[18] Parameshwaran Inian, et al., DexterJS: robust testing platform for DOM-based XSS vulnerabilities, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015.
[19] David Ross, IE 8 XSS filter architecture/implementation, August 2008. Retrieved from: http://blogs.technet.com/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation. Aspx.
[20] Jeremias Reith, Internals of noXSS, October 2008. Retrieved from: http://www.noxss.org/wiki/Internals.
[21] Giorgio Maone, NoScript. Retrieved from: http://www.noscript.net.
[22] I. Yusof, A-S.K. Pathan, Preventing persistent Cross-Site Scripting (XSS) attack by applying pattern filtering approach, in: 5th International Conference on Information and Communication Technology for The Muslim World, ICT4M, Kuching, 2014, pp. 1-6.
[23] R. Hansen, XSS (cross site scripting) cheat sheet [online]. Available: https://www.owasp.org/index.php/XSS Filter Evasion Cheat Sheet.
[24] M. Heiderich, Html5 security cheatsheet [online]. Available: http://html5sec.org.
[25] 523 XSS vectors available [online]. Available: http://xss2technomancie.net/vectors.
[26] Technical Attack Sheet for Cross Site Penetration Tests [online]. Available: http://www.vulnerability-lab.com/resources/documents/531.txt.
[27] @XSS Vector Twitter Account [online]. Available: https://twitter.com/XSSVector.
[28] Elgg social networking engine. Available: https://elgg.org.
[29] WordPress. http://wordpress.org/.
[30] Humhub social networking site. Available: https://www.humhub.org/en.
[31] Joomla social networking site. Available: https://www.joomla.org/download.html.
[32] Drupal social networking site. Available: https://www.drupal.org/download.

Shashank Gupta is currently working as an Assistant Professor in the Department of Computer Science and Engineering at Jaypee Institute of Information Technology (JIIT), Noida, Sec-128. Recently, he has submitted his Ph.D. thesis work under the supervision of Dr. B. B. Gupta in Department of Computer Engineering specialization in Web Security at National Institute of Technology Kurukshetra, Haryana, India. Prior to this, he has also served his duties as a Lecturer in the Department of IT at Model Institute of Engineering and Technology (MIET), Jammu. He has completed M.Tech. in the Department of Computer Science and Engineering Specialization in Information Security from Central University of Rajasthan, Ajmer, India. He has also done his graduation in Bachelor of Engineering (B.E.) in Department of Information Technology from Padmashree Dr. D.Y. Patil Institute of Engineering and Technology Affiliated to Pune University, India. He has also spent two months in the Department of Computer Science and IT, University of Jammu for completing a portion of Post-graduation thesis work. He bagged the 1st Prize of Cash Rs. 3000/- and consolation prize of cash Rs. 1000/- in Poster Presentation at National Level in the category of ICT Applications in Techspardha'2015 and 2016 event organized by National Institute of Kurukshetra, Haryana. He has numerous online publications in International Journals and Conferences including IEEE, Elsevier, ACM, Springer, Wiley, Elsevier, IGI-Global, etc. along with several book chapters. He is also serving as reviewer for numerous peer-reviewed Journals and conferences of high repute. He is also a student member of IEEE and ACM. His research area of interest includes Web Security, Cross-Site Scripting (XSS) attacks, Online Social Network Security, Cloud Security, Fog Computing and theory of Computation.

B.B. Gupta received Ph.D. degree from Indian Institute of Technology Roorkee, India in the area of Information and Cyber Security. In 2009, he was selected for Canadian Commonwealth Scholarship and awarded by Government of Canada Award ($10,000). He spent more than six months in University of Saskatchewan (UofS), Canada to complete a portion of his research work. Dr. Gupta has excellent academic record throughout his carrier, was among the college toppers, during Bachelor's degree and awarded merit scholarship for his excellent performance. In addition, he was also awarded Fellowship from Ministry of Human Resource Development (MHRD), Government of India to carry his Doctoral research work. He has published more than 70 research papers (including 01 book and 08 chapters) in International Journals and Conferences of high repute including IEEE, Elsevier, ACM, Springer, Wiley Inderscience, etc. He has visited several countries, i.e. Canada, Japan, Malaysia, Hong-Kong, etc. to present his research work. His biography was selected and publishes in the 30th Edition of Marquis Who's Who in the World, 2012.

He is also working principal investigator of various R&D projects. He is also serving as reviewer for Journals of IEEE, Springer, Wiley, Taylor & Francis, etc. Currently he is guiding 08 students for their Master's and Doctoral research work in the area of Information and Cyber Security. He also served as Organizing Chair of Special Session on Recent Advancements in Cyber Security (SS-CBS) in IEEE Global Conference on Consumer Electronics (GCCE), Japan in 2014 and 2015. Earlier he served as co-convener of National Conference on Emerging Trends in Engineering, Science Technology & Management (ETESTM-12), India, April, 2012. In addition, Dr. Gupta received Best Poster presentation award and People choice award for Poster presentation in CSPC-2014, Aug., 2014, Malaysia. He served as Jury in All IEEE-R10 Young Engineers' Humanitarian Challenge (AIYEHUM-2014), 2014. He has also served as founder and organizing chair of International Workshop on Future Information Security, Privacy and Forensics for Complex Systems (FISP-2015) in conjunction with ACM International Conference on Computing Frontiers (CF-2015), Ischia, Italy in May 2015. He is also serving as guest editor of various Journals. He is also serving as guest editor of various Journals.

Pooja Chaudhary has received her B.Tech. degree in Computer Science and Engineering from Bharat Institute of Technology, Meerut, India. Currently, she is pursuing M.Tech. in Computer Engineering from National Institute of Technology, Kurukshetra, Haryana, India. Her research interest includes Online Social Network security, Big data analysis and security, Database security and cyber security.