Enhancing the Browser-Side Context-Aware Sanitization of Suspicious HTML5 Code for Halting the DOM-Based XSS Vulnerabilities in Cloud

Article *in* International Journal of Cloud Applications and Computing · March 2017

DOI: 10.4018/IJCAC.2017010101

Shashank Gupta
Birla Institute of Technology and Science Pilani
56 PUBLICATIONS **672** CITATIONS
SEE PROFILE

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

Enhancing the Browser-Side Context-Aware Sanitization of Suspicious HTML5 Code for Halting the DOM-Based XSS Vulnerabilities in Cloud

B.B. Gupta, National Institute of Technology, Department of Computer Engineering, Kurukshetra, India
Shashank Gupta, National Institute of Technology, Department of Computer Engineering, Kurukshetra, India
Pooja Chaudhary, National Institute of Technology, Department of Computer Engineering, Kurukshetra, India

ABSTRACT

This article presents a cloud-based framework that thwarts the DOM-based XSS vulnerabilities caused due to the injection of advanced HTML5 attack vectors in the HTML5 web applications. Initially, the framework collects the key modules of web application, extracts the suspicious HTML5 strings from the latent injection points and performs the clustering on such strings based on their level of similarity. Further, it detects the injection of malicious HTML5 code in the script nodes of DOM tree by detecting the variation in the HTML5 code embedded in the HTTP response generated. Any variation observed will simply indicate the injection of suspicious script code. The prototype of our framework was developed in Java and installed in the virtual machines of cloud environment on the Google Chrome extension. The experimental evaluation of our framework was performed on the platform of real world HTML5 web applications deployed in the cloud platform.

KEYWORDS

Context-Sensitive Sanitization, Cross-Site Scripting (XSS) Worms, Document Object Model (DOM) Tree, DOM-Based XSS Vulnerabilities, HTML5 Web Applications

INTRODUCTION

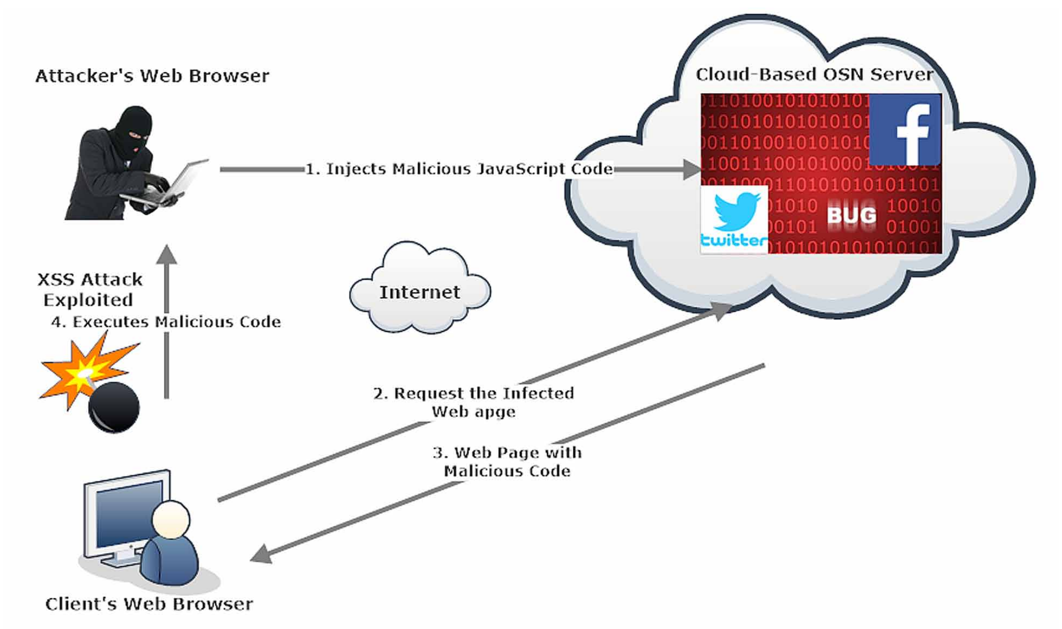
The tremendous explosion in cloud computing produced numerous security issues related to data security of cloud users (Dinh et al., 2013; Gupta et al., 2016c. The propagation of XSS worms are considered to be topmost threat originated in HTML5 web applications deployed in the framework of cloud infrastructure. In the contemporary era of cloud computing, cloud security has turned out to be a serious issue, as numerous on-demand resources are being offered by utilizing the virtualization technologies of cloud services (Modi et al., 2013). Instead of referring the outdated Internet settings for constructing an expensive setup, numerous commercial IT organizations are accessing the services of Online Social Networking (OSN) sites (such as Twitter, Facebook, LinkedIn, etc.) on the cloud platforms. In the modern era of Web 2.0 technologies and HTML5-based web applications, OSN is considered to be the most popular method for information sharing has drawn most of public attention. However, it is clearly known that the cloud settings are installed on the backbone of Internet. Therefore, numerous HTML5 web application vulnerabilities in the conventional Internet infrastructures also exist in the backgrounds of cloud-based environments.

DOI: 10.4018/IJCAC.2017010101

The most prominent attack found on HTML5 web applications is the Cross Site Scripting (XSS) attack [Gupta et al. (2016a), Gupta et al. (2016b), Gupta et al. (2015a), Gupta et al. (2015b), Gupta et al. (2014)]. Figure 1 highlights the injection of XSS worm on the OSN web server deployed in the virtual machines of cloud platforms. XSS worms have turned out to be a plague for the cloud–based HTML5 web applications. Such worms steal the sensitive credentials of the active users by injecting the malicious HTML5 script code in the form of some posts on such web applications [Gupta et al. (2015c), Duchene et al. (2014), Shahriar et al. (2011), Doupe et al. (2013), Chandra et al. (2011), Xiao et al. (2014)]. Input sanitization is considered to be the most effective mechanism for alleviating and mitigating the effect of XSS worms from the cloud-based HTML5 web applications on the virtual machines of cloud platforms. Numerous defensive methodologies had been proposed for thwarting the effect of XSS worms or XSS attacks from such platforms (Aljawarneh, 2011a; Aljawarneh, 2011b; Aljawarneh et al., 2016).

Usually, XSS attack comes in variety of three different flavors: Reflected, Stored and DOM-based XSS attack. Numerous XSS defensive methodologies have been designed for detecting and mitigating the effect of reflected and XSS worms [Parameshwaran et al. (2015), Lekies et al. (2013), David et al. (2008)]. However, very few solutions have been designed for completely alleviating the effect of DOM-Based XSS vulnerabilities from the HTML5-based web applications. 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 HTML5 Web application and display results on browser. If the data is not validated, then an attacker may inculcate the malicious scripts that is stored as a part of DOM and gets executed when data is read from DOM and triggers adverse effects. Figure 2 highlights the vulnerable code utilized for the injection of DOM-based XSS attack on HTML5 web application.

Figure 1. Exploitation of XSS attack on cloud platform


Attacker's Web Browser
Cloud-Based OSN Server
1. Injects Malicious JavaScript Code
BUG
twitter
XSS Attack Exploited
4. Executes Malicious Code
Internet
2. Request the Infected Web apge
3. Web Page with Malicious Code
Client's Web Browser

EXPLOITATION OF DOM-BASED XSS ATTACK ON CLOUD-BASED HTML 5 WEB APPLICATIONS

This class of XSS attack is somewhat analogous to reflected XSS attacks. The only dissimilarity is that there is no need of transmission of data on the web server since, all things will execute at the browser. Therefore, there is no mechanism of sanitization present on the web server for this class of attack. Mechanism of sanitization is required on the web browser. In Figure 3, by utilizing the JavaScript code, the username is displayed on HTML page in the browser. In reflected XSS attack, the JS code run as well as display a dialog box on the victim's browser. However, for the exploitation of DOM-based XSS attack, there is no requirement of generating a dialog box on the browser by the attacker. Instead, attacker can run random malicious JS code. Figure 3 illustrates the exploitation of DOM-based XSS attack on the vulnerable cloud-based OSN web server.

Following are 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 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 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 user as HTTP response.
- 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 HTML5 script performs maliciously and sends sensitive information to the attacker's domain.

EXISTING DOM-BASED XSS DEFENSIVE SOLUTIONS

Parameshwaran (Parameshwaran et al. (2015)) 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 analyses the tainted flows, identified by the first module, and constructs context-based test payloads, which can be simply tested on the vulnerable web site. Lekis (Lekies et al. (2013)) 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. Bates et al. (Bates et al. (2010) broadly investigate the condition of the ability of these three-client side XSS filters (IE8 (David et al. (2008)), noXSS (Jeremias, 2008) and NoScript (Giorgio, n.d)) 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.

Cao (Cao, 2012) 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. Sun (Sun, 2009) proposed an entirely client-side solution, employed as a Firefox plug-in that utilizes

Figure 2. Vulnerable code snippet for DOM-Based XSS attack

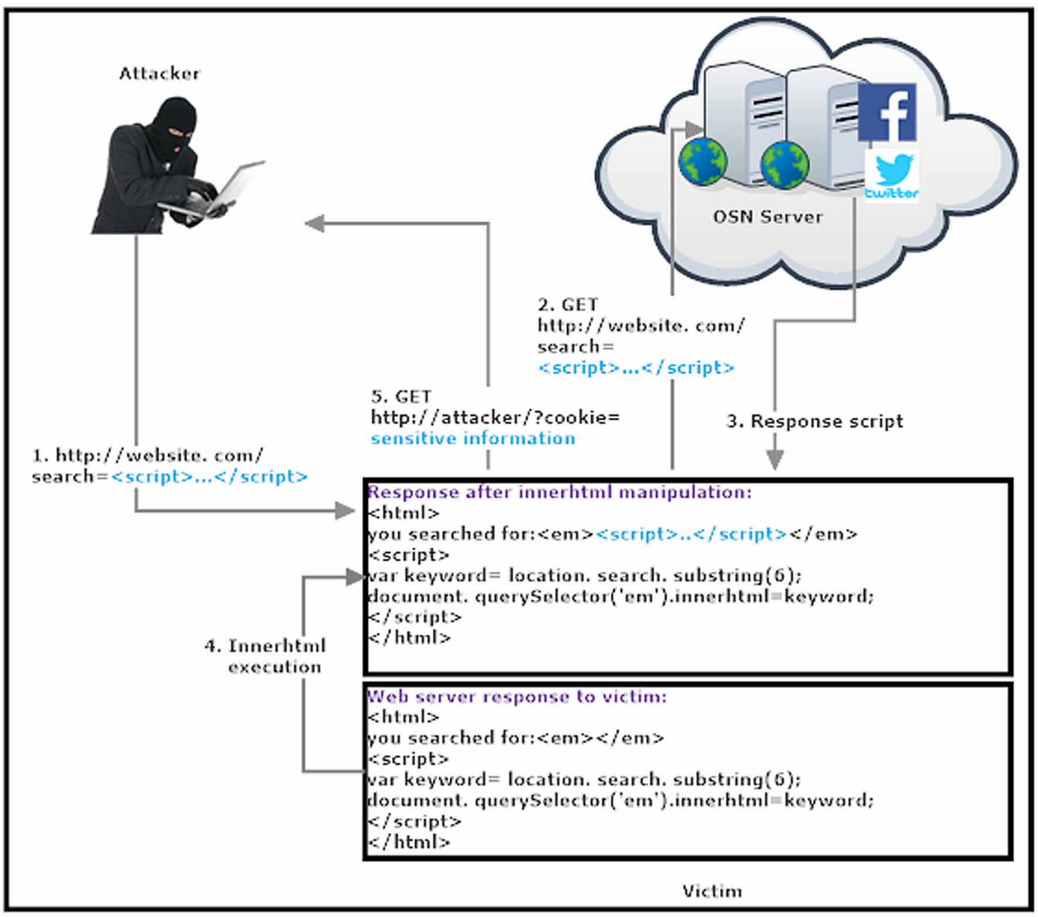
```
<html>
<body>
  <script type ="“text/javascript”>
    var usernamePos =
window.location.search
        .indexof(“username=”);
    if (usernamePos >= 0)
    {
          var username =
window.location.serach
                .substr(usernamePos + 9);
          document.write(username);
    }
</script>
</body>
</html>
```

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. Balzarotti (Balzarotti et al., 2008) 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.

EXISTING CHALLENGES

Although, numerous defensive solutions have been proposed recently, however, they are not capable enough to integrate in the existing infrastructures of cloud. Moreover, existing open source solutions cannot detect and alleviate the novel HTML5 XSS attack vectors from the cloud-based HTML5 web applications. Following are some of the untouched security challenges that needs to be integrated in the existing framework of DOM-based XSS defensive solutions:

Figure 3. Exploitation of DOM-Based XSS attack on cloud-based OSN


Attacker
OSN Server
2. GET http://website. com/ search= <script>...</script>
5. GET http://attacker/?cookie= sensitive information
3. Response script
1. http://website. com/ search=<script>...</script>
4. Innerhtml execution
Victim

```
Response after innerhtml manipulation:
<html>
you searched for:<em><script>..</script></em>
<script>
var keyword= location. search. substring(6);
document. querySelector('em').innerhtml=keyword;
</script>
</html>
```

```
Web server response to victim:
<html>
you searched for:<em></em>
<script>
var keyword= location. search. substring(6);
document. querySelector('em').innerhtml=keyword;
</script>
</html>
```

Improper Handling of Client-Side Script Injection

In order to exploit the vulnerabilities of the two categories of XSS worms (i.e., stored and reflected), 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 had been proposed for detecting and mitigating the effect of client-side malicious script injection. Properties like innerHTML, document.location, etc. are being utilized by an attacker to inject the malicious/untrusted scripts on the web browser. 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 XSS attack on the client-side. Nowadays, attacker utilizes and injects the HTML5 XSS attack vectors, alter the structure of the tree and exploit the DOM-based XSS vulnerability on the web browser.

Exclusion of New XSS Attack Vector Repositories

The XSS worm detection capability of framework of existing literature were usually tested and evaluated by referring the XSS attack vectors from single source repository (i.e. XSS cheat sheet). However, there are four other XSS attack vector repositories available on the Internet that includes latest HTML5 XSS attack vectors too. Most of the recent work has injected the XSS attack vectors on the vulnerable injection points of real world web applications. On the other hand, the machine learning-based XSS defensive methodologies had also calculated the features of this vulnerable JavaScript code by referring the XSS cheat sheet. Therefore, the XSS worm detection capability of robust solution must be evaluated by referring the XSS attack vectors from other four contemporary repositories.

No Sanitization Support for New HTML5 Features

In the contemporary era of World Wide Web (WWW), HTML5 is being utilized as an emerging platform for the development of modern web applications. The key advantage of adopting this feature is that it can be easily integrated among other platforms of web browsers. However, it introduces some new tags and attributes (such as <video>, <source>, <autofocus>, etc.) which can be utilized for creating new XSS attack vectors.

<video><source onerror= “alert(1)”></video>

The modern web browsers or existing XSS filters does not check for this HTML5 attack vector. A simple popup window will appear with message ‘1’ on the screen. Therefore, a robust 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.

Unable to Determine Nested Context

Existing literature [(Bates et al. (2010), Cao et al. (2012), Sun et al. (2009)] introduced some of the XSS defensive mechanisms that perform the context-sensitive sanitization on the untrusted/malicious variables of HTML5 and JavaScript code. Such technique determines the context of unsafe JavaScript/HTML5 variables and accordingly performs the sanitization on them. However, this sort of sanitization is no longer effective as it does not determine the nested context of such untrusted variables. Therefore, most of the inner/nested context of such variables is uncovered with the sanitization routines that lead to the exploitation of XSS worms. The XSS defensive technique must incorporate a mechanism of determining the nested context of such malicious HTML5variables and must perform the accurate placement of sanitization routines in such contexts.

Non-Optimized Scalability

Most of the existing XSS defensive solutions were tested and evaluated on the tested bed of open source HTML-based web applications. In addition, very few existing defensive frameworks were tested on the real world platforms of HTML5 OSN-based web applications. On the other hand, some of the recent frameworks demands major alterations in the existing infrastructure of web applications. Moreover, the setup of existing framework of XSS defensive solutions couldn’t be easily integrated in the virtual machines of cloud platforms. Although, no XSS defensive solutions has been designed for the platforms of cloud-based HTML5 web applications.

Instead of referring the out-dated Internet settings for constructing an expensive setup, numerous commercial IT organizations are accessing the services of OSN sites (such as Twitter, Facebook, Linkedin, etc.) on the cloud platforms. Hence, the infrastructure settings of framework solution must be capable enough to integrate in the settings of cloud environment and should evaluate the XSS worm recognition capability on cloud-based OSN platforms.

KEY CONTRIBUTIONS

Based on the severe performance issues, this article presents a cloud-based framework that scans for the DOM-based XSS vulnerabilities in the cloud-based HTML5 web applications. The framework operates in two phases: learning and validation phase. The former phase generates the clustered HTML5 attack vector templates by calculating the measure of similarity between the two extracted HTML5 attack vector strings. The clustering on such JavaScript strings optimizes the process of context-sensitive sanitization in comparison to existing XSS defensive techniques. The later phase detects the injection of untrusted HTML5 code in the DOM tree by detecting the variation in the embedded set of HTML5 code in the HTTP response messages of both the phases of our work. Any variation will indicate the injection of malicious HTML5 code in the script nodes of DOM tree. In addition, the possible different context of malicious variables embedded in such code will be identified and accordingly performs the context-sensitive sanitization on them for completely alleviating the effect of DOM-based XSS vulnerabilities from the cloud-based HTML5 web applications.

The prototype of our cloud-based framework was developed in Java development framework and integrated its settings on the virtual machines of mobile cloud platforms. Experimental evaluation of our framework was done on the tested suite of real world HTML5 web applications by referring the latest XSS attack vectors from the freely available XSS attack vector repositories (RSnake, 2008). Performance evaluation results revealed that our work detects the injection of XSS worms in the runtime generation of DOM tree with tolerable rate of false positives and false negatives and experienced acceptable performance overhead during the runtime sanitization of malicious HTML5 attack vector strings.

The rest of the paper is organized as follows: In section 2, we have introduced our work in detail. Implementation and evaluation of our work are discussed in section 3. Finally, section 4 concludes our work and discusses further scope of work.

PROPOSED DESIGN

This article presents a framework that thwarts the DOM-based XSS vulnerabilities from the HTML5 web applications deployed in the cloud platforms. The key focus of our technique is to enhance and optimize the context-sensitive sanitization procedure of HTML5 attack payloads by determining the nested context of suspicious variables embedded in such code. The novelty of our framework lies in the fact that it performs the context-sensitive sanitization on the clustered templates of malicious/untrusted HTML5 script payloads. The recent work executes this sanitization process on the individual HTML5 attack vector payloads that consumes more time in policy checks and degrades the runtime performance of web browsers in cloud environment. However, in order to optimize the process of context-sensitive sanitization, our work initially performs the clustering on the similar suspicious HTML5 strings and accordingly performs the context-aware sanitization on them. This process overall increases the web surfing experience of users on the cloud platforms and enhances the context-aware sanitization process in comparison to existing XSS defensive techniques. The next sub-section discusses the detailed design overview of our cloud-based XSS defensive framework.

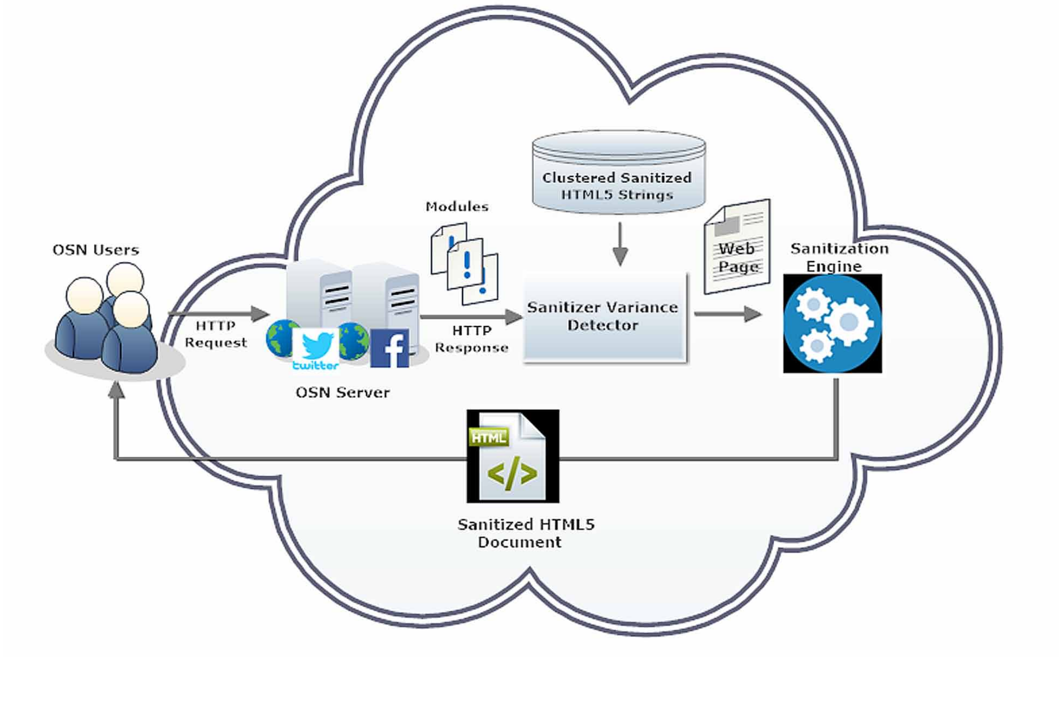
ABSTRACT DESIGN OVERVIEW

The proposed framework is a clustering and context-aware sanitization-based framework that obstructs the execution of DOM-based XSS attacks from HTML5-based web applications. Figure 4 highlights the abstract design overview of our framework. The framework extracts the web application modules embedded in the sanitized HTTP response. Such HTTP response will undergo through sanitizer variance detector for detecting the injection of suspicious HTML5 code by detecting the variation in the clustered sanitized HTTP responses generated at both the phases (i.e. learning and validation phase) of our framework. Any variation will simply locate the injection of suspicious HTML5 code in the HTTP response generated at the learning phase. The sanitization engine will find out the diverse context of such malicious HTML5 code and accordingly implements the sanitization primitives on them for alleviating the execution of such code on the mobile cloud platforms.

DETAILED DESIGN OVERVIEW

The proposed cloud-based XSS defensive framework executes in two phases: learning and validation. The learning phase executes in offline mode and activates the code tracing technique on the extracted modules of HTML5 web applications. These modules will be explored for the vulnerable injection points and the HTML5 script code embedded in such points. The key motto of this phase is to generate the clustered templates of extracted similar malicious HTML5 script and consequently executes the sanitization process on such templates in a context-sensitive manner. On the other hand, the validation phase generates the DOM tree of HTTP response generated at the OSN web server. This tree will be explored for the script nodes which are generated dynamically on the server. The extracted script nodes will undergo through the process of de-obfuscation and extract the decoded HTML5 script code. The variation indicator component will detect the variation in this obfuscated set of script code and the sanitized JavaScript strings generated at learning phase. Any variation observed in both these set of JavaScript strings will indicate the injection of malicious HTML5 code in the DOM tree generated at the OSN server. The context of malicious variables embedded in such strings will be identified, performs the context-aware sanitization on them and finally sanitized HTTP response is redirected to the web browser. Figure 6 highlights the detailed overview of cloud-based XSS defensive framework. The next-subsection discusses the two phases and their corresponding modules of cloud-based framework in detail.

Figure 4. Abstract view of our framework



LEARNING PHASE

The key goal of this phase is to perform the clustering on the extracted malicious HTML5 script code strings based on their level of similarity and consequently generate the sanitized clustered templates by executing the process of context-aware sanitization on them. Figure 5 highlights the detailed algorithm of learning phase.

This algorithm works as follows: Initially, it generates HTTP response H_{RES} corresponding to each H_{REQ}. Then, it extracts the specified web page WP_I by exploring the internal links in the response. Then, it parses the WP_I and construct DFG[N][T]. All HTML5 code then shifted to the external HTML5 file ext.file. To reduce the sanitization overhead, it performs the clustering on the extracted HTML5 code and store the result into C_Log. Then, it executes sanitization_engine functionality and stores the results into the S_Log. Finally, it outputs S_Log to be forwarded to the client.

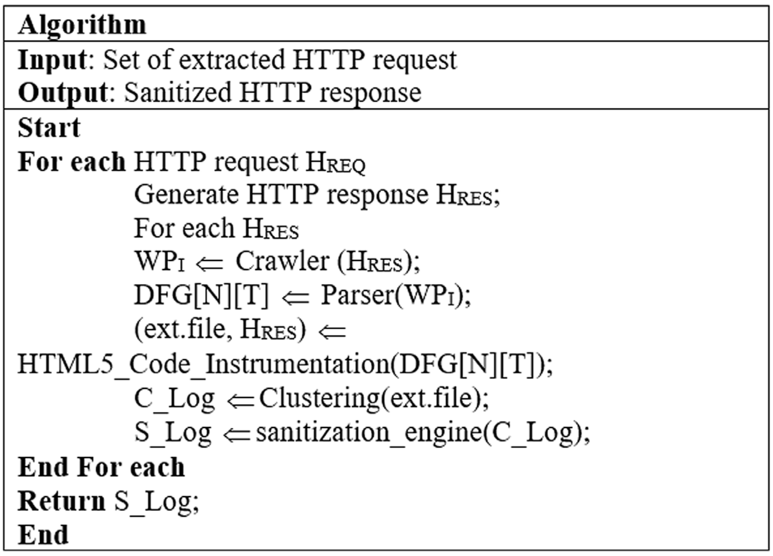
Key Modules of Learning Phase

Here, note that, the modules of learning phase are executed in offline mode and installed on the OSN web server. Following are some of the details of key modules of learning phase cloud-based framework.

Code Tracer

It consists of two key sub-components: web crawler and HTML5 parser. The web crawler component initially receives the web application module generated as the HTTP response by the OSN server. Its key goal is to construct a parse tree by the parser i.e. Document Object Model (DOM). It is a method by which browser interprets the document and display it to the user. During the parsing phase, executable HTML5 script nodes are determined and nodes are created for them in the parse tree. In addition to this, data nodes are also created in this component. Finally, this tree is passed to the next subsequent module for further JavaScript code instrumentation.

Figure 5. Algorithm of learning phase



```
Algorithm
Input: Set of extracted HTTP request
Output: Sanitized HTTP response
Start
For each HTTP request H_REQ
            Generate HTTP response H_RES;
            For each H_RES
            WP_I ⇐ Crawler (H_RES);
            DFG[N][T] ⇐ Parser(WP_I);
            (ext.file, H_RES) ⇐
HTML5_Code_Instrumentation(DFG[N][T]);
            C_Log ⇐Clustering(ext.file);
            S_Log ⇐sanitization_engine(C_Log);
End For each
Return S_Log;
End
```

HTML5 Code Instrumentation

The key goal of this module is to extract all the HTML5 script code embedded in the DOM tree. In this step, DOM tree is processed to identify all HTML5 code. In this tree, each possible path from root to leaf node depicts one output of web page.

Thus, a standard algorithm is used to compute all paths and examine each output statement to identify the HTML5 code. Nevertheless, it is not possible to manage all identified paths, since even in a simple web page unique path may explode very fast. To ease this problem, we search in the DOM tree to check for the opening and closing HTML5 tags. Then, for every couple of JavaScript tags (<script>…. </script>), we inspect each unique path between opening and closing tags by using graph traversal algorithm such as Depth First Search (DFS). Each identified path represents the JavaScript code that the web page code might result. Figure 7 highlights the algorithm for HTML5 string code instrumentation and extraction. This algorithm implements as follows: Input to the algorithm is the constructed data flow graph DFG[N][T]. It analyses each node n_I of the graph to check whether it

Figure 6. Detailed design view of our framework

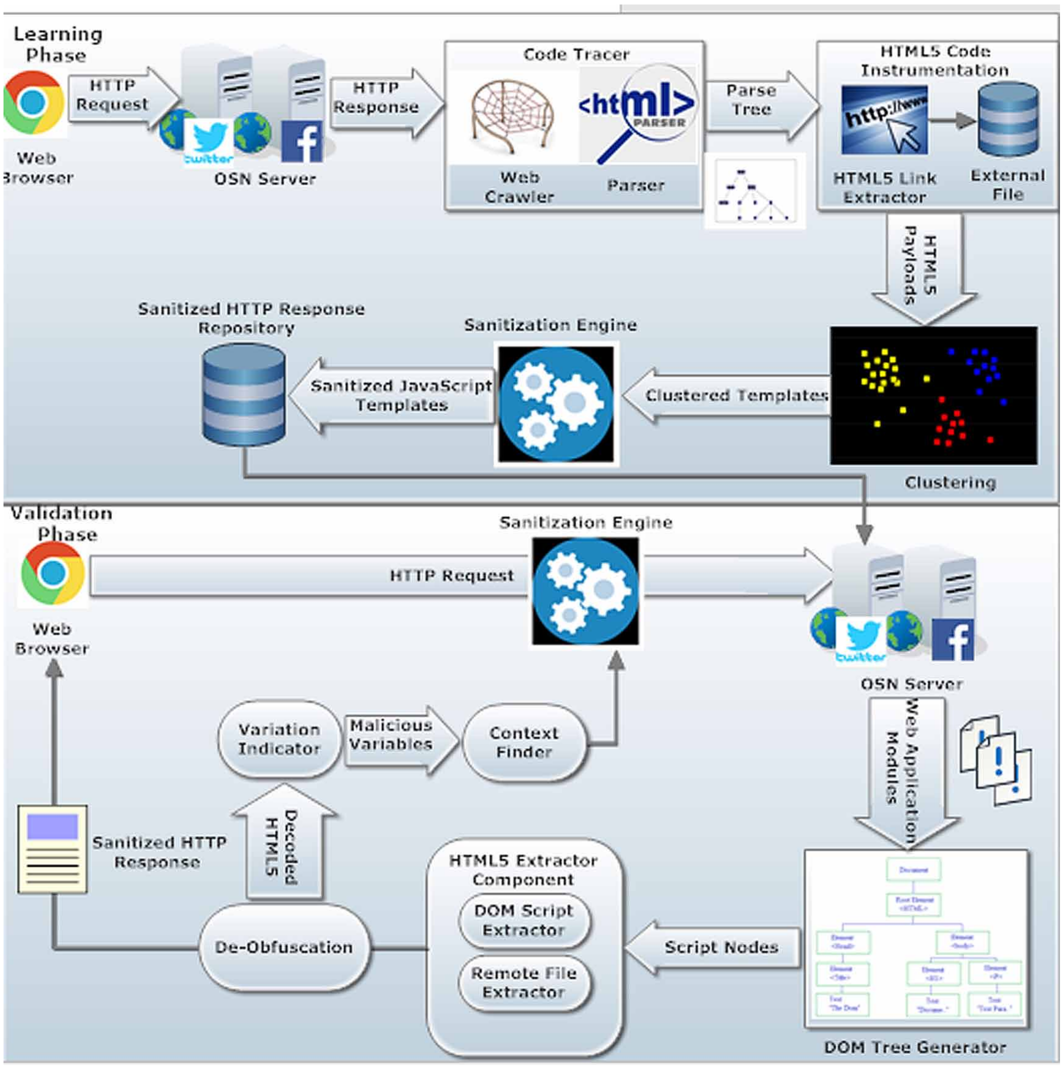


Figure 7. Algorithm for HTML5 code instrumentation

```
Input: Dataflow Graph DFG(N, T)
Output: JavaScript External File
Start
For Each $n_I \in N$
            if ($n_I$ ==<script>)
                while ($n_I$ != </script>)
                        $S.F_I \Leftarrow$ True;
                        $S.buf_I \Leftarrow n_I$.value;
                        $n_I \Leftarrow DFS(n_I)$;
                End while
            ext.file $\Leftarrow$ S.buffer;
          End if
End for each
return ext.file
End
```

is a script node or not. If it is a script node (<script>) then, it set a script flag S.FI and moves entire content into a buffer S.bufI until it traverses to a closing script node (</script>). Flag and buffer are unique to each script node pair in the response. Finally, it shifts the entire content of buffer in the external HTML5 file ext.file and produces this file as its output. In the algorithm shown in the Figure 8, S_File is a log maintained to store all possible recognized HTML5 file. This algorithm processes a data flow graph as follows: We audit each node (n) in the graph (N, T) to check its content. If it is <script> tag, then we extract the content of each node in S_File that appears during the traversal of unique path, until a node with content </script> is found. Finally, it outputs S_File comprising of HTML5 script code that a web page might result.

Clustering

This component implements an algorithm (as shown in figure 9) for clustering the extracted attack vectors payloads depending on their similarity ratio. Consequently, a clustered template is generated that describes the attack vectors in compressed form. Consider the example as shown below:

```
<script>alert(48a$bc);</script>
<script>alert(48xv&ez);</script>
```

These scripts only differ by their argument value. In this view, a compressed template is generated by applying the proposed algorithm as shown in figure 9. A template is a string produced by several types of lexical tokens that are considered to be common for each attack vector payload in a cluster, along with variable portion, represented by the placeholders. N- used as substitute for numbers and S- used as substitute for alphanumeric characters. Thus, template for the above set of scripts is denoted as:

```
<script>alert(48-S-);</script>
```

Figure 8. Algorithm for HTML5 string extraction

```
Input: Control Flow Graph (N, T)
Output: Retrieved HTML5 Script Code.
Start
S_File ← $\phi$;
For Each n ∈ N
    If (n.val!= "<script>") then
        Ignore n;
    Else (n.val == "<script>")
        While ($n_I$.val!= "</script>")
            S_log ← $n_I$.val;
            $n_I$ ← DFS(n);
        End while
    End If
End For
Return S_log
End
```

The input to the algorithm is the WU_{JS} that contains the list of the extracted untrusted HTML5 code. In all iterations, firstly, it compares a pair of attack vectors J_I and J_{I+1} and then uses Levenshtein distance (L_I) to generate the templates. It is defined as the minimum amount of single character deletion, insertion or substitution required to convert one form of string to another. If L_I is less than a selected threshold (β) then, extract the similar character between the pair of attack vectors J_I and $J_{I+1.}$ Non-similar characters are replaced by the placeholders (N/S). Otherwise, pair is discarded and it selects another pair for comparison. Final output is the clustered template as shown by above example. The generated template T is stored in the WC_{JS} for further processing.

Sanitization Engine

Sanitization is a process for substituting the untrusted user variable with the sanitized variable. Clustered scripts templates are sanitized according to the context in which they are used in the HTML5 document. In addition, the same clustering algorithm is applied on the sanitized templates of the malicious XSS attack vectors. Figure 10 describes the proposed algorithm used by the sanitization engine to sanitize the HTML5 string templates. Lastly, the clustered sanitized templates are now saved to the sanitized HTTP response repository, which will be further utilized by the variation indicator for detecting the injection of untrusted HTML5 strings in the DOM tree.

The algorithm shown in Figure 10 executes as follows: San_file includes the sanitizer vector used for sanitization. M_V is an array used to hold the untrusted variables. S_V denote an array used to hold the sanitized variable. CL_file stores the list of the clustered templates and SCL_file is used to store sanitized clustered template. For every template T_P retrieved from CL_file, the algorithm searches for the untrusted variable and stores it in the M_V to determine the context (C_F) and then apply the

Figure 9. Algorithm for clustering of HTML5 attack vectors

```
Input: Untrusted HTML5 Script Code
Output: Clustered Template of Attack Vector Payloads
Threshold ($\beta$):= 0;

Start
WU_JS ← Whitelist of extracted untrusted HTML5 code;
WC_JS ← NULL;
V_I ← 0
For Each untrusted HTML code $J_I \in WU_{JS}$
        Compare($J_I$, $J_{I+1}$);
        $L_I$ ← Levenshtein_distance($J_I$, $J_{I+1}$);
        If ($L_I > \beta$)
            Accept ($J_I$, $J_{I+1}$);
                Generate template $T_P \in (J_I, J_{I+1})$;
                $WC_{JS} \leftarrow T_P \cup WC_{JS}$;
        End If
        Else
            Eliminate ($J_I$, $J_{I+1}$);
            Pick next pair ($J_{I+1}$, $J_{I+2}$);
        End Else
End For Each
Return WC_JS
End
```

sanitizer (S_I) according to the context in which M_V is used. Sanitized variable is stored in S_V and then it is appended to the San_file for more effective result. After sanitization of each template in CL_file, we apply clustering algorithm as shown in Figure 9 to the sanitized template array San_file and store the sanitized clustered template in SCL_file. All the sanitized variables are then injected to the HTML document at their respective locations and modified HTML document is displayed to the user.

VALIDATION PHASE

The key goal of this phase is to detect the injection of untrusted/malicious HTML5 strings in the DOM tree and alleviate the execution of such malicious strings by applying the process of context-aware sanitization on the untrusted variables of such strings. Figure 11 highlights the detailed algorithm of working of this phase. The working of the algorithm is defined as follows: Initially, it generates HTTP response H_{RES} corresponding to each received HTTP request. Then, it constructs DOM tree as R_D and stores all the content between the pair <script>…</script> in scr_log. To detect the XSS attack, it compares both the extractedHTML5 code present in San_file and scr_log. If no variance occurs, then no XSS attack. Otherwise, malicious untrusted variable is extracted 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.

Figure 10. Algorithm for sanitization of clustered templates of HTML5 attack vectors

```
Input: Set of Clustered Template of Attack Vector Payloads
($T_{P1}$, $T_{P2}$,.....$T_{PN}$)
Output: Sanitized Attack Vectors Templates
Start
San_file $\Leftarrow$ Whitelist of available sanitizers routines ($San_1$,
$San_2$,----$San_3$)
CL_file $\Leftarrow$ Whitelist of clustered templates;
SCL_file $\Leftarrow \phi$;
$M_V \Leftarrow \phi$;
$S_V \Leftarrow \phi$;
For Each template $T_P \in$ CL_file
        $M_V \Leftarrow$ malicious variable($T_P$);
          $C_F \Leftarrow$ Context_finder($M_V$);
          $S_I \Leftarrow$ ($S_I \in$ San_file) $\cap$ ($S_I$ matches $C_F$);
          $S_V \Leftarrow S_V(M_V)$;
          San_file $\Leftarrow S_V \cup$ San_file;
End For Each
For Each $S_V \in$ San_file
        SCL_file $\Leftarrow$ Clustering($S_V$);
End For Each
Return SCl_file;
End
```

Key Modules of Validation Phase

All the modules of this phase are deployed on the virtual machines of cloud platforms. The detailed workings of some of the modules of this phase are as follows:

DOM Tree Generator

The DOM is a convention for representing and interacting with objects in HTML, XML documents. The DOM is platform independent and nodes of every document are organized in a tree structure and this type of tree representation of HTML, XML objects are called DOM Trees. The node in a DOM tree can have several child nodes and these nodes are called siblings. The DOM trees can be generated using DOM Tree generator and these trees can be compared to verify the addition of malicious scripts in webpages.

Figure 12 explains the detailed algorithm of DOM tree generation. The working process of the algorithm is described as follows: Initially, it extracts HTTP response web page as W_{PI}. Firstly, it discovers first HTML tag and stores it in AT_I and also pushes it into the stack S_K. Then, AT_I is added to the empty DOM tree as its root node. All the identified tags are stored into the repository HTML_file as well as extended to the Stack. Until stack is empty, it extracts each HTML tag as 'Pos' and check if it is an opening tag or closing tag. If opening tag then, it finds out the size of stack in P and traverses the partially generated DOM tree up to depth level P. When it reaches the last traversed node Q, then it adds the extracted HTML tag. Additionally, it add it to the stack and in to the HTML5_file. Otherwise, it pops out from the stack and returns the root node of the generated DOM tree for further processing to other component.

Figure 11. Algorithm of validation phase

```
Input: Set of extracted HTTP request
Output: Modified sanitized HTTP response
Start
For each $H_{REQ}$
        Generate $H_{RES}$;
        for each $H_{RES}$
                $R_D \Leftarrow$ DOM Tree
Generator($H_{RES}$);
                For each node $N_D \in R_D$
                if ($N_D$ == <script>) then
                   while ($N_D$ != </script>) then
                         scr_log $\Leftarrow R_D$;
                         $R_D \Leftarrow$ DFS($R_D$);
                   End while
               End if
       End for each
       Perform decoding on scr_log
       X $\Leftarrow$ Compare(San_file, scr_log)
       if (X == true) then
          No XSS;
          Return $H_{RES}$;
      Else
         $S_I \Leftarrow$ Untrusted Variable($V \in$ scr_log)
         context $\Leftarrow$ Context_finder($S_I$);
         $H_{RES} \Leftarrow$ Sanitization engine(context);
         Return $H_{RES}$;
     End if
  End for each
End
```

De-Obfuscation

This component performs the decoding of the obfuscated HTML5 code. In order to bypass the XSS filters applied at server and/or client side, attackers, generally, performs the obfuscation on the simple malicious HTML5 code. Therefore, it is necessary to de-obfuscate the injected HTML5 code to clearly identify the malicious elements present in it. Table 1 highlights some of the list of escape codes. 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.

Context Finder

The goal of this component is to identify the context of each type of untrusted variable included in the HTML5 string entered as untrusted input by user. It will reveal the context in which each untrusted variable in HTML5 input will be safely rendered and sanitizers’ functions are selected accordingly.

Figure 13 illustrates the algorithm processed to determine the context. The working of this algorithm is described as follows: Input to the above algorithm is the set of the extracted JavaScript code as SC_file.

Figure 12. Algorithm for DOM tree generation of HTTP response

```
Input: Set of HTTP response
Output: Address of root node of extracted DOM tree
Start
$S_K \leftarrow \phi$;        /* initially stack empty*/
$R \leftarrow \phi$           /* root node of the DOM tree*/
HTML_file $\leftarrow \phi$;
For Each $H_{RES}$
    $WP_I \leftarrow$ web page $\triangleq HR_I$;
    $AT_I \leftarrow$ retrieve_attribute($WP_I$);
    $S_K$.insert($AT_I$);
    Root.insert_node(ATI, NULL);
    HTML_file $\leftarrow AT_I \cup$ HTML_file;
    While ($S_K$!= " ")
        Pos $\leftarrow$ retrieve_attributes($WP_I$);
        If ( initial_tag(Pos)) then
          P $\leftarrow S_K$.size();
          Q $\leftarrow$ DFS(Root, P);
          Root.insert_node(Pos, Q);
          $S_K$.insert(Pos);
          HTML5_file $\leftarrow$ Pos $\cup$ HTML_file;
        Else If ( closing_tag(Pos)) then
          $S_K$.remove();
        End If
    End While
End For Each
Return root
End
```

Table 1. List of escape codes

Display	Hexadecimal Code	Numerical Code
"	&#x22;	&#34;
#	&#x23;	&#35;
&	&#x26;	&#38;
'	&#x27;	&#39;
(	&#x28;	&#40;
)	&#x29;	&#41;
/	&#x2F;	&#47;
;	&#x3B;	&#59;
<	&#x3C;	&#60;
>	&#x3E;	&#62;

Con_ file is a log maintained to store context of each untrusted variable present in the JavaScript code. For each untrusted JavaScript variable M_I, we attach a context finder CF in the form as $CT_I \leftarrow (CF)M_I$. The generated output is the internal representation of the extracted JavaScript code embedded with the context finder CF corresponds to each untrusted variable present in it. After this, it is merged with the Con_file as $Con\_file \leftarrow CT_I \cup$ Con_file. For each $CT_I \in$ Con_file, we generate and solve the type constraints. Here, ϕ represents the type environment that performs the mapping of the JavaScript variable to the Context finder CF. 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 $\phi \mapsto$ e: CF.

It indicates that at any program location, e has context finder CF in the type environment ϕ. Finally, all CT_I variables have been assigned the context dynamically (string, regular expression, numeric etc.) and produce the modified log Con_file as output. This step outputs untrusted variable, present in the extracted HTML5 code, with their context in which browser interprets it.

UNDERSTANDING CONTEXT-SENSITIVE SANITIZATION

In our work, we have described the technique of sanitizing the different categories of HTML5 script code. We have developed some manual functions that are utilized for sanitizing the malicious code of HTML5 in a context-aware manner.

Sanitization of Suspicious Keywords

A doubtful keyword falls in the category of identified malicious data for blocking illegitimate data from execution. Our technique has utilized a fixed template of sanitization which will sanitize the user input in a precise manner. If any user-injected string matches the template of sanitization, then it is recognized as a XSS worm and would be further substituted with resultant keywords. Usually, document.write, parentNode, <embed, document.cookie, <script etc. are considered to be some of the insecure or doubtful keywords. Our technique has statically generated a compiled list of blacklisted keywords and blocked those HTML5 attack vectors whose keywords falls in those blacklisted category. The details of this are as shown in the Figure 14.

Sanitization of Character Escaping

In order to alleviate XSS worms, web application requires confirming that every variable outputs associated with a web page are properly encoded/encrypted before being reverted towards the web browser. Thwarting XSS attack vectors implies to replace all possible exceptional characters utilized in the exploitation of such attacks. Our technique has escaped the malicious characters through utilizing the &# arrangement followed through its code of character. Table 1 highlights some of the list of escape codes. 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 in the form of a portion of a web page, however the web browser could not run the script. Figure 15 illustrates the technique related to filtering character escaping. We have incorporated the feature of list of common escape codes in our technique and thus, our cloud-based framework can stop the web browsers of OSN users from executing such scripts.

Recent work performs the context-aware sanitization on the malicious variables embedded in JavaScript. However, they could not able to determine the nested context of such malicious code. Our wok initially determines the nested context of suspicious variables embedded in HTML5 code and accordingly performs the sanitization on them. This order of context-aware sanitization completely alleviates the DOM-based XSS vulnerabilities from the HTML5 web applications.

Figure 13. Algorithm for determining diverse context of HTML5 code

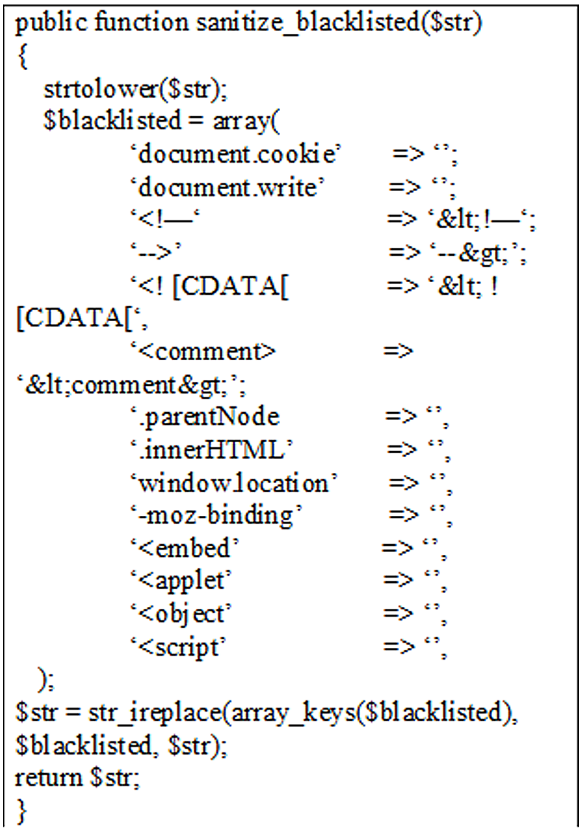
```
Input: Whitelist of decoded malicious HTML5 code
Output: Context of each untrusted variable in HTML5.
Start
Context finder: $CF_1| CF_2|...|CF_N$;
SC_file ← Whitelist of retrieved HTML5 Script code.
Con_file ← $\phi$;          /* list for type qualifier variables*/
For Each retrieved script code $SC_I$
        $M_I$ ←malicious variable $\triangleq SC_I$;
        $CT_I$ ← $CF(M_I)$;
        Con_file ← $CT_I \cup$ Con_file;
End For Each
For Each $CT_I \in$ Con_file
        Parse($SC_I$);
        If ($M_I \in$ String) then
            $\varphi \mapsto CT_I$: String;
        Else if ($M_I \in$ Numeral) then

            $\varphi \mapsto CT_I$: Numeral;
        Else if ($M_I \in$ Regular Expression) then
            $\varphi \mapsto CT_I$: Regular expression;
        Else if ($M_I \in$Literal) then
            $\varphi \mapsto CT_I$: Literal;
        Else if ($M_I \in$ Constant) then
            $\varphi \mapsto CT_I$: Constant;
        End If
End For Each
Con_file ← newvalue($CT_I$);
Return Con_file;
End
```

KEY MERITS

The novelty of our cloud-based framework lies in the fact that it detects the propagation of malicious HTML5-based XSS attack vectors from the vulnerable injection points of HTML5 web applications. Recent XSS defensive techniques usually refer the XSS attack vectors from the XSS cheat sheet (Rsnake, 2008) that does not comprise the HTML5 attack vectors. While evaluating the HTML5 attack vector detection capability of our framework, we inject such attack vectors from the five freely available XSS attack vector repositories. This validates the robustness of our approach. In addition, we have enhanced the context-sensitive sanitization procedure of sanitizing such attack vectors by determining the nested context of the malicious variables embedded in such attack payloads. Existing work failed to determine the nested context of suspicious/untrusted variables encapsulated in HTML5 attack payloads. As a result, false negative rate of such recent approaches was unacceptable.

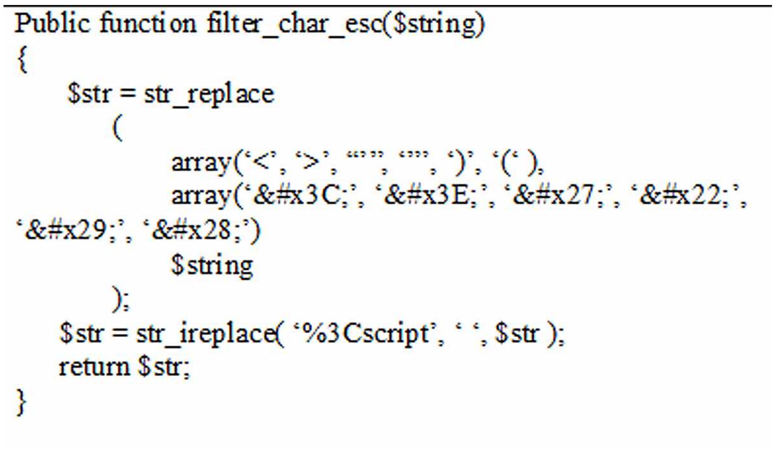
Recent models are very difficult to integrate in the existing infrastructures of cloud settings as they demand lot of modifications in the source code of web applications. Nevertheless, our work is a cloud-based framework that can be easily integrated in the virtual machines of cloud computing

Figure 14. Function for sanitizing insecure keywords



```
public function sanitize_blacklisted($str)
{
   strtolower($str);
   $blacklisted = array(
            ‘document.cookie’     => ‘’;
            ‘document.write’      => ‘’;
            ‘<!—‘                 => ‘&lt;!—‘;
            ‘-->’                 => ‘--&gt;’;
            ‘<! [CDATA[           => ‘&lt; !
[CDATA[‘,
            ‘<comment>           =>
‘&lt;comment&gt;’;
            ‘.parentNode         => ‘’,
            ‘.innerHTML’          => ‘’,
            ‘window.location’     => ‘’,
            ‘-moz-binding’        => ‘’,
            ‘<embed’             => ‘’,
            ‘<applet’            => ‘’,
            ‘<object’            => ‘’,
            ‘<script’            => ‘’,
  );
$str = str_ireplace(array_keys($blacklisted),
$blacklisted, $str);
return $str;
}
```

Figure 15. Function for sanitizing character escaping



```
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;
}
```

environment with little amendments required in the source code of web applications. Moreover, we have also assessed the HTML5 attack vector detection capability of our model in non-cloud environment with very low rate of false positives and false negatives. This validates the scalability of our approach.

Our technique can also survive with application-specific sanitizations. Partial HTML5 attack string identification is capable enough for handling the application-specific sanitizations that might happen. For example, when a '(' character is substituted with &#x28;. On the other hand, a strict and exact identification algorithm (as followed by the existing XSS filters) would fail to resemble even if an individual substitution occurs.

IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We have developed a prototype of our proposed framework in Java via introducing the Java Development Kit and integrate this framework in the virtual machine of cloud server. The experiment background is simulated with the help of a normal desktop system, comprising 2.6 GHz Intel core i3 processor, 4 GB DDR RAM and Windows 10 operating system. The DOM-based XSS worm detection capability of our framework was evaluated on a tested suite of five open source OSN websites deployed in virtual machines of cloud platforms. Table 2 highlights the detailed configuration of these HTML5 web applications.

EXPERIMENTAL EVALUATION

We estimate the testing, accuracy and the performance of our cloud-based proposed framework. In terms of testing, we have tested our cloud-based XSS defensive mechanism on five open-source real world web applications (i.e., Elgg, Humhub, BloggIt, Scarf and OsCommerce). All the platforms of such web applications have numerous injection points that are vulnerable to XSS attacks. In order to include the capabilities of our cloud-based framework in these web applications, developers of these websites have to do less quantity of effort. The motivation to select these five different platforms of web applications is that, we simply need to mark an argument that web sites deployed in the cloud settings can utilize the capabilities of our 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 HTML5 web applications determines the flexible compatibility of our framework. We deployed such web applications on an XAMPP web server with MySQL server as the backend database. We select such applications for accessing the forms to potentially supply modified pages and access the HTML forms. Table 3 highlights the five different categories of the malicious HTML5 context, including Tags, SVG, Attributes, Cascading Style Sheet (CSS) Attack Vectors (CSSAV) and URL Attack Vectors (UAV). The categories of different context of attack vectors are generated using the open source

Table 2. Detailed configuration of HTML5 web applications

Web Applications	Explanation	Lines of Code (K)	Identified Vulnerabilities	Injection Points Instrumented	Version
Elgg	Social Networking Engine	1.3	XSS Attack, Session Hijackig	22	1.12.5
HumHub	Social Intranet Application	2.9	JS Worm Injection	19	0.20
BloggIt	Blog Search Engine	1.8	Authentication Violation	37	1.01
Scarf	Conference Management System	2.6	Authentication Violation	32	2006-09-20
OsCommerce	e-Commerce	2.1	Workflow Violation	17	3.0.2

XSS attack vector repositories. The observed results of our framework on five different platforms of HTML5 web applications deployed in the virtual machines of cloud servers are highlighted in Table 3. We have injected the five different contexts of HTML5 attack vectors on the injection points of these web applications.

We have analyzed the experimental results of our 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 Table 4 that the highest numbers of TPs are observed in BlogIt web application. Figure 16 represents the graphical representation of our observed results.

In addition, the observed rate of false positives and false negatives is acceptable in all the platforms of web applications. We have also calculated the DOM-based XSS attack payload detection rate of our framework on different platforms of 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. Table 4 highlights the detection rate of our cloud-based framework on five different platforms of web applications w.r.t. individual category of context of HTML5 attack vectors. It is clearly reflected from the Table 4, the highest malicious HTML5 detection rate is observed for BlogIt web application.

PERFORMANCE ANALYSIS

This sub-section discusses a detailed validation and performance analysis of our cloud-based framework by conducting two statistical analysis methods (i.e., F-Score and F-test Hypothesis). We have also compared the suspicious HTML5 code detection capability of our proposed framework with other recent 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.

Performance Analysis Using F-Score

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

$$\textit{False Positive Rate (FPR)} = \frac{False\,Positves\,(FP)}{False\,Positives\,(FP) + True\,Negatives\,(TN)}$$

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

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

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

$$F - Score = \frac{2\left(TP\right)}{2\left(TP\right) + FP + FN}$$

Here we calculate the precision, recall and finally F-Score of observed experimental results of our framework on different platforms of HTML5 web applications. See Table 5. F-Score generally analyzes the performance of system by calculating the harmonic mean of precision and recall. The

Table 3. Different Classes of HTML5 Attack Vectors

Class	HTML5 Attack Payload	Description
Tags	<video onerror= “alert(1)”><source></source></video>	Inject the XSS attack payloads into the ‘onerror’ function. During the loading of video, as error occurs, scripts get executed.
	<video src= “#” onerror=alert(1)>	
	<video><source onerror= “alert(1)”>	
	<audio onerror= “alert(1)”><source></source></audio>	Inject the XSS attack payloads into the ‘onerror’ function. During the loading of audio, as error occurs, scripts get executed.
	<audio src= “#” onerror= “alert(1)”></audio>	
	<audio><source onerror= “alert(1)”></audio>	
	<EMBED SRC= “data:image/svg+xml;base64,PHN2ZyB4bWxuczpzdmc9Imh0dHA6Ly93 d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxucz0iaHR0cDovL3d3d y53M5vcmcv cvMjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xo xOTk5L3hsaW5rIiB2ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxO TQiIGhlaWdodD0iMjAwIiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZ WNtYXNjcmlwdCI+YWxlcnQoIlhTUyIpOzwvc2NyaXB0Pjwvc3ZnPg== “type = “image/svg+xml” AllowScriptAccess= “always”></EMBED>	Injecting this attack vector via utilizing SVG graphics and <embed> tag. Finally, encrypt this attack vector by using 64 base encoding schemes. As a result, this attack payload will display a window with message ‘XSS.
SVG	<svg onload= “JavaScript:alert(1)”xmlns= “http://www.w3.org/2000/svg”></svg>	Inject the XSS attack payloads into the onload function under the tag <svg> or <g>.
	<svg xmlns= “http://www.w3.org/2000/svg”> <g onload= “JavaScript:alert(1)”></g></svg>	
	<svg xmlns= “http://www.w3.org/2000/svg”> <a xmlns:xlink= “http://www.w3.org/1999/xlink”xlink:href= “JavaScript:alert(1)”> <rect width= “1000” height= “1000”fill= “white”/></a></svg>	XSS attack payload generates a rectangle with specified lengths and heights. The scripts in the xlink:href can be executed only, when the victim clicks on this rectangle.
Attributes	<body oninput=alert(1)><input autofocus>	Inject the scripts inside the event functions. Scripts only execute due to the inclusion of attribute ‘autofocus’.
	<input onfocus=alert(1) autofocus>	
	<input onblur=alert(1) autofocus><input autofocus>	
	<form><button formaction= “JavaScript:alert(1)”>X</button>	Inject the scripts inside the attribute “formaction”. The scripts would be executed upon clicking of the user.
CSSAV	<LINK REL= “stylesheet” HREF= “http://ha.ckers.org/xss.css”>	Inject the scripts by encapsulating the “stylesheet” attribute in the URL-embedded script.
	<STYLE>li{list-style-image: url(“javascript:alert(‘XSS’)”);} </STYLE><UL><LI>XSS</br>	
	<STYLE>BODY{-moz-binding:url(“http://ha.ckers.org/xssmoz.xml#xss”)} </STYLE>	
	<STYLE>@import’http://ha.ckers.org/xss.css’;</STYLE>	
UAV	<a onmouseover=alert(document.cookie)>xxs link</a>	Inject the scripts embedded in the URL.
	<A HREF= “http://ha.ckers.org@google”>XSS</A>	
	<A HREF= “javascript:document.location=’http://www.google.com/’”>XSS</A>	
	<A HREF= “http://0102.0146.0007.00000223/”>XSS</A>	

analysis conducted reveals that our cloud-based framework exhibits high performance as the observed value of F-Score in five platforms of web applications is greater than 0.9. Table 6 highlights the detailed performance analysis of our proposed framework on five real world web applications. It is clearly reflected from the Table 6 that the performance of our cloud-based framework on five different platforms of web applications is almost 97% as the highest value of F-Score is 0.962. In addition to this, the lowest False Negative Rate (FNR) is observed in OsCommerce. This validates the performance of our framework.

Figure 16. Graphical representation of our observed on HTML5 web applications

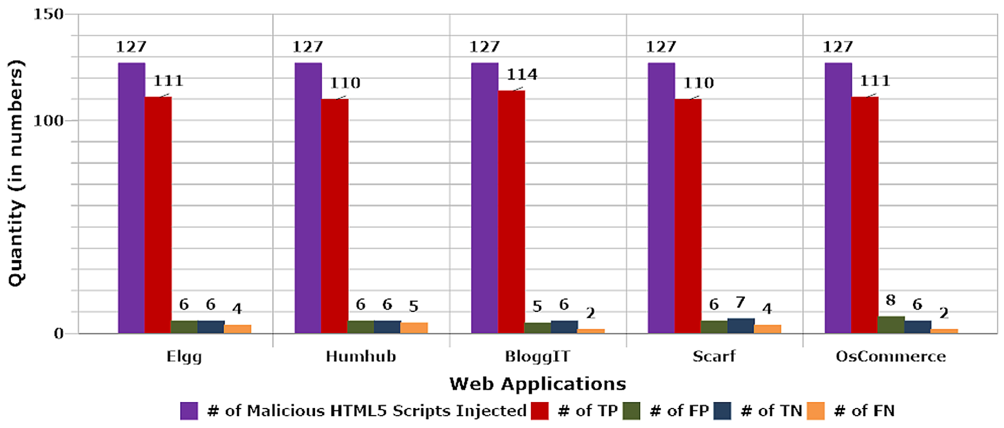


Table 4. Detection rate of suspicious HTML5 code on all platforms of web applications

HTML5 Attack Vector categories	Web applications				
	Elgg	Humhub	BlogIt	Scarf	OsCommerce
Tags	86	89	94	86	91
SVG	80	80	90	80	85
Attributes	87	93	80	80	80
CSSAV	91	82	86	95	86
UAV	91	86	91	89	89

Performance Analysis Using F-Test

In order to prove that the number of malicious HTML5 scripts detected (i.e. number of True Positives (TP)) is less than to the number of HTML5 malicious scripts injected; we use the F-test hypothesis, which is defined as:

Null Hypothesis

H_0 = Number of HTML5 malicious scripts detected is less than the number of HTML5 malicious scripts injected. ($S1^2 = S2^2$)

Alternate Hypothesis

H_1 = Number of HTML5 malicious scripts injected is greater than number of HTML5 malicious scripts detected ($S1^2 < S2^2$)

The level of Significance is $(\alpha = 0.05)$. The detailed analyses of statistics of XSS attack worms applied and detected are illustrated in the Tables 7 and 8. In our work, we utilized and injected total of 127 HTML5 attacks vectors from the freely available XSS attack repositories in all the five web applications. But here note that, for evaluating and validating the performance of our framework by

Table 5. Observed results of our work on HTML5 web applications

Malicious Attack vector Categories	Performance parameters				
	# of Malicious scripts injected	# of TP	# of FP	# of TN	# of FN
Elgg					
Tags	35	30	1	2	2
SVG	20	16	1	2	1
Attributes	15	13	1	0	1
CSSAV	22	20	2	0	0
UAV	35	32	1	2	0
Total	**127**	**111**	**6**	**6**	**4**
HumHub					
Tags	35	31	2	1	1
SVG	20	16	1	2	1
Attributes	15	14	1	0	0
CSSAV	22	19	1	1	1
UAV	35	30	1	2	2
Total	**127**	**110**	**6**	**6**	**5**
BloggIt					
Tags	35	33	1	1	0
SVG	20	18	1	1	0
Attributes	15	12	1	1	1
CSSAV	22	19	1	2	0
UAV	35	32	1	1	1
Total	**127**	**114**	**5**	**6**	**2**
Scarf					
Tags	35	30	2	2	1
SVG	20	16	1	2	1
Attributes	15	12	1	1	1
CSSAV	22	21	0	1	0
UAV	35	31	2	1	1
Total	**127**	**110**	**6**	**7**	**4**
OsCommerce					
Tags	35	32	1	1	1
SVG	20	17	1	2	0
Attributes	15	12	2	1	0
CSSAV	22	19	2	1	0
UAV	35	31	2	1	1
Total	**127**	**111**	**8**	**6**	**2**

Table 6. Performance analysis of our cloud-based framework by calculating F-Score

Non-OSN web application	Total	# of TP	# of FP	# of TN	# of FN	Precision	FPR	FNR	Recall	F-Score
Elgg	127	111	6	6	4	0.948	0.5	0.034	0.965	0.962
HumHub	127	110	6	6	5	0.948	0.5	0.043	0.956	0.958
BloggIt	127	114	5	6	2	0.931	0.533	0.035	0.964	0.954
Scarf	127	110	6	7	4	0.948	0.6	0.035	0.964	0.962
OsCommerce	127	111	8	6	2	0.932	0.8	0.017	0.982	0.962

using F-test, we injected different number of HTML5 malicious scripts in all different platforms of 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.
- $F_{CALC} = S_1^2 / S_2^2 = 0.774$
- The tabulated value of F-Test at $df_1 = 4$, $df_2 = 4$ and $\alpha = 0.05$ is
- $F_{(df1,df2,1-\alpha)} = F_{(9,9,0.95)} = 3.1789$
- 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.

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 HTML5 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 9 highlights the response time of our proposed framework in different environment (i.e. without cloud infrastructure) and cloud platform.

On the other hand, we have also compared the performance analysis of existing client-side XSS filter (i.e. XSSFilt (Pelizzi et al. (2012)) with our cloud-based framework. XSSFilt is installed as an extension of Firefox web browser. Here also, we have verified the malicious HTML5 detection capability of XSS-Auditor by injecting 127 XSS worms on our five HTML5 web applications. Table 10 highlights the performance comparison of our cloud-based framework with existing client-side XSS filter (XSSFilt). It can be clearly observed from the Table 10 that the value of F-Score is decreasing in all the platforms of OSN-based web applications for XSSFilt in comparison to our work.

Although, XSSFilt performs the sanitization on the untrusted variables of JavaScript/HTML5 attack payloads in a context-aware manner, yet this filter is not capable enough to determine all

Table 7. Statistical analysis of malicious scripts injected

Web Applications	# of Malicious Scripts Injected (X_i)	(X_i - μ)	(X_i - μ)^2	Standard Deviation $S_1 = \sqrt{\sum_{i=1}^{N1} (Xi-\mu)^2 / (N1-1)}$
Elgg	120	-1	1	2.449
HumHub	124	3	9	
BloggIt	122	1	1	
Scarf	118	-3	9	
OsCommerce	119	-2	4	
	Mean (μ) = $\sum Xi / N1$ = 121		$\sum_{i=1}^{N1} (Xi-\mu)^2$ = 24	

Table 8. Statistical analysis of malicious scripts detected

Web Applications	# of Malicious Scripts Detected (X_i)	(X_i - μ)	(X_i - μ)^2	Standard Deviation $S_2 = \sqrt{\sum_{i=1}^{N1} (Xi-\mu)^2 / (N2-1)}$
Elgg	115	0	0	2.783
HumHub	118	3	9	
BloggIt	118	3	9	
Scarf	112	-3	9	
OsCommerce	113	-2	4	
	Mean (μ) =2 $\sum Xi / N2$ = 115		$\sum_{i=1}^{N1} (Xi-\mu)^2$ = 31	

possible contexts of such untrusted variables. Therefore, the sanitization of such variables in such malicious variables becomes ineffective for the XSSFilt. However, our cloud-based framework is capable enough to determine the probable different context of malicious variables of JavaScript prior to the execution of context-sensitive sanitization procedure. The proposed framework is a purely cloud-based XSS defensive framework that detects and alleviates the DOM-based XSS worms from the HTML5 web applications deployed in the cloud computing environment. We tested the response time of XSS attack detection capability and sanitization performance on the cloud platforms as well as in other environments. The response time observed was low for all the five platforms of HTML5 web applications deployed on the cloud settings in comparison to other settings of infrastructure.

Table 9. Performance analysis using response time calculation

Non-OSN Platforms	Response Time (in ms)	
	Without Cloud Platform	On Cloud Platform
Elgg	2096	2018
HumHub	2312	2216
BloggIt	2789	2676
Scarf	3126	2986
OsCommerce	2753	2697

Table 10. Performance comparison of XSSFilt with our work

Web Application	# of TP	# of FP	# of TN	# of FN	Precision	Recall	F-Score
Cloud-Based							
Elgg	116	5	4	2	0.958	0.983	0.970 ↑
HumHub	115	5	5	2	0.958	0.982	0.970 ↑
BloggIt	112	7	6	2	0.941	0.982	0.961 ↑
Scarf	114	6	5	2	0.950	0.982	0.967 ↑
OsCommerce	117	6	3	1	0.951	0.991	0.970 ↑
XSSFilt (Pelizzzi et al. (2012))							
Elgg	102	10	11	4	0.910	0.962	0.935 ↓
HumHub	109	6	6	6	0.947	0.947	0.947 ↓
BloggIt	105	7	8	7	0.937	0.937	0.937 ↓
Scarf	97	12	10	8	0.889	0.923	0.906 ↓
OsCommerce	101	9	8	10	0.918	0.909	0.914 ↓

CONCLUSION AND FUTURE WORK

In this paper, we proposed a DOM-based XSS defensive framework for the HTML5 web applications deployed in the infrastructure of cloud. The technique enhances the context-aware sanitization procedure for malicious HTML5 attack vectors employed in recent work. The framework is dual-based: Initially, it executes in learning phase for the extraction of malicious HTML5 strings and subsequently generates the clustered strings corresponding to their level of similarity. Such templates will undergo through the process of sanitization in learning phase and save the snapshot of such sanitized templates for further reference in the validation phase. The key goal of the validation phase is to detect the injection of untrusted HTML5 strings in the DOM tree by observing the variation in set of set of scripts embedded in the HTTP responses of both these phases. Finally, the context of malicious variables injected in such strings will be identified and consequently alleviates the effect of such malicious strings by executing the process of context-aware sanitization at runtime. The novelty of our framework lies in the fact that it reduces and optimizes the process of runtime context-sensitive sanitization by generating the clustered templates of malicious JavaScript strings. Experimental testing of our cloud-based framework revealed that our work detects the injection of malicious HTML5 attack payloads in the DOM tree of HTTP response with low and tolerable rate of false positives and false negatives in comparison to existent XSS filters. We will try to evaluate the performance of our cloud-based framework on more tested suite of real world HTML5 web applications as a part of future work.

ACKNOWLEDGMENT

The authors would like to thank the 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 with the economic aid provided by the World Bank.

REFERENCES

Aljawarneh, S. (2011). A web engineering security methodology for e-learning systems. *Network Security*, *2011*(3), 12–15. doi:10.1016/S1353-4858(11)70026-5

Aljawarneh, S. (2011). Cloud Security Engineering: Avoiding Security Threats the Right Way. *International Journal of Cloud Applications and Computing*, *1*(2), 64–70. doi:10.4018/ijcac.2011040105

Aljawarneh, S. A., Moftah, R. A., & Maatuk, A. M. (2016). Investigations of automatic methods for detecting the polymorphic worms signatures. *Future Generation Computer Systems*, *60*, 67–77. doi:10.1016/j.future.2016.01.020

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., & Vigna, G. (2008, May). Saner: Composing static and dynamic analysis to validate sanitization in web applications. *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (SP '08) (pp. 387-401). IEEE.

Bates, D., Barth, A., & Jackson, C. (2010, April). Regular expressions considered harmful in client-side XSS filters.*Proceedings of the 19th international conference on World wide web* (pp. 91-100). ACM. doi:10.1145/1772690.1772701

Cao, Y., Yegneswaran, V., Porras, P. A., & Chen, Y. (2012, February). PathCutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks. Proceedings of NDSS.

Chandra, V. S., & Selvakumar, S. (2011). BIXSAN: Browser independent XSS sanitizer for prevention of XSS attacks. *Software Engineering Notes*, *36*(5), 1–7. doi:10.1145/1968587.1968603

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

Dinh, H. T., Lee, C., Niyato, D., & Wang, P. (2013). A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, *13*(18), 1587-1611.

Doupé, A., Cui, W., Jakubowski, M. H., Peinado, M., Kruegel, C., & Vigna, G. (2013, November). deDacota: toward preventing server-side XSS via automatic code and data separation. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (pp. 1205-1216). ACM.

Duchene, F., Rawat, S., Richier, J. L., & Groz, R. (2014, March). KameleonFuzz: evolutionary fuzzing for black-box XSS detection.*Proceedings of the 4th ACM conference on Data and application security and privacy* (pp. 37-48). ACM. doi:10.1145/2557547.2557550

Giorgio Maone. NoScript. Retrieved from: http://www.noscript.net

Gupta, B. B., Gupta, S., Gangwar, S., Kumar, M., & Meena, P. K. (2015). Cross-site scripting (XSS) abuse and defense: Exploitation on several testing bed environments and its defense. *Journal of Information Privacy and Security*, *11*(2), 118–136. doi:10.1080/15536548.2015.1044865

Gupta, S., & Gupta, B. B. (2014). BDS: browser dependent XSS sanitizer. Book on Cloud-Based Databases with Biometric Applications. In *Handbook of Research on Securing Cloud-Based Databases with Biometric Applications* (pp. 174-191). Hershey, PA: IGI Global.

Gupta, S., & Gupta, B. B. (2015). Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, *2015*, 1-19.

Gupta, S., & Gupta, B. B. (2015, May). PHP-sensor: a prototype method to discover workflow violation and XSS vulnerabilities in PHP web applications.*Proceedings of the 12th ACM International Conference on Computing Frontiers* (p. 59). ACM. doi:10.1145/2742854.2745719

Gupta, S., & Gupta, B. B. (2016). *JS-SAN: defense mechanism for HTML5-based web applications against javascript code injection vulnerabilities*. Security and Communication Networks.

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

Gupta, S., & Gupta, B. B. (2016c). XSS-secure as a service for the platforms of online social network-based multimedia web applications in cloud. *Multimedia Tools and Applications*. doi:10.1007/s11042-016-3735-1

Lekies, S., Stock, B., & Johns, M. (2013, November). 25 million flows later: large-scale detection of DOM-based XSS. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (pp. 1193-1204). ACM.

Modi, C., Patel, D., Borisaniya, B., Patel, A., & Rajarajan, M. (2013). A survey on security issues and solutions at different layers of Cloud computing. *The Journal of Supercomputing*, *63*(2), 561–592. doi:10.1007/s11227-012-0831-5

Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A., & Saxena, P. (2015, August). DexterJS: robust testing platform for DOM-based XSS vulnerabilities.*Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 946-949). ACM. doi:10.1145/2786805.2803191

Pelizzi, R., & Sekar, R. (2012, May). Protection, usability and improvements in reflected XSS filters. Proceedings of ASIACCS (p. 5). doi:10.1145/2414456.2414458

Reith, J. (2008). Internals of noXSS. Retrieved from http://www.noxss.org/wiki/Internals

Rsnake. XSS Cheat Sheet. (n. d.). Retrieved from http://ha.ckers.org/xss.html, 2008

Shahriar, H., & Zulkernine, M. (2011, July). Injecting comments to detect JavaScript code injection attacks. Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW) (pp. 104-109). IEEE. doi:10.1109/COMPSACW.2011.27

Sun, F., Xu, L., & Su, Z. (2009, September). Client-side detection of XSS worms by monitoring payload propagation. *Proceedings of theEuropean Symposium on Research in Computer Security* (pp. 539-554). Springer Berlin Heidelberg. doi:10.1007/978-3-642-04444-1_33

Xiao, W., Sun, J., Chen, H., & Xu, X. (2014, July). Preventing Client Side XSS with Rewrite Based Dynamic Information Flow. *Proceedings of the2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming* (pp. 238-243). IEEE. doi:10.1109/PAAP.2014.10

APPENDIX

While evaluating the XSS attack detection capability of our work, we also explored and injected the malicious script code available in the open source XSS attack vector repositories apart from default XSS cheat sheet (RSnake, 2008). See Table 11.

Table 11. List of XSS attack vector repositories

S. No.	XSS Attack Vector Repositories	Reference
1.	HTML5 Security Cheat Sheet	http://html5sec.org/
2.	523 XSS vectors available	http://xss2.technomancie.net/vectors/
3.	Technical Attack Sheet for Cross Site Penetration Tests	http://www.vulnerability-lab.com/resources/documents/531.txt
4.	@XSS Vector Twitter Account	https://twitter.com/XSSVector