Defeating Cyber Attacks Due to Script Injection

Debasish Das, Dhruba Kumar Bhattacharyya
(Corresponding author: Debasish Das)

The Department of Computer Science and Engineering, Tezpur University[1]
Napaam, Sonitpur, Assam 784028, India
(Email: ddas@tezu.ernet.in)

(Received July 15, 2016; revised and accepted Mar. 11, 2017)

Abstract

Offensive operations have been promoted by the aggressors using computer as a tool or target, resulting, a cyber attack in *web-application*s of an organization or the infrastructure of entire nation. Depending upon the attacker's target, one can classify some of the mostly occurred cyber attacks into five broad categories. It reports some of the common methods adopted in conducting these attacks and their defending techniques. This paper mainly address the possibility of cyber attacks due to the execution of malicious or unintended nature of scripts. It formulates a verification method of web document and perform experiment in the client-side using its benign *script* structure. This method is capable of detecting any malicious script which inserts in the web-document during transportation from server to the client or due to the previously stored content in the client or server operation. Satisfactory results have been found with the own-generated and publicly available data-set.

Keywords: Benign Logical Structure; Classification; Cyber Attack; Malicious Script Insertion; Web-Application

1 Introduction

Today's literate population is becoming totally dependent on accessing web applications using browser and performing activities such as - email, banking, domestic appliances etc. The advancement of web technology helps software developer in developing user friendly applications so that user can work with those applications easily. A large number of organizational documents and non-web based applications being transferred into web based applications so that the efficiency and effectiveness of accessing organizational data over the Internet can be improved. Using every latest technology, in one way software developers are increasing simplicity in working with these applications, other way, these applications becoming targets of intruders or malicious users.

A major security issue arises specially during accessing such applications, as a large number of attacks are possible if the vulnerability in those applications are not properly addressed. Attackers, intrude logically into an application by misusing those vulnerabilities and they try to disrupt its normal functioning. Researchers, have been working largely to address web security threats and they could achieve partially to address the issues. Intruders also get cleverer with time and they formulate newer ways of attack which helps in bypassing the security mechanism of an application. To formulate a concrete solution, researchers need more work on every possibility that may lead a successful attack. In this paper first, it presents various categories of cyber attacks and present practices of their defense. Second, it formulates and effective verification of web document. The method detects malicious scripts contained in a response page or web document, which are responsible for defacing user access.

Finally, it reports test results based on experimental evaluation. Satisfactory results have been found for a series of evaluation using the own-generated and publicly available data-set.

2 Security Risks in Accessing Web Applications

In a typical scenario, accessing an application software using web-browser over the *Internet* is shown in Figure 1. The sequence of user request for a static resource (e.g., organization's web-site or web-page), stored in the web server and its response is shown in Lines 1, 2, 3, 4.

However, user may requests for the processed details e.g., students' *grade card* containing the information about grades secured on various subjects and the total grade point earned (both cumulative and semester).

It can be retrieved by accessing application page stored in the *application server* followed by the database processing with the parameter values - *username*, *password*, student's *roll number* and *term* (*say* Spring/Autumn, Year). The response page or web-document generated by the application may contain some client-sided *script*(s). In the client-side, the *script*s perform validation on the user input data and send it to the originating server. The appli-

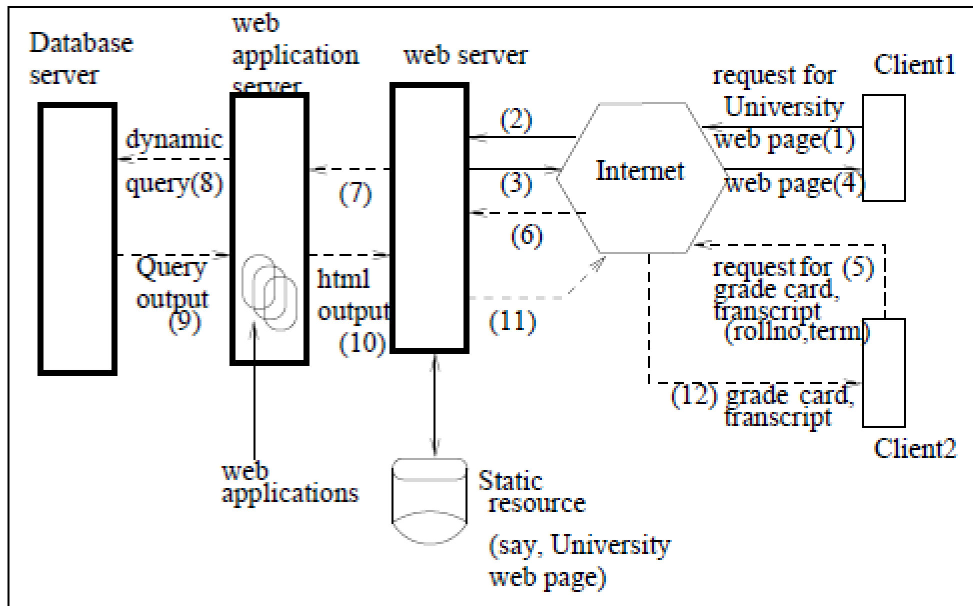

Figure 1: Typical Web application access

cation performs server-side processing using the parameter values, generates reports or web document (grade-card) and send it to the client over the Internet.

User requests for *grade-card* may come randomly to the application server over the *Internet*. For each request, the response page from the server generates one dynamic script. In the client, such scripts perform validation on user input - rollno, term_year and term_type (in Figure 1, dotted lines are shown - 5, 6, 7, 8, 9, 10, 11, 12). When such validation performs in the server, based on the generated scripts, it is called server side validation. The effectiveness of input validation can be increased by performing such at the both end. In Figure 2, it is shown how a legitimate access can be defaced due to the insertion of attack payload so that the legitimate access may re-direct to the attacker's site. Defacing user access mainly occurs due to the change in logical structure of a web-document by inserting the malicious script into it. During legitimate access the input parameter values or other session details such as - *user name*, *password* may be redirected to the attacker's site so that output of such application (i.e., grade-card) can also be redirected from the application server to the attacker's site without user's knowledge.

2.1 Contribution

We introduce a method of detecting *script vulnerabilities* causing cyber attacks based on verifying web document using its benign structure. We report various classes of cyber attacks and the identify the harmfulness of those. The deficiencies of existing detection techniques and the present day importance in addressing scripting vulnerabilities are highlighted. The effectiveness of the proposed method in real-life applications is also reported.

2.2 Cyber Attacks and Their Categories

Cyber attack may be defined as an offensive exercise employed by individuals or group that targets - information systems, infrastructures, computer networks, and/or personal computer devices. It generally, originate from an anonymous source and its main objective is to steal, alter, or destroy to a specified target. Based on their targets cyber attacks can be classified into five different categories described in Table 1.

In EPW, attackers mislead network users with false information by masquerading as a trustworthy entity. They mostly, misguide users through telephonic communication or pull to surf some decorated web sites loaded with false information. By surfing such or responding to their telephonic call, user may fall into the attacker's trap. In UKANF category, attacker tries to gather information on an application network by going through the organization's web pages or by executing some network commands that are commonly used for legitimate access. Attackers misuse vulnerability in a *web-application* under the MAV category and maliciously enter into the system so that they try to execute some unintended scripts or queries resulting undue knowledge of database, data-store, session details etc.

By running a successful MAV attack, attackers can manage to change the integrity of a data-store. Under MSVA category of attack, attacker tries to make the computer system or browser vulnerable by maliciously executing malware or virus or worms. By doing so, attackers try to redirect user access into a malicious *web-application* site without his/her knowledge. The DDoS category of attacks mainly targets network or computer resources. It generally executes by flooding with auto-generated re-

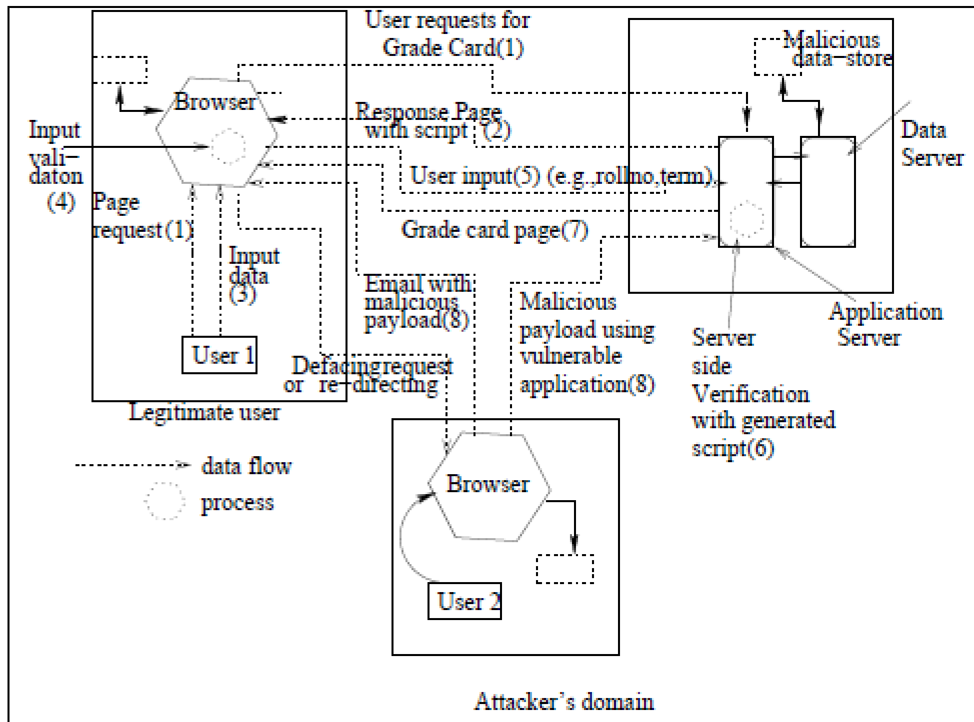

Figure 2: Typical attack scenario

quests through botnet or some network configured specially for this purpose. To amplify the power of such attack, it generally implements in a distributed way with compromised hosts. Denial of service occurs to the legitimate users when congestion arise in the network bandwidth or the bandwidth of targeted web server's - CPU or disk or database.

2.3 Related Work

Under MAV category of cyber attack where, an attacker attempts to bypass authentication mechanism or steal session details without the knowledge of legitimate users. Insertion of unintended or malicious script in a web document during the execution of user access, is an effective method deploying by the attackers. Several solutions have been proposed by the researchers to address this menace so that the scripts execution is restricted in both client and server-side application. However, such restriction may effect application's functionality. Attack *script*s are prepared and injected into web document retrieves session details or deface user access or to perform other malicious operations. To overcome the scripting vulnerabilities, some of the defending coding practices are reported in [5, 18, 19]. Input validity checking for possible scripting vulnerabilities using predefined constraint, such as - black list, white list and syntax grammar, are popularly applied my most of the researchers [3, 17, 21, 23, 26].

The method of identifying vulnerabilities in a *web-application* and deploy defensive coding practices is proposed by some of the researchers [8, 12, 16]. Defensive coding approach is the primary requirement in preventing scripting vulnerabilities. However, dynamic *script* validation is equally important to prevent scripting attacks. To defend from typical scripting attacks, researchers proposed methods which avoid the process of escaping characters during execution of a web document [5, 18, 19].

In spite of implementing the defensive coding approach by the researchers it is found that the malicious *script*s execution comes to be the biggest threat in cyber world. Executing such *script* attackers may success in - defacing user access, stealing session details, changing the logical structure of web document, storing malicious strings in the machine etc. As a result, attackers can proceed for targeting computers or users in the social network.

The property of dynamic script generation in a *web-application* domain can be misused by the attackers, leading to scripting attacks. The current detection techniques [6, 9, 24] attempted to identify the vulnerable scripts in a web document that are dynamically generated. In a *web-application* domain, the method of bounding the script execution within the legitimate frame-work is proposed by some of the researchers [2, 10, 15]. However, such constraint may restricts application's functionality. Parse tree based verification technique to identify deviation between request and response is carried out by some of the researchers [1, 25]. However, due to the modified and enhanced crafting mechanism of *script* injection adopted by the attackers, it needs continuous effort for upgraded solution.

Table 1: Detection approaches and their pros and cons

Attack Class	Attack Method	Common Attacks	Defense
1. Exploit Psycho Weakness (EPW)	Exploring with pretext	Access, Phishing, Porno	Password restriction in login, filtering
2. Undue Knowledge of application & Network Finger-Print (UKANF)	Unintended actions - exploring networks, security scans and network audit	Reconnaissance, port scan, trust, MANNET passive, storage	Data encrypt, Rule based defense
3. Misuse Application's Vulnerability (MAV)	Insert/Execute malicious payload	Hacking, SQL injection, Cross-site scripting, Forgery, MANNET Active	Filtering input data, Firewall/rule-based protection, web-document verification
4. Makes System Vulnerable to Attacks (MSVA)	Tools: Crack-Security Sniffer, malware through stealth viruses, Trojan, Piggy-back	Spam, Hacking espionage, Cyber terrorism Cyber war, black hole, MANNET Active	email filter, Dynamic Firewall or rule-based protection
5. Distributed Denial of Service (DDoS)	Distributed flooding to targeted network or device using bot traffic	DDoS, Application layer DDoS	Deploy IDS, IPS at strategic point

Authentication mechanism is explored to identify vulnerabilities and formulated enhanced technique of authentication based on personalized policies, maintained by the application server [13, 27, 31]. To avoid unauthorized access, researchers have proposed authority-based delegation of encryption mechanism for multi-proxy signature scheme [7, 14, 20]. Proxy-based delegation mechanism for signature based verification is widely explored by the researchers to protect from unintended script execution [28, 29, 30].

3 Motivation

Insertion of malicious *script*s in a web document called *script* injection, may cause some notorious *web-application* attacks, such as - cross-site scripting(XSS), cross-site request forgery etc, To illustrate *script* injection, let us consider the typical *web application* access scenario as shown in Figure 1. The application is a simple on-line *student information system* (SIS) that allows to view academic details of students' admitted in various programme.

In a typical application report, student retrieves his/her grade-card containing information on various subjects in which he/she has registered, grade secured, grade point earned, semester, to credit completed etc. To view such details, user must log on to the proxy server (LINUX O.S.) configured in our University intranet (www.tezu.ernet.in). After successful login, it disconnects from such server and redirect the connection to the actual application server where SIS package (developed with PHP) exist. User must enter his/her valid login details (username and password) in the main page of this package. Login verification is done by a function module developed using the frame-work called *PHP Telnet*. The module retrieves password *script* file of the proxy server, containing account details and verify accordingly.

After successful login, the application generates a response page containing client-sided scripts and send the page to the client machine. The script verify user inputs - roll number or name or registration number and term (spring or autumn) so that by verifying user input with client-sided scripts, these are send to the server. Server-side application processes with these parameter values and generates response page or grade-card. During the process of user access if any unintended script containing some Java functions embedded in the web document, it will also execute. This unintended script may generate due to the previously store malicious content in the client or server side. It can also generate during communication with the previously stored malicious strings. It can be possible if user is not careful in clicking an application or user does not have awareness in accessing cyber-world. Due to the insertion of malicious payload or unintended client-sided script, user access to the application may be defaced because of the followings:

1) Change in logical structure of the web document;
2) Redirect the displayed page to the malicious site;
3) Session details stored in the cookies, send to the malicious site;
4) Malicious data-store.

The logical structure of accessing a web-document or document object model (DOM) defines how it is accessed and manipulated. DOM is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a web document. Let us consider a client-sided script file containing validity function to verify the input

value of student's roll number as given in Figure 3. In this function, it checks whether the length of input string for rollno is equal to 8 or not. If so, the string value is posted to the server in which the server page *abcd.phd* executes by taking this value as argument.

```
function rollnoFunction(){
var currentXhr=null; //Global variable is declared
$("#roll_no").keyup(function()){
var string = $(this).val();
//input value is stored in string variable
var word = string.toUpperCase(); //case conversion
if (word.length=1) {
currentXhr=$.ajax( {
type: "POST",
url: "abcd.php",
data:'keyword='+word,
beforeSend: function(){
$("#roll_no").css("background","#FFF url(Circle.gif)
no-repeat 380px");
if(currentXhr != null && currentXhr.readyState!=4) {
currentXhr.abort();}},
success: function(data){
$("#suggesstion-1").show();
$("#myModal").modal("show");
$("#suggesstion-1").html(data);
$("#roll_no").css("background","#FFF"); }) } }
else
$("#suggesstion-1").hide();
$("#myModal").modal("hide"); }) }
```

Figure 3: Input string (Rollno) validation

Due to the insertion of malicious payload or unintended functional statement (say, Javascript function in a typical application) in the script file of web document, user access may executes in an unexpected manner. The displayed page may redirect to the malicious site and it may post some unintended script function injected into the web document. If the malicious payload containing function as given in Figure 4 say, *setCookie* (function to store user name in cookie variable) and *getCookie* (function to return the value of specified cookie).

There may occur some unintended action of sending information from client to the attacker's site. This can be done by executing the script code containing *url* of attacker's site concatenated with the string as - http://www.xul.fr/vulnera ble.html?cookie="cookie_name"&cookie_valuse=name.

To prevent such kind of unintended execution, filtering of *script* from a web document is the common practices based on black list. Some of the defense method also restricts *script* in a web document. However, filtering mechanism or such restrictions may effect functionality of an application. In a *script* functional statements are mainly applied for client sided validation. Intended *script* functions available in a web document may be poisoned

```
function setCookie(cookie_name,cookie_val,exp_days)
{
var d = new Date();
d.setTime(d.getTime() + (exp_days*24*60*60*1000));
var expires = "expires="+d.toUTCString();
document.cookie = cookie_name + "=" + cookie_val
+ "; " + expires;
}
function getCookie(cookie_name) {
var name = cookie_name + "=";
var ca = document.cookie.split(';');
for(var i = 0; i < ca.length; i++) {
var c = ca[i];
while (c.charAt(0) == ' ') {
c = c.substring(1);
}
if (c.indexOf(name) == 0) {
return c.substring(name.length, c.length);
}
}
return "";
}
```

Figure 4: Unintended script to set and return cookie value

due to the unintended action. Thus, before executing script functions of a web document it must be verified based on legitimate structure.

4 Proposed Approach

A response page or web document send by the *web-application* server against user request is associated with its *benign script structure* called *script proof*. The *script proof* is attached with the page so that before browsing a web document, its structure is verified. The detection approach is implemented in co-ordination with application server and client's web browser. A program module retrieves structure of various *scripts* available in a web document or response page and embed into it before sending to the client's end. Before browsing a web document in the client machine, a plugged-in detection module retrieves - (1) its structure (excluding *script proof* part) called *candidate proof* and (2) the attached *script proof*. The module finally verify the two proofs so that if deviation between them is found, it is considered as an unsafe or vulnerable execution that may cause a scripting attack. The proposed idea is to dynamically construct the *script proof* whenever the application program constructs the response page against a web request. A *script proof* of a response page is considered to be self-evidently non-attacking so that any new insertion of *script* during transportation or due to the previously stored malicious content, a web document can be verified with its *script proof*.

4.1 Problem Formulations

Accessing a *web-application* is comprised of the following steps relevant to scripting attacks:

1) It generates dynamic *script*s based on user request;

2) The generated *script*s are embedded into the response page;

3) User inputs are verified based on the instructions on the *script*s;

4) While browsing a web document or response page, it does not verify the nature of *scripts/scripts function* - intended or malicious;

5) Both server and the client do not maintain the details of the scripts embedded into a response page against an user request;

6) A malicious script may inject into a response page due to the previously stored to the careless attempt by the user;

7) The logical structure of the response page or web document may be effected due to the insertion of malicious scripts stored in the client's machine;

8) Browsing of web document may be defaced due to the non-persistent strings other than as mentioned in Items 6) and 7);

4.2 Definitions

Web-application: A *web-application* is comprised of a set of *web-document*s written in web language such as - *html*, ASP, JSP, API, PHP etc. Each *web-document* can be expressed as an ordered set of - scripts, *html* strings and input-output data. A *web-document*(F_w) maps one or more out-going run-time expressions(Ri_{exp}).

A *web-document* can be expressed as -

$$W_d : \quad (\{h_1, h_2, h_3, \cdots, h_m\} \cup \{s_1, s_2, s_3, \cdots, s_k\} \cup \{d_1, d_2, \cdots, d_n\}),$$

where, $\{s_1, s_2, \cdots, s_k\} \leftarrow$ set of all scripts;

$\{h_1, h_2, h_3, \cdots, h_m\} \leftarrow$ set of all *html* string;

$\{d_1, d_2, \cdots, d_n\} \leftarrow$ input(1) or output(o) data (user input data or data generated by the web-application server);

The mapping of a server-sided *web-document*s of a *web-application*(F_w) into a response page (R_p) can be expressed as -

$$\begin{aligned} R_p &= \langle H \cup S \cup D \rangle \\ &= \langle \{h_1, h_2, \cdots, h_k\} \cup \{s_1, s_2, s_3, \cdots, s_m\} \\ &\quad \cup \{o_1, o_2, \cdots, o_n\} \rangle; \end{aligned}$$

Legitimate web-document:
A web-document (W_d) is defined as *legitimate* or *normal* if its *candidate proof* match with the *script proof*. The function $f_b(S)$ retrieves the *script proof* (B_S) of *web-document* W_d containing scripts $S = \{s_1, s_2, s_3, \cdots, s_k\}$. $f_b(S) = B_S$, embed into the web document or response page (R_p) from the server so that the final web document passed from server to the client, can be expressed as: $R_p^{'} = \langle R_p, B_S \rangle$. The function f_c of the detection module retrieves *candidate proof* C_S from R_p containing the *script*s, $S^{'} = \{s_1^{'}, s_2^{'}, s_3^{'}, \cdots, s_k^{'}\}$, $f_b(S^{'}) = B_S$. Thus, a test web document is legitimate if $C_S = B_S$.

Malicious or attack web-document:
A web-document (W_d) is defined as *malicious* or *attack* if its benign structure or *script proof* does not match with its *candidate proof*, i.e., $C_S \neq B_s(R_p)$.

The *benign structure* or *script proof* of a *web-document* is the sequence of its scripts structure. The structure of a *script* is the sequence of its *functional statement*s or *web language function*s, e.g., the benign structure of the function *getCookie(cookie_name)* is *document.cookie.split,return* as shown in Figure 4. In the application server, an auto-generated module generates the *script proof* of a response page or *web-document* before sending it to the client. In the client, before browsing a *web-document* using client's *web-browser* its structure is verified with its *script proof* using detection module attached to the web browser.

4.3 Detection Algorithm

The detection of nature of a *web-document* as *normal* or *malicious* is done using two different algorithms. First algorithm *web-document-structure*, return structure of a *web-document* by retrieving each script structure. The input to this algorithm is - *web-document* and list of web language functions. The output of this algorithm is the *script* structure of web document, containing one or more sub-strings separated with coma(,). Each sub-string is a web-language function. The second algorithm *exact matching*, retrieves the embedded sequence from the *web-document* called *script proof* and match it with *candidate proof* of the web document. The input to this algorithm is the response page or test *web-document*. The algorithm performs exact matching between the two strings, so that if it is found matched it gives output as *normal*, otherwise, declare as *attack*.

5 Experimental Results

To experiment the proposed detection approach, a three-tire platform is configured having database server at the back-end and web-application server in the middle, as shown in Figure 1. A small web-application is developed to maintain students' examination database and to

Table 2: Matching results of benign and candidate structure of script

Ex.	Benign Structure	Candidate Structure	Malicious String	attack/normal
1	Keyup, val, to Uppercase ajax, css, abort, show, modal, html, css	Keyup, val, to Uppercase ajax, css, abort, show modal, html css	nil	normal
2	Keyup, val, to Uppercase ajax, css, abort, show, modal, html, css	Keyup, val, to Uppercase, ajax, css, abort, show, modal, html css, cookie, split, substring	cookie, split, substring	attack

retrieve their academic information - result sheet, grade card and transcript. Accordingly, the application generates three response pages or web-document based on user request.

In the server side operation, each page containing two functional scripts, in which, the first two page containing scripts having validation code to validate user input data - roll number, term year, term type (Spring or Autumn). The third page containing script to validate user input value for the roll number or registration number. An application program executes to retrieve *script proof* of the generated web document and embedding into it before executing the send command. Each of the page containing one additional *script* for keeping *script proof*.

In the client-side operation, a plug-in module called scripts verification (SV), verifies the response page before browsing the actual page content. The module performs three operations - (1) it retrieves *script proof* of the page; (2) generates scripts structure (*candidate proof* of the page by sequentially retrieving each *script* other than the *script* containing *script proof*; (3) performs exact matching between *candidate proof* and *script proof* and gives output message based on the matching result.

In Figure 4, two malicious scripts containing unintended or *attack* Java function *document.cookie()*, *document.cookie.split(';')* and *return c.substring(name.length, c.length)*. These are mainly applied for printing the cookie details, having session information. This information can be redirected to the attacker's site so that without the knowledge of legitimate user, information may be retrieved by the attacker. The proposed detection module is tested with a series of web-document having *normal* and *attack* scripts written with Java functions. The proposed detection module also tested using some of the publicly available data-set containing malicious scripts [4, 11, 22]. Using these publicly available attack scripts, simulation of test module has been done with 20 successive operations against each response page or web-document. In each operation, one response page is tested by inserting one and more malicious scripts available in the data-set. Satisfactory results has been found in every operation. In Table 2, sample results of some of the client-sided experiment are shown. In this Table 2, the *script proof* retrieved from the web-document based on the sequence of functions available in the script. The first document containing one *script*, accordingly, all the functions starts with dot (.), other than the statement define file with the extension dot(.), i.e., while retrieving from first document, it is not included the statement *url (blueCircle.gif)*. The sequence of *function*s in the respective proof are kept separated with delimiter coma (,). While testing with public data-set, we did not find any false positive.

In Table 2, the sample results are shown based on our experiment using a typical application with limited numbers of user access. In a largely accessible web application, where, in generates different run-time scripts for every response page, the proposed method of generating benign *script* structure and embedding as *script proof* into it, may effects the performance of web application access. To improve the efficiency, an indexing technique is applied to a sequence of probable *script proof*s generated during training phase. The index of a script structure is prepared by considering first 3 characters of each Java function available in it. At run-time, the server-side module retrieves the appropriate *script proof* of a response page using its index. To test the indexing technique, separate applications are developed so that for every legitimate user request it generates different run-time scripts. The proposed indexing-based verification process is tested with 100 numbers of distinct run-time scripts. Using indexing technique, performance of proposed method has been improved while testing at run-time. Depending upon the accessibility and dimension of web applications, the performance of proposed method can be upgraded using multiple indexing technique. A sample shot of indexed *script proof*s are shown in Table 3. A comparative report based on results found in series of experiments is reported in Table 4.

6 Conclusion

Scripting attacks due to the unintended functional statement in the web document is considered to be one of dangerous threats to the web enabled applications. To avoid such attack, both application developer and user must aware with this attack and accordingly, they must be careful in developing software and accessing the same

Table 3: Index of script structures

Sl	Index	Script Structure
1	KeyvaltoUajacssa boshomodhtmcss	Keyup, val, toUppercase, ajax, css, abort, show, modal, html, css
2	Keyvalajacssa boshomodhtmcss	Keyup, val, ajax, css, abort, show, modal, html, css
3	KeyvaltoUajacs sabomodhtmcss	Keyup, val, toUppercase, ajax, css, abort, modal, html, css
4	Keyvalajacssabo shomadhtlcss coosplsub	Keyup, val, toUppercase, ajax, css, abort, show, modal, html, css, cookie, split, substring

Algorithm 1 Function to retrieve structure of a script

1: Functionweb-document-structureInput:$list_web_functions(L)$,*web* – *document*(W_d) *Output:structure*$W_s[]$
2: Begin
3: $W_s[] \leftarrow 0$;
4: **while** not all *scripts* in W_d done **do**
5: Read script S;
6: **while** not all *substrings* in S done **do**
7: Retrieve a sub-string W_s from W_d;
8: **while** not functions in L from 1 to n done **do**
9: Read a function in $L \leftarrow f_w$;
10: **if** $W_s = f_w$ **then**
11: $W_s[] \leftarrow concat(W_s[], W_s)$;
12: **end if**
13: **end while**
14: **end while**
15: **end while**
16: Return $(W_s[])$;
17: EndFunction
18: End

Algorithm 2 Script structure matching

1: Function exact-matchingInput:Test web-document R_p *Output:page-status*
2: begin
3: Read R_p;
4: Retrieve benign structure B_s from R_p;
5: Cut portion of B_s from $R_p \leftarrow Ri_p^{'}$
6: $C_s =$ call $web - document - structure(R_p)$;
7: **if** $C_s = B_s$ **then**
8: page-status='Normal web-document';
9: **else**
10: page-status='Malicious or Attack web-document';
11: **end if**
12: Return (page-status);
13: EndFunction
14: end

respectively. In the server-side operation, the previously stored malicious content that need to be addressed. In a typical method, the *script proof* must be generated before retrieving the stored data that effects in constructing the *scripts* of a web document. This is due to the fact that if the previously stored data is a malicious content than the *script proof* also will be effected. However, this may not be possible always, as, there can be applications where scripts in web document generates dynamically with the stored data. To avoid such situation where the script generates using the stored content the filtering of stored content or retrieved data must be required based on the initialized constraint. However, in the client-side application this method has no limitations and can be applied directly to avoid *script* attacks. However, if the dynamically generated *scripts* in the response page are due to the multiple *web-application* servers before coming to the client's machine, then the proposed method need to be explored with signature generation and verification in each server.

Acknowledgments

This work is funded by Ministry of IT, Govt of India under the scheme of ISEA, Phase II.

References

[1] E. Athanasopoulos, A. Krithinakis, and E. P. Markatos, "Hunting cross-site scripting attacks in the network," in *Proceedings of the 4th Workshop on Web 2.0 Security Privacy (W2SP'10)*, pp. 1–8, 2010.

[2] T. S. Barhoom and S. N. Kohail, "New server-side solution for detecting cross site scripting attack," *International Journal of Computer Information Systems*, vol. 3, no. 2, 2011.

[3] P. Bisht and V. N. Venkatakrishnan, "XSS guard: Precise dynamic prevention of cross-site scripting attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 23–43, 2008.

[4] Blwood, *Multiple XSS Vulnerabilities Intikiwiki 1.9x, Mailing List Bugtrap*, May 2006. (`http://www.securityfocus.com/archive/1/435127/30/120/threaded`)

[5] CWE-79, *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*, June 2010. (`http://cwe.mitre.org/data/definitions/79.html`)

[6] J. H. Hayes and A. J. Offutt, "Input validation analysis and testing," *Empirical Software Engineerings*, vol. 11, no. 4, pp. 493–522, 2006.

[7] M. S. Hwang, S. F. Tzeng, and S. F. Chiou, "A non-repudiable multi-proxy multi-signature scheme," *In-*

Table 4: Comparative results based on simulation

Method	Constraint for application functionality	Implementation Pros and Cons	Average DR using public/own data
Third party black list with dynamic update (IMPERVA 2012)	Exists	(1) Research oriented taint & white lists (3) Depend on third party lists	90%
Bounding script execution [2, 10, 15]	Exists	(1) Reliable for script execution (2) Unable to handle typical application due to restrictions	80-90%
Initialize taint or trust codes in client or server [3, 17, 21, 23, 26]	Exists	(1)Reliable for script execution (2) Unable to handle typical application due to initialization	89%
Proposed Method	Not exists	(1)Independ-ent handling of server & client operations (2) Effective verification of malicious content with script proof	98%

novative Computing, Information and Control Express Letters, vol. 3, no. 3, pp. 259–264, 2009.

[8] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *IEEE Symposium on Security and Privacy*, pp. 263–268, 2006.

[9] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 1–11, 2009.

[10] E. Kirdaa, N. Jovanovicb, C. Kruegelc, and G. Vignac, "Client-side cross-site scripting protection.," *Computer and Security*, vol. 28, no. 7, pp. 592–604, 2009.

[11] J. Kratzer, "Jspwiki multiple vulnerabilities. posting to the bugtrap mailinglist," Sept. 2007. (http://seclists.org/bugtrap/2007/Sep/0324.html)

[12] H. T. Le and P. K. K. Loh, "Identification of performance issues in contemporary black-box web application scanners in SQLI," in *Latest Advances in Information Science and Applications*, pp. 211–216, 2012.

[13] C. C. Lee, C. H. Liu, and M. S. Hwang, "Guessing attacks on strong-password authentication protocol," *International Journal of Network Security*, vol. 15, no. 1, pp. 64–67, 2013.

[14] C. Lin, K. Lv Y. Li, and C. C. Chang, "Ciphertext-auditable identity-based encryption," *International Journal of Network Security*, vol. 17, no. 1, pp. 23–28, 2015.

[15] M. Madou, E. Lee, J. West, and B. Chess, "Watch what you write: Preventing cross-site scripting by observing program output," in *Application Security Conference*, pp. 1–14, 2008.

[16] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: A program query language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference*, pp. 1–19, 2005.

[17] K. K. Mookhey and N. Burghate, *Detection of SQL Injection and Cross-site Scripting Attack*, Mar. 17, 2004. (http://www.symantec.com/connect/articles/detection-sql-injection-and-cross-site-scripting-attacks)

[18] A. Mueller, *Cross Site Scripting (XSS)*, May 2009. (http://elegantcode.com/2009/05/28/cross-site-scripting-xss/)

[19] OWASP, *XSS (Cross Site Scripting) Prevention Cheat Sheet*, Jan. 2010. (http://www.owasp.org/index.php/XSS-Prevention-Cheat-Sheet)

[20] C. Pan, S. Li, Q. Zhu, C. Wang, and M. Zhang, "Notes on proxy signcryption and multi-proxy signature schemes," *International Journal of Network Security*, vol. 17, no. 1, pp. 29–33, 2015.

[21] R. Pelizzi and R. Sekar, *Protection, Usability and Improvements In Reflected XSS Filters*, 2012. (http://www.seclab.cs.sunysb.edu/seclab1/pubs/xss.pdf)

[22] A. Pigrelax, *XSS in Nested Tag in PHPBB 2.0.16. Mailing List Bugtrap*, July 2005. (http://www.securityfocus.com/archive/1/404300)

[23] R. Sekar, *An Efficient Black-box Technique for Defending Web-application Attacks*, Defense Advanced Research Project Agency, the Naval Surface Weapons Center, 2009.

[24] H. Shahriar and M. Zulkernine, "Mutec: Mutation-based testing of cross site scripting," in *Proceedings of the 5th International Workshop on Software Engineering for Secure Systems*, pp. 47-53, IEEE, 2009.

[25] J. Shanmugam and M. Ponnavaikko, "A solution to block cross site scripting vulnerabilities based on service oriented architecture," in *EEE/ACIS International Conference on Computer and Information Science (ICIS'07)*, 2007.

[26] Z. Su and G. Wassermann, "Static detection of cross-site scripting vulnerabilities," in *In 30th ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[27] C. S. Tsai, C. C. Lee, and M. S. Hwang, "Password authentication schemes: Current status and key issues," *International Journal of Network Security*, vol. 3, no. 2, pp. 101-115, 2006.

[28] S. F. Tzeng, M. S. Hwang, and C. Y. Yang, "An improvement of nonrepudiable threshold proxy signature scheme with known signers," *Computers and Security*, vol. 23, no. 2, pp. 174–178, 2004.

[29] S. F. Tzeng, C. C. Lee, and M. S. Hwang, "A batch verication for multiple proxy signature," *Parallel Processing Letters*, vol. 21, no. 1, 2011.

[30] S. F. Tzeng, C. Y. Yang, and M. S. Hwang, "A nonrepudiable threshold multi-proxy multi-signature scheme with shared verication," *Future Generation Computer Systems*, vol. 20, no. 5, pp. 887–893, 2004.

[31] H. C. Wu, M. S. Hwang, and C. H. Liu, "A secure strong-password authentication protocol," *Fundamenta Informaticae*, vol. 68, no. 4, pp. 399–406, 2005.

Biography

Debasish Das, received his Ph.D. in Computer Science and Engineering from Tezpur University in 2015 in the field of Network Security. He has been involved in application systems design & development and teaching for last 20 years. Presently, he is working as Systems Analyst in the Department of Computer Science and Engineering at Tezpur University. His research area includes Network and Information Security, Machine Learning and Financial Computing.

Dhruba Kr Bhattacharyya received his Ph.D. in Computer Science from Tezpur University in 1999 in the field of Cryptography and Error Control Coding. He has been in teaching for last 24 years and presently, he is a Professor at HAG in the Computer Science and Engineering Department at Tezpur University. His research areas include Machine Learning, Network Security and Bioinformatics. Prof. Bhattacharyya has published more than 240 research articles in the leading International Journals and peer-reviewed conference proceedings. Dr. Bhattacharyya also has written/edited 12 books. He is a fellow of IETE. Dr Bhattacharyya is on the editorial boards of several international journals and also on the programme committees/advisory bodies of several international conferences/workshops.