2011 35th IEEE Annual Computer Software and Applications Conference Workshops

Injecting Comments to Detect JavaScript Code Injection Attacks

Hossain Shahriar and Mohammad Zulkernine
School of Computing
Queen's University, Kingston, Canada
{shahriar, mzulker}@cs.queensu.ca

***Abstract*— Most web programs are vulnerable to cross site scripting (XSS) that can be exploited by injecting JavaScript code. Unfortunately, injected JavaScript code is difficult to distinguish from the legitimate code at the client side. Given that, server side detection of injected JavaScript code can be a layer of defense. Existing server side approaches rely on identifying legitimate script code, and an attacker can circumvent the technique by injecting legitimate JavaScript code. Moreover, these approaches assume that no JavaScript code is downloaded from third party websites. To address these limitations, we develop a server side approach that distinguishes injected JavaScript code from legitimate JavaScript code. Our approach is based on the concept of injecting comment statements containing random tokens and features of legitimate JavaScript code. When a response page is generated, JavaScript code without or incorrect comment is considered as injected code. Moreover, the valid comments are checked for duplicity. Any presence of duplicate comments or a mismatch between expected code features and actually observed features represents JavaScript code as injected. We implement a prototype tool that automatically injects JavaScript comments and deploy injected JavaScript code detector as a server side filter. We evaluate our approach with three JSP programs. The evaluation results indicate that our approach detects a wide range of code injection attacks.**

Keywords: Comment injection; JavaScript code injection.

I. INTRODUCTION

Cross site scripting (XSS) is one of the worst vulnerabilities in web-base programs. XSS vulnerabilities are exploited by injecting HTML contents or JavaScript code [16]. However, the most notorious effect can be achieved by injecting JavaScript code that gets executed by browsers. The injected JavaScript code can access sensitive information present in web pages and transfer it to third party websites. Recent research has shown that injected JavaScript code is the primary source of different exploitations such as cross site request forgery [3] and phishing [4]. Thus, detection of injected JavaScript code is an essential step towards mitigating XSS vulnerability exploitations.

A number of mitigation techniques (*e.g.*, static analysis [7], testing [18], and scanning [9]) are widely used before program deployments to detect injected code. However, the number of reported JavaScript code injection exploits in released programs is increasing day by day [10]. This indicates that development of vulnerability free programs is still far from reality [1, 2]. Given that, development of a complementary runtime detection of injected JavaScript code exploitations for web-based environment is important. Currently, browsers enforce the Same Original Policy (SOP) that prohibits accessing the Document Object Model (DOM) of web pages downloaded from one domain to the DOM of web pages downloaded from different domains [5]. However, this protection is not enough to stop the execution of injected JavaScript code. For example, a webpage downloaded from *http://www.xyz.com* might include a script as *<script src = http://www.badwebsite.com/x.js ></script>*. The script (*x.js*) is downloaded from the *www.badwebsite.com*, and the code gets executed at the browser side in the context of *www.xyz.com*. Thus, server side might be the first place to look for symptoms of injected JavaScript code and mitigate the XSS vulnerability exploitations early.

Existing server side approaches that detect injected JavaScript code (*e.g.*, [17, 22, 23]) suffer from a number of limitations. First, detection of injected JavaScript code is performed at browsers where the server side gathers information on legitimate JavaScript code and transfers it to the client side. As a result, browser implementations need to be modified to interpret the information sent by the server side and execute JavaScript code accordingly [24]. We believe that browser modification is an error prone task and brings maintenance issues. Second, some approaches detect benign or legitimate JavaScript code present in program implementations and match them against JavaScript code present in response pages [23]. This idea may not detect injected JavaScript code, if it is similar to benign script. For example, an attacker might choose to inject a method call that is already present in a response page to introduce subtle unwanted behavior. Finally, most of the approaches assume that websites do not include JavaScript code from third party websites. In practice, a program might include JavaScript files from third party websites to have enhanced client side functionalities. However, an attacker might access to these files and introduce unwanted side effects such as overriding method definitions [19].

In this paper, we address these issues by developing a server side JavaScript code injection detection approach. We pre and postpend each legitimate JavaScript code block with comment statements that include identical random token. We identify the expected features of a JavaScript code block (*e.g.*, method call, method definition), save the features in policies, and embed the policy information into comments. During the deployment phase, we perform a number of checks to detect injection attacks. These include (i) code without comment, (ii) code with correct and duplicate comment, and (iii) code with correct and non-duplicate

978-0-7695-4459-5/11 $26.00 © 2011 IEEE
DOI 10.1109/COMPSACW.2011.27

comment; however, the actual code features are not matching with the intended features specified in a policy.

We apply the proposed approach for server side programs implemented in Java Server Pages (JSP). We develop a prototype tool in Java to inject JavaScript comments and generate policies based on legitimate code features. We deploy the injected code detector as a server side filter. We evaluate our approach with three real world JSP programs. Our evaluation indicates that the proposed approach can mitigate many types of injected JavaScript code that might contain arbitrary and legitimate method call injection, and method definition overriding. The results show that the approach suffers from zero false negative rates. Our proposed approach has several advantages. First, no information needs to be sent to the browsers, and no modification of browser or server script engine is required. Second, a program suffering from weak client or server side input validation can be safe against JavaScript code injection attacks. Finally, our approach does not require executing actual JavaScript code. Rather, it detects injected code and nullifies any potential damages of website users early.

This paper is organized as follows. Section II provides an example of code injection attack. Section III discusses related works. Section IV presents JavaScript comment injection followed by attack detection approach. Section V provides implementation and evaluation details. Finally, Section VI concludes this paper and discusses future works.

II. JAVASCRIPT CODE INJECTION ATTACK EXAMPLES

In a method call injection attack, an attacker invokes a method that is defined by a user, or supported by native (*e.g.*, *Array*) or host (*e.g.*, *Document*) objects. An attacker might inject a method definition followed by a call to perform malicious activities. In a method definition overriding attack, an attacker modifies the definition of an already implemented method that is provided by a programmer or supported by native or host objects [19, 20]. If method definitions are injected without calls, then the attacker relies on method calls that are present in the implemented code.

We provide an example of XSS vulnerability in a JSP program as shown in Figure 1 (adopted from *JAuction* [14], an online auction engine). The program displays the profile of an existing user. Lines 1-8 contain JavaScript code that defines two methods: *banUser* and *activateUser*. An admin can set the status of an auction user as banned and active with these two implemented methods. Line 10 extracts a user id (*request.getParameter("id")*), and Line 11 performs a database query to retrieve the user profile (*userMethod.getUserDetails*). Lines 12-20 display the retrieved profile of a user related to the id in a table. We note that the first name and current status of a user are retrieved at Lines 14 (*rs.getString("first_name")*) and 16 (*rs.getString("status")*), respectively. The table generates two hyperlinks whose *href* attributes invoke the defined methods. Note that the *first_name* field is not filtered (*e.g.*, replacing '<' with *#<*) before including its value in a response page. Thus, the code is vulnerable to code injection attacks. Let us assume that an attacker injects an arbitrary method call such as *"<script>alert('xss');</script>*. The response becomes *<td><input type="text" name="first_name" value=""<script>alert('xss');</script >></td>*. When a browser interprets the response, an alert box would appear with the text *xss*. An attacker might toggle the behavior of the two methods (*banUser* and *activateUser*) and inject the payload as *"><script>function banUser(){document.form.status.value="Active";}function activateUser(){document.form.status.value="Banned";}</sc ript>*. Note that in the injected script code, the two defined methods have similar names to the legitimate code implemented by a programmer. However, the *banUser* method sets the *status* field value as *active*, and *activateUser* method sets the *status* field value as *banned*. Thus, legitimate method definitions have been overridden by injected code. Note that the attack does not require injecting any method calls as necessary calls (*i.e.*, *banUser* and *activateUser*) are already present in the implementation.

```
1. <script>
2.   function banUser(){
3.     document.form.status.value = "Banned";
4.   }
5.   function activateUser(){
6.     document.form.status.value = "Active";
7.  }
8. </script>
9. <%
10.  String uid = (String)request.getParameter ("uid");
11.  ResultSet rs = userMethods.getUserDetails(uid);  %>
12.  <table>
13.      <tr> <td>First Name:</td>
14.          <td><input type= "text"  name= "first_name"  value = "<%=rs.getString("first_name")%>"> </td>
15.          <td>User Status:</td> </tr>
16.     <tr> <td><input type = "text" name= "status"  value = "<%=rs.getString("status")%>">
17.                  <a href = "javascript:banUser();"> Ban User </a>
18.                  <a href= "javascript:activateUser();">Activate User </a>
19.          </td>
20.     </tr>
```

Figure 1. Code snippet vulnerable to JavaScript code injection attacks

In practice, many web-based programs generate response pages where JavaScript methods are defined before they are invoked. However, between method definition and method invocation, these pages also render unsanitized user supplied contents.

III. RELATED WORKS

Table I provides a summary comparison of related works compared to our approach based on five features: modification of JavaScript code, detection of legitimate method call injection, method definition overriding, and injected JavaScript code downloaded from remote website, and usage of JavaScript code features.

TABLE I. SUMMARY OF JAVASCRIPT INJECTION DETECTION WORKS

Work	Script code mod.	Method call injection	Method def. overriding	Injected code in remote website	Script code feature
Gundy *et al.* [24]	No	Yes	Yes	No	No
Nadji *et al.* [21]	Yes	Yes	Yes	No	No
Jim *et al.* [8]	No	No	No	No	No
Johns *et al.* [15]	Yes	No	No	No	No
Wurzinger *et al.* [17]	Yes	No	No	No	No
Bisht *et al.* [23]	No	No	No	No	No
Pietraszek *et al.* [25]	No	Yes	Yes	No	No
Futoransky *et al.* [22]	No	Yes	Yes	No	No
Chin *et al.* [12]	No	Yes	Yes	No	No
This work	**No**	**Yes**	**Yes**	**Yes**	**Yes**

Gundy *et al.* [24] randomize XML namespace prefix in generated HTML pages at the server side. They annotate HTML tags and attributes with random tokens and send specific rendering policies from the server side to the client side. The policies specify rules for allowing or disallowing rendering of HTML elements at browsers. Nadji *et al.* [21] detect XSS attacks through the notion of document structure integrity where attacker injected code cannot alter the parse tree of DOM. Jim *et al.* [8] generate hashes for legitimate JavaScript code at the server side and send them to browsers to get validated by browsers. Johns *et al.* [15] detect reflected XSS attacks based on the observation that injected code is present in HTTP requests and responses. Wurzinger *et al.* [17] encode all legitimate JavaScript function calls as syntactically invalid code so that attacker injected code gets executed. Bisht *et al.* [23] compare the parse tree of benign JavaScript code with runtime generated JavaScript code to detect attacks. However, the parse tree of programmer implemented legitimate method call and an injected legitimate method call would be identical. Thus, the approach could be circumvented for specific code injection attacks that our approach can address. Some approaches apply tainted data flow-based analysis by marking inputs as tainted, propagating taint information during program operations, and checking operations where tainted data is used. However, propagation of tainted information requires reimplementation of APIs and script interpreters [12, 22, 25].

IV. INJECTED JAVASCRIPT CODE DETECTION

We mark the boundary of legitimate JavaScript code present in each source files by injecting comment statements before and after the code block. For a given legitimate code block, the comment statements include identical random tokens and the features of the legitimate JavaScript code (*e.g.*, method definitions and calls). These features should be present in generated response pages. We store features in policies and embed the policy information in comment statements for runtime checking. When a response page is generated, we intercept the page and perform a number of checks to detect attacker injected code. We first check if any JavaScript code without comment is present or not. If no comment is present, then it is identified as injected code, and the code is removed from response page. If code has comments, then the module checks the validity of random tokens and policy information present within comments. If a token is invalid, then the code is considered as injected. Otherwise, the module checks whether the suspected JavaScript code features match with the features specified in the policy information of comment. If a mismatch is identified, then the code is injected. Finally, the module checks the presence of duplicate comments to detect attacker injected comments. If injected JavaScript code is detected in a response page, then it is removed along with injected comments before forwarding response pages to browsers. We discuss more on comment injection and policy generation in Subsections A and B, respectively.

A. Comment injection

We consider some of the most common cases where JavaScript code might be present: (i) inline (*<script> … </script>*), (ii) script inclusion using local source file (*e.g.*, *<script src= "a.js"> </script>*), (iii) script inclusion using remote source file (*e.g.*, *<script src= "http://www.xyz.com/a.js"> </script>*), (iv) event handler code (*e.g.*, *onclick = "foo"*), and (v) URL attribute values (*e.g.*, *<a href = "javascript:window.history.goback()">*). Table II shows examples of JavaScript comment injection for these five cases. In the first example, an inline script code defines a method named *foo*. We pre and postpend the definition with comment statements (*/*t_1*/*). Here, t_1 is a random token (*e.g.*, a sequence of 32 random characters). The second example shows that JavaScript code is loaded from a local resource file named *a.js*. We do not instrument comments for this case as the local JavaScript files are analyzed separately. This allows us avoiding injection of multiple comments. The third example shows a remote JavaScript source file inclusion. We pre and postpend script tags with the comments. This approach is taken as we assume that we cannot modify the remote script file. The fourth example shows an event handler method call named *foo*. Here, we insert a comment for the method call. The last example shows a JavaScript code present in an anchor tag attribute (*href*) value and corresponding comment injection.

TABLE II. EXAMPLES OF JAVASCRIPT COMMENT INJECTION

Type	Example code	Code with injected comment
Inline	<script> function foo(){ … }; </script>	<script>**/*t1*/** function foo(){ … }; **/*t1*/**</script>
Script inclusion (local)	< script src= "a.js">	< script src= "a.js">
Script inclusion (remote)	<script src= "http://www.xyz.com/a.js"></script>	**<script>/*t1*/</script>** <script src= "http://www.xyz.com /a.js"> </script> **<script>/*t1*/</script>**
Event handler	<input … onclick= "foo();"/>	<input … onclick= "**/*t1*/**foo();**/*t1*/**"/>
URL attribute value	<a href = "javascript:window.history.goback()">	<a href = "**/*t1*/** javascript:window.history.goback()**/*t1*/**" >

B. JavaScript code feature-based policy generation

We now discuss some of the most useful features that we extract from legitimate code followed by the generation of policies in the next two subsections.

1) Feature identificaiton

Injected code might not be meaningful unless method calls are present. Similar conclusion can be drawn for injected method definition that might override the behavior or programmer implemented method. Thus, we extract the signature of method definition and call as the most interesting features from legitimate JavaScript code. We identify method names, parameters, and arguments by leveraging a JavaScript parser (*e.g.*, Rhino [11]). Table III shows three examples of legitimate method definitions (user defined named and anonymous, host object method overriding), and two examples of legitimate method calls (simple, nested).

TABLE III. EXAMPLE FEATURES OF METHOD DEFINITIONS AND CALLS

Type	Example code	Expected feature
User defined method (named)	function foo (x, y){ … };	[foo, 2, x, y]
User defined method (anonymous)	var foo = function (x, y){ … };	[foo, 2, x, y]
Host object method (override)	document.write = function (arg1){ … };	[document.write, 1, arg1]
Simple method call	foo(2,3)	[foo, 2, 2, 3]
Nested method call	foo(2, foo(2,3))	[foo, 2, 2, [foo, 2, 2, 3]]

The second column shows code snippets for these examples, and the third column shows expected features. In the first example, a user defined method named *foo* has two parameters (*x*, *y*). We denote the expected feature as [*foo*, *2*, *x*, *y*]. Here, the first and second entities indicate the method name and parameter count, respectively. The remaining entities are the parameter names. The second row shows an anonymous method (*i.e.*, a function without a given name). In this case, the method definition is saved in a variable named *foo*. This would allow a programmer to invoke the anonymous method with *foo*. We identify the name of the immediate variable where the method definition is assigned along with the number of parameter and their names. The third example shows a host object method (*document.write*) definition overriding. Here, the expected feature includes *document.write* (as the defined method), 1 (parameter count), and *arg1* (argument). The fourth example shows a simple method call that takes two arguments. We identify the expected features as [*foo*, *2*, *2*, *3*]. The last example shows a nested method call and the corresponding expected features ([*foo*, *2*, *2*, [*foo*, 2, 2, 3]]).

2) Policy generation

The extracted code features are stored in policy files so that JavaScript code present in response pages can be compared against these known features to detect injected JavaScript code. We store the feature information in XML format. Moreover, comments include the policy information. Table IV shows example code, policy information, and modified comments with policy information in the first, second, and third columns, respectively.

TABLE IV. EXAMPLES OF POLICY GENERATION

Example code	Example policy	Modified comment
<script> function foo(x, y){ … }; </script>	<policyID>1</policyID > <type>def</type> <name>foo</name> <paramCount>2</paramCount> <param>x</param> <param>y</param>	<script> **/*t1:1*/** function foo (x, y){ … }; **/*t1:1*/** </script>
<input onclick = "foo (2, 3);" … />	< policyID>2</policyID > <type>call</type> <name>foo</name> <argCount>2</argCount> <arg>2</arg> <arg>3</arg>	<input onclick = "**/*t1:2*/** foo (2, 3); **/*t1:2*/**" … />

The first example shows a method definition *foo* with two parameters named *x* and *y*. We encode the expected feature information with four mandatory fields: *policyID* (a unique policy id), feature *type* (*def* for method definition), *name* (the name of the defined method), and *paramCount* (the number of parameters). Depending on the number of parameters, the *param* field contains the parameter name. The third column shows the modified comment that contains both random token and the policy id (*i.e.*, */*t1:1*/*). Similarly, an example policy is shown for the method call

foo in the second example. Here, the field *type* is set as *call* and the *argCount* saves the number of supplied arguments. The *arg* saves the arguments.

V. IMPLEMENTATION AND EVALUATION

We implement a prototype tool in Java that parses source files with Jericho [13] and injects comments in JavaScript code blocks present. To insert comments, we modify appropriate program locations (*e.g.*, event handler method call embedded with comments). The *OutputDocument* object allows us to save the modified program back to a source file. The Rhino [11] is used to parse the JavaScript code and extract the features based on the abstract syntax tree. The random token is generated before inserting comments. We store the generated policies in web program containers. The JavaScript code feature comparator module is implemented as a server side filter. The filter intercepts all the generated responses to detect injected JavaScript code.

We evaluate our approach with three open source JSP programs chosen randomly from the *sourceforge.net*. Table V shows the summary of the JavaScript code that is present in these programs: inline, script source inclusion (local), script source inclusion (remote), event handler, and URL attribute value. Most of the JavaScript code is present as inline code or event handler attribute value. These programs employ very few number of local JavaScript source file inclusion. However, none of the programs include remote JavaScript file source. Table VI shows a summary of the JavaScript code features and the number of policy generated for these programs. The code features include the number of method definition, parameter count, method call, argument count, user defined method call, and host object method call.

TABLE V. SUMMARY OF JAVASCRIPT CODE TYPES

Program name	Inline	Script inclusion (local)	Script inclusion (remote)	Event handler	URL attribute
JVote	2	2	0	2	0
JAuction	21	1	0	11	4
JInsure	15	2	0	13	21

TABLE VI. SUMMARY OF FEATURES AND POLICY GENERATION

Program name	Method definition	Param. count	Method call	Arg. count	User method call	Host method call	Total policy
JVote	6	17	13	17	11	2	14
JAuction	45	82	84	81	81	3	44
JInsure	26	38	36	2	13	23	52

The number of policy generated for these programs vary mainly due to the way method definitions and calls are present in these programs. For example, multiple method definition and calls are present in just one code block (*e.g.*, inline). Thus, we get less number of policies (*e.g.*, *JAuction*). However, *JInsure* and *JAuction* have the highest number of calls present in event handler and URL attribute value.

A. Experimental setup and evaluation results

We deploy the modified programs (injected comments) in a Jakarta Tomcat server (version 7.0) running on Windows 7. All programs store information in the MySQL database. Our evaluation is a two step process. First, we execute programs to inject JavaScript code while performing functionalities that accept inputs and store the injected code (denote as *initial* functionalities). Second, we execute programs to perform functionalities so that injected inputs are retrieved from storages and appear in response pages (denote as *complementary* functionalities). In particular, we observe whether the responses contain injected attack payloads or not while enabling the filter. Table VII shows the number of initial functionalities, brief description of initial and complementary functionalities. We choose one (*F1*), four (*F2-F5*), and three (*F6-F8*) functionalities from *JVote*, *JAuction*, and *JInsure*, respectively.

TABLE VII. SUMMARY OF FUNCTIONALITY

Program	Fn.	Initial	Complementary
JVote	F1	Create poll	View poll
JAuction	F2	Add user,	View profile,
	F3	Add top level category,	View top level category,
	F4	Add first level category,	View first level category,
	F5	Add auction	View auction list
JInsure	F6	User registration,	Update user profile,
	F7	Agent registration,	Update agent profile,
	F8	Add policy	View policy

We apply eight types of JavaScript code injection attack during our evaluation. Table VIII shows a summary of these attacks (denoted as a_1-a_8). The first four attacks (a_1-a_4) inject no injection of comments. However, the remaining attacks include injection of incorrect (a_5 and a_6) and correct (a_7-a_8) comments. Moreover, the attacks include injection of arbitrary (a_1 and a_2) and legitimate code injection that is present in programs (a_3-a_8). The code injection includes both method call (a_1, a_3, a_5, and a_7) and method definition overriding (a_2, a_4, a_6, and a_8).

Table IX shows a summary of the injected JavaScript code (columns 3-10) for the three programs. We observe that in the *JVote* program, only one host object method call is present (*confirm*) in the response page related to *F1*. The method generates a dialog box to take consent for performing specific actions. Thus, we are able to generate all the eight types of attack cases (a_1-a_8). For the *JAuction* program, the response page of viewing profile (*F2*) contains two method definitions and calls. This allows us to apply all test cases (a_1-a_8). However, the response pages for the remaining functionalities (*F3-F5*) contain no legitimate JavaScript method call or definition. Thus, we are able to inject only a_1 and a_2 attack test cases. In *JInsure*, multiple method definitions (*e.g.*, *checkAll* and *checkNull*) are present in response pages related to complementary functionalities (*F6-F8*). This allows us to apply all the injected attack types (a_1-a_8). The last two columns show the total number of injected attacks and total number of detected attacks, respectively. We observe that our approach detects all the injected test cases for all the programs. The obtained false

negative rate is zero which is computed as *1- (# of detected attacks/# of injected attacks*).

TABLE VIII. SUMMARY OF ATTACK TYPES

Attack	Brief attack description
a_1	Inject a method call dissimilar to legitimate method call (no comment).
a_2	Inject a method definition and a call dissimilar to a legitimate method call (no comment).
a_3	Inject a method call similar to a legitimate method call (no comment).
a_4	Override a method definition (no comment).
a_5	Inject a method call similar to a legitimate method call (incorrect comment).
a_6	Override a method definition (incorrect comment).
a_7	Inject a method call similar to a legitimate method call (correct comment).
a_8	Override a method definition (correct comment).

TABLE IX. SUMMARY OF INJECTED ATTACKS AND DETECTION RATES

Program	Fn.	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	Inj.	Det.
JVote	F1	1	1	1	1	1	1	1	1	8	8
JAuction	F2	1	1	1	1	1	1	1	1	8	8
	F3	1	1	0	0	0	0	0	0	2	2
	F4	1	1	0	0	0	0	0	0	2	2
	F5	1	1	0	0	0	0	0	0	2	2
JInsure	F6	1	1	1	1	1	1	1	1	8	8
	F7	1	1	1	1	1	1	1	1	8	8
	F8	1	1	1	1	1	1	1	1	8	8

VI. CONCLUSIONS AND FUTURE WORK

XSS vulnerabilities could be leveraged to perform malicious activities by injecting JavaScript method calls and method definition overriding. Existing server side JavaScript code injection detection approaches can be eluded by injecting legitimate JavaScript code or altering code downloaded from trusted websites. This paper addresses these challenges by injecting comments for legitimate JavaScript code that encode legitimate code features in terms of method definition and call signatures. This approach makes it very difficult to inject arbitrary and legitimate JavaScript code. During response page generation, we perform a number of checks to detect injected JavaScript code. The approach detects a wide range of attacks including arbitrary and legitimate method call injection and overriding legitimate method definition. We implement a prototype tool to inject comments in JavaScript code. Our evaluation results on three real world JSP programs show that the proposed approach detects a subset of code injection attacks and suffers from zero false negative rates. Due to the limitation of the JavaScript parser, some tasks require pre-processing of JavaScript such as removal of comment tags and removal of return keyword in event handlers. Future work includes finding automated ways to perform these tasks. We also plan to evaluate runtime overhead of our approach.

ACKNOWLEDGMENT

This work is supported by the Natural Sciences and Engineering Research Council of Canada and Bell Canada.

REFERENCES

[1] H. Shahriar and M. Zulkernine, "Mitigation of Program Security Vulnerabilities: Approaches and Challenges," *ACM Computing Surveys* (to appear).

[2] H. Shahriar and M. Zulkernine, "Taxonomy and Classification of Automatic Monitoring of Program Security Vulnerability Exploitations," *Journal of Systems and Software*, Elsevier, Vol. 84, Issue 2, Feb 2011, pp. 250-269.

[3] H. Shahriar and M. Zulkernine, "Client-Side Detection of Cross-Site Request Forgery Attacks," *Proc. of ISSRE*, San Jose, November 2010, pp. 358-367.

[4] H. Shahriar and M. Zulkernine, "PhishTester: Automatic Testing of Phishing Attacks," *Proc. of the SSIRI*, Singapore, June 2010, pp. 198-207.

[5] JavaScript Security in Mozilla, Accessed from www.mozilla.org/projects/security/components/jssec.html

[6] JVote, Accessed from http://sourceforge.net/projects/jspvote/

[7] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities", *ICSE*, Germany, 2008, pp. 171-180.

[8] T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," *Proc. of WWW*, Banff, Alberta, May 2007, pp. 601-610.

[9] Paros - Web application security assessment, Accessed from http://www.parosproxy.org/index.shtml

[10] Open Source Vulnerability Database, http://osvdb.org

[11] Rhino, Accessed from http:// www.mozilla.org/rhino

[12] E. Chin and D. Wagner, "Efficient Character-level Taint Tracking for Java," *Secure Web Services*, Nov 2009, pp. 3-11.

[13] Jericho HTML parser, http://jericho.htmlparser.net/

[14] JAuction-0.3, http://sourceforge.net/projects/jauction/

[15] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-Side Detection of Cross-Site Scripting Attacks," *Proc. of the ACSAC*, California, December 2008, pp. 335-344.

[16] Cross Site Scripting, www.owasp.org.

[17] P. Wurzinger, C. Platzer, C. Ludl, E. Krida, and C. Kruegel, "SWAP: Mitigating XSS Attacks using a Reverse Proxy," *Proc. of the SESS*, Vancouver, May 2009, pp. 33-39.

[18] H. Shahriar and M. Zulkernine, "MUTEC: Mutation-based Testing of Cross Site Scripting," *Proc. of the 5th ICSE Workshop SESS*, Vancouver, Canada, May 2009, pp. 47-53.

[19] S. Paola and G. Fedon, "Subverting Ajax," *Proc. of the 23rd CCC Conference*, Berlin, December 2006.

[20] B. Hoffman and B. Sullivan, *AJAX Security*, Addison-Wesley.

[21] Y. Nadji, P. Saxena, and D. Song, "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense," *Proc. of the NDSS*, San Diego, California, February 2009.

[22] A. Futoransky, E. Gutesman, and A. Waissbein, "A Dynamic Technique for Enhancing the Security and Privacy of Web Applications," *Proc. of Black Hat USA*, Las Vegas, 2007.

[23] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," *Proc. of the 5th DIMVA,* Paris, July 2008, pp.23-43.

[24] M. Gundy and H. Chen, "Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-site Scripting Attacks," *Proc. of NDSS*, San Diego, Feb 2009.

[25] T. Pietraszek and C. Berghe, "Defending Against Injection Attacks through Context-sensitive String Evaluation," *LNCS*, Vol.3858, September 2005, pp. 124.