ISSN: 2321-7782 (Online)
Volume 2, Issue 9, September 2014
International Journal of Advance Research in Computer Science and Management Studies
Research Article / Survey Paper / Case Study
Available online at: www.ijarcsms.com

An Enhanced Mechanism for Automated Removal of Cross-site Scripting Vulnerabilities using Input Validation

Jiss Varghese[1]
PG Scholar in CSE
Amal Jyothi College of Engineering
Kanjirappally – India

Treesa Antony[2]
PG Scholar in CSE
Amal Jyothi College of Engineering
Kanjirappally – India

Lisha Varghese[3]
Asst. Prof. Dept. of CSE
Amal Jyothi College of Engineering
Kanjirappally – India

Abstract: ***Cross-site scripting (XSS) vulnerabilities are today's most common and serious web application vulnerabilities. XSS enables attackers to inject harmful scripts into Web pages viewed by other users. When a web application executes these scripts, the result is a successful XSS attack. The malicious scripts will run with the same privileges as trusted content allowing it to run malicious code within the browser, impersonate the user to trusted servers, steal a victim user's private data and authentication credentials, or present forged content to the victim. Thus we need a mechanism to sanitize user inputs properly. The existing sanitization mechanisms are concerned with only some of the input contexts. We propose an enhanced scheme which considers a wide range of input contexts and performs validation also. It can be applied to real-world web applications and efficiently removes all the XSS vulnerabilities in them.***

Keywords: cross-site scripting,; input validation; escaping; canonicalization; context discovery; secure source code replacement.

I. INTRODUCTION

XSS vulnerabilities are one of the most dangerous and commonly reported vulnerabilities in web applications [10]. They are surprisingly easy to create and difficult to mitigate completely. Any web application that fails to properly sanitize user input before displaying it to other users will be vulnerable to XSS attacks.

Web browsers protect multiple web applications running within the same browser instance by isolating them according to the "Same Origin Policy". The Same Origin Policy does not allow web application from accessing the private data of other web applications. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the Same Origin Policy. Cross-site scripting uses known vulnerabilities in web-based applications, their servers, or plug-in systems they rely on. Exploiting one of these, they fold malicious content into the content being delivered from the compromised site. When the resulting combined content arrives at the client-side web browser, it has all been delivered from the trusted source, and thus operates under the permissions granted to that system. By finding ways of injecting malicious scripts into web pages, an attacker can gain elevated access-privileges to sensitive page content, session cookies, and a variety of other information maintained by the browser on behalf of the user. Cross-site scripting attacks are therefore a special case of code injection.

Most experts distinguish XSS attacks between at least two primary flavors: *non-persistent* and *persistent*. Some sources further divide these two groups into *traditional* (caused by server-side code flaws) and *DOM-based* (in client-side code). Non-persistent (reflected) cross site scripting attacks are the most common types. In a reflected XSS attack, the victim visits a page controlled by the attacker. The attacker encodes malicious content into a link or web form that targets a trusted web site. The victim clicks the link or submits the form and the malicious script is reflected by the trusted web server back to the victim's web

browser. In a stored XSS attack, the attacker causes malicious script to be stored directly on a trusted web server. When the victim visits the trusted web server at a later time, the attacker's malicious content will be delivered to the victim's web browser. Traditionally, cross-site scripting vulnerabilities would occur in server-side code responsible for preparing the HTML response to be served to the user. DOM-based vulnerabilities occur in the content processing stages performed by the client, typically in client-side JavaScript.

The solutions can be classified into defensive coding practices, input validation and XSS testing techniques, vulnerability detection techniques and attack prevention techniques. Except defensive coding practices, none of them remove XSS vulnerabilities exist in the program. The defensive coding practices are capable of removing the vulnerabilities. The main process of defensive coding approach for effective prevention [15], [16] of XSSVs is to escape all the user inputs used in HTML documents according to the contexts in which these inputs are referenced. Automated removal of cross site scripting vulnerabilities in web applications [3] uses the method of escaping user inputs based on their context. Escaping is the process of transforming the characters that have special meanings to a client-script interpreter into the representations that removes the special meanings that exist.

The proposed mechanism considers one of the major problems that the escaping approach may not consider. The automated removal based on escaping deals with only certain kinds of input context. The proposed mechanism deals with a wider range of input context and the input validation mechanism enhances the performance of the system. The remainder of the paper is organized as follows: Section 2 contains some related work. Section 3 proposes our XSS vulnerability detection and removal approach. Section 4 concludes the paper.

II. Related Work

A. *Static Analysis*

Static program analysis [8] is a traditional method to detect XSS vulnerabilities. Static analysis approaches cannot be both sound and complete, forcing a choice between false positives or missed vulnerabilities. By favoring a dynamic analysis approach, we can avoid loss of precision due to round-trips to the browser and difficult to support PHP features [17][18]. If greater precision is needed, our technique can be integrated with a full information flow tracking system.

B. *Information Flow Tracking*

Dynamic information flow tracking [14] is a hardware mechanism to protect programs against malicious attacks by identifying spurious information flows and restricting the usage of spurious information. Every security attack to take control of a program needs to transfer the program's control to malevolent code. The operating system identifies a set of input channels as spurious, and the processor tracks all information flows from those inputs. A broad range of attacks are effectively defeated by disallowing the spurious data to be used as instructions or jump target addresses [13].

C. *XSS Prevention Rules*

XSS injection is general attack achieved by switching to a code context illegally from a data context by using special characters (e.g., ‘‘‘’’, ‘‘‘‘’’, ‘‘;’’, ‘‘<’’). These are commonly significant to a targeted client side interpreter. OWASP [19] specifies a set of systematic XSS prevention rules to be followed. This ensures that any user input referenced in an HTML output is only treated as a data [3].

Appropriate escaping mechanism must be applied to the user input according to the HTML context in which the input is referenced, based on the above mentioned rules. Escaping disables the effect of special characters contained in user input and prevents them from invoking client side interpreters. XSSVs can be completely avoided by following OWASP's rules and there is also no harm in escaping the referenced entrusted data even if the HTML output is not actually vulnerable.

D. Escaping/Encoding

The above mentioned XSS prevention rules are enforced by escaping APIs implemented by ESAPI [1]. The ESAPI project is implemented for different web languages such as Java,.NET, and PHP. It provides a variety of security mechanisms such as authentication, validation, encoding, encryption, security wrappers, filters, and access control to mitigate various web security issues. This paper mainly focuses on the usage of validation and escaping services provided.

III. PROPOSED APPROACH

The proposed approach consists of three phases: (1) Canonicalization, (2) Input Validation and (3) Escaping.

A. Canonicalization

Canonicalization is the operation of reducing an encoded string down to its simplest form. The attackers may frequently use encoding to change their input in a way that will bypass validation filters, but still be interpreted properly by the target of the attack. Data encoded more than once is not something that a normal user would generate and should be regarded as an attack. All input received by the application should be canonicalized before we attempt to validate it to ensure we are validating the simplest form of the data and avoiding the risk posed by performing the validation on encoded data.

To perform canonicalization, we can call the ESAPI method:

String canonicalize(String input) throws Encoding Exception;

The method takes the input string to be canonicalized and a canonicalized string(or an error) will be returned.

B. Validation

Validation is a clear, simple, actionable guidance for providing Input Validation[2] security functionality in web applications. It prevents attacks as early as possible in the processing of the user's (attacker's) request. In addition, it detects unauthorized input before it is processed by the application. There are two approaches for validation: (1) Black list validation and (2) White list validation.

1) Black list validation: This is the approach which tries to detect attack characters and patterns like the ' character, the string 1=1, or the <script> tag. This is a massively flawed approach [4]. It is trivial for an attacker to avoid getting caught by such filters. Sometimes these filters frequently prevent authorized input, when some special character is being filtered out.

2) White list validation: This is the approach appropriate for all input fields provided by the user. If it's well structured data, like dates, social security numbers, zip codes, e-mail addresses, etc. then the developer should be able to define a very strong validation pattern, usually based on regular expressions, for validating such input[12]. If the input field comes from a fixed set of options, like a drop down list or radio buttons, then the input needs to match exactly one of the values offered to the user in the first place. The most difficult fields to validate are so called 'free text' fields, like blog entries. However, even those types of fields can be validated to some degree, you can at least exclude all non-printable characters, and define a maximum size for the input field.

The Input Validation Secure Development Principle

The Input Validation Secure Development Principle is certainly not a silver bullet but if you ensure that all of the data received and processed by your application is sufficiently validated you can go a long way towards preventing many of the common vulnerabilities being actively exploited by malicious users.

To address these issues we need to create your own functions to perform basic input validation as well more complicated security tasks such as canonicalization and escaping of potentially malicious input. The creation of such functions is a daunting and difficult task even for experienced developers. This is where the OWASP ESAPI can help us out, we can use the ESAPI to address the issues without needing to understand how to write functions to validate and securely process data in the web application.

OWASP Enterprise Security API Toolkits help software developers guard against security-related design and implementation flaws. Applications and services can be OWASP ESAPI-enabled (ES-enabled) to enable applications and services to protect themselves from attackers. We have to validate input, encode data where needed and return a nicely sanitised error message to the user if potentially malicious data is entered into the application. The ESAPI addresses all of these issues.

ESAPI Validation

The ESAPI Validator module provides a set of methods which will allows us to canonicalize and validate all user input into the application. The Validator module have several ready to use validation methods but the real strength is the ability to extend it to include customized validation methods or even modify the built in ones to suit the current requirements.

This example shows how to validate one of our defined data types using an inbuilt validation method. The data type we will validate is the credit card data type, this one is very easy with the ESAPI because we have a validation method called "**isValidCreditCard**" which does exactly as its name implies – it will take an input and validate it against a regular expression (the ESAPI only supports whitelist validation) for a credit card data type. And that right there is an area where the ESAPI can either be a simple "plug and play" security quick fix or something you take and customise to meet your own requirements.

booleanisValidCreditCard(String context, String input, booleanallowNull) throws IntrusionException;

This method will take input string and return a boolean value which makes it easier to handle the exception caused by invalid input into the application. This method has three parameters which are explained below:

String context – The parameters that are passing in to be validated, so in this case it would be CC Number.

String input – The input to validate.

booleanallowNull – The allowNull parameter allows to make this input mandatory or not. If the allowNull parameter is set to true then the input is allowed to be a NULL value or empty without throwing an exception. If this parameter is set to false a NULL or empty string will cause an exception.

throwsIntrusionException – If the input fails the validation check an exception is thrown.

C. Escaping

The XSS prevention rules force an appropriate escaping mechanism to be applied to the user input. This is based on the HTML context in which the input is referenced. User input usually contains special characters. Escaping is a transformation process which disables the effect of special characters upon client side execution.

The entire process of escaping is based on the following rules defined by OWASP [11]. A brief overview of these rules is essential in this context:

- Rule#0: Do not reference user inputs in any other cases except the ones defined in Rule#1–Rule#5. In some contexts, no special character may be required to perform XSS injection; and therefore, escaping rules could become complex or insufficient. Thus, escaping rules inRule#1–Rule#5 only apply to the typical contexts where user inputs are commonly referenced. This Rule#0 conditions that no user input is to be referenced in any other cases. This rule is the most important among all XSS prevention rules as it implies a white list approach .
- Rule#1: Use HTML entity escaping for the untrusted data referenced in an HTML element. For example, <body> htmlEscape(untrusted_data) </body>, where''htmlEscape()'' is the HTML entity escaping method, conforms to this rule.

- Rule#2: Use HTML attribute escaping for the untrusted data referenced as a value of a typical HTML attribute such as name and value. This rule does not apply to the two dangerous attributes—href and src. The only allowable way to reference untrusted data as values of href and src attributes is stated in Rule#5. Any other cases of untrusted data referenced in the contexts of href and src are disallowed under OWASP's escapingrules and OWASP recommends the use of only programmer defined data in such cases because the referenced untrusted data may simply point to a JavaScript source without the use of special characters. This rule also does not apply to all event-handler attributes such as on click.Event-handler attributes should be handled according to Rule#3. For example,<input value='htmlAttrEscape(untrusted_data)'>,where ''htmlAttrEscape()'' is the HTML attribute escaping method, conforms to this rule; however, <a href='htmlAttr-Escape(untrusted_data)'> does not.

- Rule#3: Use JavaScript escaping for the untrusted data referenced as a quoted data value in a JavaScript block or an event handler. This rule does not apply to the untrusted data referenced as any other ways in a code block except as a quoted data

  value. For example, <bodyonload=''x='javascriptEscape(untrusted_data)'''>. However, this rule applies to other client side scripts such as VBScript and Flash.

- Rule#4: Use CSS escaping for the untrusted data referenced as a value of a property in a CSS style. For example, <tablestyle=''width:cssEscape(untrusted_data)''>, where''cssEscape()'' is the CSS escaping method, conforms to this rule.

- Rule#5: Use URL escaping for the untrusted data referenced as a HTTP GET parameter value in a URL. For example, <ahref='http://www.site.com?name=urlEscape(untrusted_data)'> and <imgsrc='http://www.site.com?imgid=

  urlEscape(untrusted_data)'>, where ''urlEscape()'' is the URL escaping method, conform to this rule.

The process of escaping consists of two major phases: (1)XSS vulnerability detection and (2) XSS vulnerability removal. The first phase identifies potential XSS vulnerabilities in server programs. The second phase first identifies the code locations where the untrusted data can be adequately escaped, second determines the required escaping mechanisms, and then escapes the untrusted data using ESAPI's APIs.

1) XSS Vulnerability Detection: In a Control Flow Graph (CFG), a node x is transitively data dependent on a node y if there exists a sequence of nodes, y0 = y, y1, y2, . . . ,yn = x, in the control flow graph such that nP2 and yj is data dependent on yj_1 for all j, 1 6 j 6 n. The input node is a node i at which the data that may be controlled by an external user is accessed. Thus, input nodes include all the nodes which access untrusted data from direct input sources, such as HTTP request parameters, HTTP headers, and cookies; and indirect input sources, such as session variables, persistent objects, and database records. Direct input sources are known as sources of reflected XSS. Indirect input sources are known as sources of stored XSS.

```
public class Login extends HttpServlet {
 public void doGet ( HttpServletRequest req, HttpServletResponse resp ) {
1 String memID = req.getParameter("id") ;
2 String pwd = req.getParameter("paasword") ;
3 String html = "<HTML><BODY><h1>" ;
3    if(LoginValid) {
5      String name = rs.getString("LastName") ;
6      html += "Welcome" + name + "! </h1>" ;
```



```
    }
    else {
7       memID = "S123456" ;
8       html += "Welcome" + name + "! </h1>" ;
    }
9   memID = "S123456" ;
10  html += "<input type = 'hidden' name= 'member_id' value= ' " + memID + " '> " ;
11  out.print(html);
                                                    . . . . .
    }
```

Fig.1 Code snippet of a sample Login Servlet Program

A node o is called an HTML output node if o produces an HTML response output. We define the HTML output node o as a potentially vulnerable output node (pv-out) if o satisfies at least one of the following conditions:

(1) o is also an input node (e.g., out.print(req.getParameter(''input''))).

(2) o is data dependent on an input node i.

(3) o is transitively data dependent on an input node i.

A node in a CFG represents one program statement and we shall use the term ''node'' and ''statement'' interchangeably depending on the context. Based on the above definitions, we have implemented the identification of pv-outs by tracking the flow of untrusted data between input nodes and HTML output nodes[5]. We made use of this previous work to extract the potential XSSV information from a given program and pass the extracted information to the next phase.

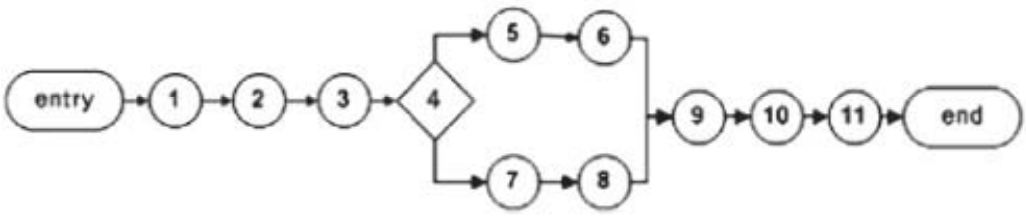

Fig.2 CFG of the program

2) XSS Vulnerability Removal: This phase contains two major steps—HTML context discovery and secure source code replacement. HTML context discovery identifies the statements at which the untrusted data referenced in an HTML output statement can be escaped. Then, the structure is considered and identifies HTML context. Based on the identified context, the appropriate escaping mechanism is chosen. Secure source code replacement generates secure code structures using ESAPI's escaping APIs as replacements for original code structures.

HTML Context Discovery

When an HTML output statement is identified as vulnerable, the untrusted data referenced in that statement must be escaped according to the HTML context the referenced data is in.

For each pv-out, first find the statements at which the untrusted data can be properly escaped. Such statements are called as escaping statements and the nodes representing them in a CFG as escaping nodes. Then, the HTML context is identified by analyzing the HTML document structure surrounding the escaping statements.

Let o be a pv-out in the CFG of a program. The following three conditions ensure that there is always an escaping statement for each pv-out.

1) If o satisfies the first condition of pv-out node definition, the node o is marked as *escape_stmt* and the method that retrieves the untrusted data (e.g., *req.getParameter(‘‘ input’’))* as *to_be_ escaped*.

2) If o satisfies the second condition of pv-out node definition, there is an input node i and a variable v defined in i and used in o. The node o is marked as *escape_stmt* and node vas *to_be_escaped*.

3) If o satisfies the third condition of pv-out definition, there is at least one sequence of nodes, {i = x0, x1,. . .,xn = o}, such that o is transitively data dependent on an input node i.

After an escaping statement is identified for a pv-out, the scheme examines any raw string data used in the escaping node and applies pattern matching to extract any identifiers found in thosedata that can be identified as HTML document structure. For pattern matching, an HTML pattern library is used which stores the patterns of HTML document structures. The document patterns are defined according to HTML 4.01 specification from W3C recommendation. Any document structure extracted is matched against those patterns from the library and identified with an HTMLcontext. If no identifier is found or the context is not recognized,it continues to explore two entities: (1) the nodes on which the escaping node is (transitively) data dependent and (2) the HTMLoutput nodes surrounding the escaping node and the nodes on which those output nodes are (transitively) data dependent; and analyze the raw string data found in those nodes in order. Once the context is identified for the untrusted data referenced in the escaping node, the appropriate escaping mechanism for the variable containing the untrusted data or the method which accesses untrusted data is determined based on the XSS prevention rules.

For example, if the HTML context is identified as HTML element, according to Rule#1, HTML entity escaping is required. If the algorithm cannot identify the HTML context until a preset timeout or there is no further node to explore, it assumes that the case belongs to Rule#0. Therefore, this algorithm ensures that any untrusted data referenced in an HTML output statement shall be secured by applying one of the XSS prevention rules.

Secure Source Code Replacement

In this phase, first we identify the appropriate escaping API *escape_api*from ESAPIlibrary. This corresponds to the required escaping mechanism of the variable or the method marked as *to_be_escaped* in *escape_stmt*. For example, if the required escaping mechanismis HTML entity escaping, the appropriate API to use is ESAPI.encoder().encodeForHTML(). Then modify *escape_stmt* by wrapping the object marked as *to_be_escaped*with *escape_api*. Finally we remove (comment out) the original statement and insert the modified statement into the same code location.

As an input, the information of escape_stmts, ie, *to_be_escapeds*, and *to_be_escapeds’*escaping rules and corresponding escaping methods are recieved. First, it declares the required ESAPI packages above the class declaration statement of the input program. Next, for each *escape_stmt* in each pv-out o, the untrusted data referenced in *escape_ stmt* is wrapped with appropriate escaping APIs using the following threesteps:

1) Identify the appropriate escaping API escape_api from ESAPI library that corresponds to the required escaping mechanism of the variable or the method marked as *to_be_escaped*in *escape_stmt*. For example, if the required escaping mechanism is HTML entity escaping, the appropriate API to use is ESAPI.encoder().encodeForHTML().

2) Modify *escape_stmt* by wrapping the object marked as *to_be_escaped* with *escape_api*.

3) Remove (comment out) the original statement and insert the modified statement into the same code location.

If the pv-out corresponds to Rule#0 (i.e., either the identified HTML context does not belong to the contexts stated in Rule#1–Rule#5 or no context could be identified), the scheme provides two options to user. If the lenient option is chosen, it reports the corresponding XSSV information to user, requests the appropriate sanitization/escaping scheme, and sets the user's input as *escape_api*. If the strict option is chosen, it sets a default sanitization method which returns an empty string as *escape_api*. Secure source code replacement is then performed in the same way as the above steps using this escaping API *escape_api*.

```
1   <HTML>
2   <script>  var x = 'untrusted_data1'  </script>
3   <BODY>
4   <img src =  'untrusted_data2' />
5   <p style = "color : untrusted_data3" >
6   untrusted_data4
7   </p>
8   <a href=  'untrusted_data5' > link </a>
9   <input type='hidden' name= 'id' value = 'untrusted_data6' />
10  </BODY>
```

Fig.3 Example of a vulnerable web page generated by a Java Servlet program.

```
import org.owasp.esapi.ESAPI;
public class Login extends HttpServelet  {
String memID = req.getParameter("id");
String pwd = req.getParameter("password");
String html = "<HTML> <BODY><h1>" ;

 If (loginValid)  {
    String name = rs.getString("LastName");
    // html += "Welcome"+  name + "! </h1>" ;
       Html += "Welcome" +
            ESAPI.encoder().encodeForHTML(name)  + "!   </h1>" ;
 }

 Else {
    memID = "S123" ;
    html +=  "Welcome Guest..</h1>"  ;
```



```
}

memID = memID.substring(0, 7) ;

// html += "<input type = 'hidden' name= 'member_id'

                                value= ' " + memID + " '> " ;

html += "<input type = 'hidden' name= 'member_id' value= ' " +

        ESAPI.encoder().encodeForHTMLAttribute(memID ) + " '> " ;

out.print(html);
```

Final Report
Login Servlet

pv-out	*input*	*escaped data*	*escaping mech*	*escaping stmt*	*comment*
Line 11	Line 1	memID	HTML attribute	Line 10	modified
Line 11	Line 5	name	HTML entity	Line 6	modified
. . .					

Fig.4 Login Servlet program secured with ESAPI's security APIs and the report produced by the algorithm.

For example, in Fig. 3, the untrusted data at line 4 is referenced in a complex HTML context which does not belong to any of the contexts stated inRule#1–Rule#5. For such case, this algorithm in strict mode performs the following:

<imgsrc= '<%=saferXSS.emptyStrAPI(untrusted_variable)%>'/>.

The resulting HTML output is:

<imgsrc= ''/>.

Therefore, our scheme in strict mode shall produce unintended HTML outputs for the cases belonging to Rule#0. However, such cases always involve high security risks and no escaping or sanitization is often possible to avoid the risks. Hence, removal of all XSSVs from input programs is fully automated by the above scheme. And code modification required is very minimal because only objects containing untrusted data are wrapped with escaping API calls. To facilitate software maintenance, this method feedbacks the source line numbers of modified statements, input statements, and pv-outs; and the escaped data and its associated escaping mechanism used to user. For the Login Servlet program in Fig. 1, the algorithm will produce the output as shown in Fig. 4. The modified statements are shown in bold.

IV. Conclusion

In this paper, we presented a three phase approach, for detecting and removing potential XSS vulnerabilities in web application. During the first phase, canonicalization is performed, in which input strings are reduced down to its simplest form. In the second phase input validation is performed. In the third phase of escaping, HTML contexts are identified and corresponding escaping mechanisms are done. Then, it performs source code generation and replacement to secure potentially vulnerable statements.

We presented that, the proposed approach is fully focused on detecting and removing XSSVs with minimal user intervention.

In future work, we intend to enhance our scheme so as to provide extended security with the integration of encryption mechanisms. Also we can extend the scheme with authentication features in order to support secure login.

References

1. ESAPI, OWASP Enterprise Security API, 2009. <http://www.owasp.org/index.php/ESAPI#tab=Project_Details> (accessed February 2010).
2. J.H. Hayes, A.J. Offutt, Input validation analysis and testing, Empirical Softw.Eng. 11 (4) (2006) 493–522.
3. Lwin Khin Shar, Hee Beng Kuan Tan "Automated removal of cross site scripting vulnerabilities in web applications".
4. H. Liu, H.B.K. Tan, Testing input validation in web applications through automated model recovery, J. Syst. Softw. 81 (2) (2008) 222–233.
5. L.K. Shar, H.B.K. Tan, Auditing the defense against cross site scripting in web applications, in: Proceedings of the 5th International Conference on Security and Cryptography (SECRYPT'10), 2010, pp. 505–511.
6. W3C, 1999, HTML 4.01 Specification. http://www.w3.org/TR/html401/(accessed April 2010).
7. W3C, 2002, XHTML 1.0 Specification. http://www.w3.org/TR/xhtml1/(accessed August 2011).
8. V.B. Livshits, M.S. Lam, Finding security errors in Java programs with staticanalysis, in: Proceedings of the 14th Usenix Security Symposium (USENIXSecurity'05), 2005, pp. 271–286.
9. RSnake, XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html(accessed March 2010).
10. CWE/SANS. Top 25 most dangerous software errors [accessed20.03.11], http://cwe.mitre.org/top25/; 2010.
11. OWASP, November 2009, OWASP Top Ten project 2010. <http://www.owasp.org> (accessed January 2010).
12. H. Liu, H.B.K. Tan, Covering code behavior on input validation in functional testing, Inform. Softw. Technol. 51 (2) (2009) 546–553.
13. M.S. Lam, M. Martin, B. Livshits, J. Whaley, Securing web applications with static and dynamic information flow tracking, in: Proceedings of the ACMSIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, 2008, pp. 3–12.
14. G. Edward Suh, Jaewook Lee, SrinivasDevadas "Secure Program Execution via Dynamic Information Flow Tracking".
15. T. Jim, N. Swamy, M. Hicks, Defeating script injection attacks with browser enforced embedded policies, in: Proceedings of the 16th International Conference on World Wide Web (WWW'07), 2007, pp. 601–610.
16. E. Kirda, C. Kruegel, G. Vigna, N. Jovanovic, Client-side cross-site scripting protection, Comput. Security 28 (2009) 592–604.
17. Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: Proceedings of the 15th USENIX Security Symposium (USENIXSecurity'06), 2006, pp. 179–192.
18. N. Jovanovic, C. Kruegel, E. Kirda, Pixy: a static analysis tool for detecting web application vulnerabilities, in: Proceedings of the IEEE Symposium on Securityand Privacy (S&P'06), 2006, pp. 258–263.
19. OWASP, June 2010, XSS (Cross Site Scripting) Prevention Cheat Sheet. <http://www.owasp.org/index.php/

