AUDITING THE DEFENSE AGAINST CROSS SITE SCRIPTING IN WEB APPLICATIONS

Lwin Khin Shar and Hee Beng Kuan Tan
School of Electrical and Electronic Engineering, Block S2, Nanyang Technological University
Nanyang Avenue 639798, Singapore, Republic of Singapore

Keywords: Cross Site Scripting, Static Analysis, Code Auditing, Input Validation and Filtering.

Abstract: Majority attacks to web applications today are mainly carried out through input manipulation in order to cause unintended actions of these applications. These attacks exploit the weaknesses of web applications in preventing the manipulation of inputs. Among these attacks, cross site scripting attack -- malicious input is submitted to perform unintended actions on a HTML response page -- is a common type of attacks. This paper proposes an approach for thorough auditing of code to defend against cross site scripting attack. Based on the possible methods of implementing defenses against cross site scripting attack, the approach extracts all such defenses implemented in code so that developers, testers or auditors could check the extracted output to examine its adequacy. We have also evaluated the feasibility and effectiveness of the proposed approach by applying it to audit a set of real-world applications.

1 INTRODUCTION

Recent reports on security attacks to web applications consistently showed that many successful attacks are carried out by illegal input manipulation. Cross-site scripting (XSS) attack is one such common attack as the attackers inject carefully crafted malicious scripts through the input parameters of client-side web pages in order to cause unintended actions of the web applications and achieve the attackers' purpose. Injected scripts can be all kinds of client-side scripts such as JavaScript, ActionScript, and VBScript.

XSS attacks can cause web applications to perform unintended actions such as exposing clients' confidential information to the attacker. The underlying problem is because those applications accept inputs from user and use those inputs in output operations such as display and database updates, without proper sanitization. XSS attacks may come from non-persistent input sources such as form-fields and URLs. This form of attack is known as reflected XSS. Or they may come from persistent input sources such as database records and persistent beans. This form of attack is known as stored XSS. Therefore, to avoid security violations, it is important to guard against all kinds of inputs which can be manipulated by the attackers. XSS attacks could be conducted in numerous ways by injecting a wide variety of HTML tags (e.g., `<script> hack(); </script>`) and attributes (e.g., `<b onmouseover = alert (document.cookie) >`). In order to circumvent most sanitization schemes used by web applications, more sophisticated XSS attacks could also be conducted by using various character encoding schemes. For example, using Base64 encoding scheme, `<script>hack(); </script>` can be replaced with `PHNjcmlwdD5oYWNrKCk7PC9zY3JpcHQ+Cg==`. As such, it is important to ensure that sanitization schemes used by web applications cover all such XSS attack vectors.

Basically, most web applications implement two type of sanitization mechanisms to defend against XSS attacks: (1) input validation; (2) input filtering. Input validation checks the user input against the user interface specifications whereas input filtering removes, replaces, or escape potentially dangerous characters such as "<". But sanitization mechanisms often fail in protecting the applications because either it is hard to sanitize all complete set of malicious scripts or the implementation is inadequate. In some applications, users are allowed to input HTML tags. It makes the escaping routines irrelevant. As such, it is important to provide

Khin Shar L. and Beng Kuan Tan H. (2010).
AUDITING THE DEFENSE AGAINST CROSS SITE SCRIPTING IN WEB APPLICATIONS.
In *Proceedings of the International Conference on Security and Cryptography*, pages 505-511
DOI: 10.5220/0002963905050511

software testers or developers with comprehensive information about the security mechanisms implemented in code in order to be able to check their adequacy for XSS defense.

To date, existing approaches mainly focus on dynamically preventing potential XSS attacks or statically identifying potential XSS vulnerabilities in web applications. In most cases, software project teams prefer their applications to be free from vulnerability risks and hence, they may want to perform software vulnerability audits from time to time. To the best of our knowledge, no approach has focused on providing the software testers or developers with necessary information about XSS defense features implemented in code to facilitate such audits. Hence, in this paper, we propose a novel code auditing approach that focuses on automatically extracting all possible XSS defenses implemented in code. The extraction is based on the possible coding patterns for implementing XSS defense. From the extracted output, one can examine its adequacy and identify the potential risks.

2 THEORY FOR EXTRACTING XSS DEFENSE FEATURES FROM CODE

The proposed theory is built on modeling of the possible code patterns of sanitization methods that prevent illegal input manipulation: (1) input validation; (2) input filtering.

In this paper, the basic definitions of interprocedural control flow graph (CFG) are adopted from Sinha et. al (2001). The following gives further definitions used in this paper.

In a CFG, a node x **transitively references** to a variable v defined/submitted at node y if there is a sequence of nodes, $y = x_0, x_1, \ldots, x_n = x$, and a sequence of variables $v = v_0, v_1, \ldots, v_n$, such that $n \geq 1$, for each j, $1 \leq j \leq n$, x_j defines variable v_j and references to variable v_{j-1}, and any path from x_{j-1} to x_j that does not repeat any loop is a definition-clear path with respect to $\{v_{j-1}\}$.

In a web application, a node u at which an input submitted by user can be referenced and u dominates all nodes w at which the input can also be referenced, is called **input node**. These inputs include both persistent and non-persistent types of inputs. Variables defined/submitted at an input node u are called **input variables** of u. For example, Figure 1 shows a JSP code snippet of Guestbook SignIn page and Figure 2 shows its CFG. In this CFG, node 1 and 6 are input nodes; *guestname* and *rs* are input variables of node 1 and 6 respectively. A variable o referenced in a node is called **potentially vulnerable variable** (**pv-variable**) if o is submitted at an input node or o is defined at a node that transitively references to input variable/s submitted at input node/s. In Figure 2, *guestname* and *name* are pv-variables of the node 5 and 11 respectively.

A program path from an input node u to the exit node is called a **prime input path** of u if it does not repeat any loop and pass through u again. In a web application program, a statement that references at least one pv-variable and performs a HTML response operation is called a **HTML operation statement** (**html-o-statement**). In a CFG, the node that represents this statement is called a **HTML operation node** (**html-o-node**). In Figure 1, statements at line 5 and 11 are html-o-statements, and in Figure 2, node 5 and 11 are html-o-nodes.

2.1 Extraction of Defense through Input Validation

Let k be a html-o-node in a CFG. Throughout this section, we shall address k as a html-o-node. To prevent illegal manipulation of inputs referenced at k, it is common that predicate nodes are used to ensure that only inputs with legitimate values are allowed to be operated at k. Next, we shall define a terminology to characterize such node pattern.

A predicate node d is called an **input validation node** (**iv-node**) for k if the following properties hold:

1) Both k and d transitively reference to a common input variable of an input node u.
2) A prime input path m of u, that follows one branch of d, passes through k; and no prime input path m' of u, that follows the other branch of d, passes through k.

In Figure 2, both html-o-node, node 5, and predicate node, node 2, transitively reference to the input variable, *guestname*, of the input node, node 1. Prime input paths of node 1 starting with (1, 2, 3, 4, 5, 6, …), which follow the branch (2, 3) of node 2, pass through node 5. No prime input path that follows the other branch (2, 6) of node 2 passes through node 5. Hence, node 2 is an iv-node for node 5. Likewise, the while-predicate node, node 8 is also an iv-node for the html-o-node, node 11.

Next, we shall next formalize the detection of unprotected html-o-nodes in Property 1, which can be proved directly from its definition.

Property 1 – Unprotected Html-O-Node. If there is no input validation node for k such that both transitively reference to a common input variable v of an input node u, then k is unprotected from XSS attack through illegal manipulation of v.

A path through a CFG is called a **1-path** if it does not repeat any loop. Let Ω be the set of 1-paths through the CFG. The partition of 1-paths in Ω, such that paths which contain the same set of iv-nodes for k and follow the same branch at each of these nodes are put in the same class, is called the **input validation partition** (**iv-partition**) for k. In the CFG shown in Figure 2, {(entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 8, end), (entry, 1, 2, 3, 4, 5, 6, 7, 8, end), (entry, 1, 2, 6, 7, 8, 9, 10, 11, 8, end), (entry, 1, 2, 6, 7, 8, end)} is the set of 1-paths. Hence, in this CFG, {{(entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 8, end), (entry, 1, 2, 3, 4, 5, 6, 7, 8, end)}, {(entry, 1, 2, 6, 7, 8, 9, 10, 11, 8, end), (entry, 1, 2, 6, 7, 8, end)}} is the iv-partition for the html-o-node, node 5; and {{(entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 8, end), (entry, 1, 2, 6, 7, 8, 9, 10, 11, 8, end)}, {(entry, 1, 2, 3, 4, 5, 6, 7, 8, end), (entry, 1, 2, 6, 7, 8, end)}} is the iv-partition for the html-o-node, node 11.

Next, we shall formalize the computation of valid and invalid conditions for a html-o-node in Property 2 and 3, which can be proved directly from the definitions of input validation node and input validation partition.

Property 2 – Invalid Input Condition. IVC_k = {C | X is a class in the iv-partition of k such that there is a path in X which passes through some iv-nodes for k but does not pass through k; and C = the conjunction of all the branch conditions of branches at iv-nodes for k that any path in X follows} is the set of invalid input conditions for k.

Property 3 – Valid Input Condition. VC_k = {C | X is a class in the iv-partition of k such that there is a path in X which passes through some iv-nodes for k and also passes through k; and C = the conjunction of all the branch conditions of branches at iv-nodes for k that any path in X follows} is the set of valid input conditions for k.

Note that for each class X stated in Property 2 and 3, the conjunction of all the branch conditions of branches at iv-nodes for k that any path in X follows is identical.

From the iv-partition computed for node 5, we compute according to Property 2 and 3 that IVC = {(!(guestname!=null && guestname.length()<20))} and VC={(guestname!= null && guestname.length() <20)} are the sets of invalid and valid input conditions for this html-o-node. IVC and VC for the html-o-node, node 11 can also be computed in a similar way.

```
1. String guestname = request.getParameter("guestname");
2. if(guestname !=null && guestname.length() < 20) {
3.     stmt.executeUpdate("insert into guestbook values (" +
          guestname + ")" );
4.     guestname.replace("<", "&lt;");
5.     out.println("Welcome "+ guestname);
   }
6. ResultSet rs = stmt.executeQuery("select * from guestbook");
7. out.println("Guest List:")
8. while(rs.next()) {
9.     String name = rs.getString("guestname");
10.    name.replace("<", "&lt;");
11.    out.println(name);
   }
```

Figure 1: JSP code snippet for Guestbook SignIn.

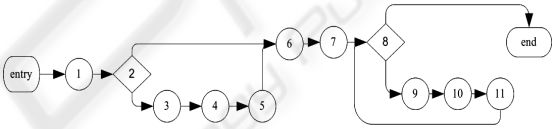

Figure 2: The CFG of the JSP code snippet.

2.2 Extraction of Defense through Input Filtering

For defending against XSS attack through a html-o-statement, another alternative is to remove, replace, or escape any dangerous characters from inputs that may define the values of pv-variables referenced at this html-o-statement. This filtering approach is generally carried out by a sequence of statements that may influence the values of all pv-variables of a html-o-statement. We shall formalize a terminology to characterize such coding pattern.

Potentially Vulnerable Input Filter (**PV-Input Filter**) for a html-o-statement k in a program P is a sequence F of following statements according to their order in P:

1) All the statements at which a pv-variable of k is defined/submitted.
2) Statements on which statements in F are data dependent.
3) Statements on which statements in F are control dependent such that k is not transitively control dependent on these statements.

For example, the sequence of following statements forms the pv-input filter for the html-o-statement at line 5 shown in Figure 1:

```
1. String guestname=
     request.getParameter("guestname");
4. guestname.replace("<", "&lt;");
```

Note that if a variable is not a pv-variable, it is impossible to manipulate its value to perform an XSS attack. This is because all possible values of non pv-variables can only be defined by the programmer.

3 PROPOSED AUDITING APPROACH

Based on the theory discussed, we propose a code auditing approach for auditing the XSS defense features implemented in web applications through the following two major steps:

Step 1. Extract XSS defenses through input validation and input filtering automatically from code.

Step 2. Examine the extracted output to determine its adequacy.

The first step can be fully automated using interprocedural control flow and data flow between input nodes and html-o-nodes. The second step in the proposed approach examines whether the input validation and input filtering implemented for each html-o-statement is sufficient for defending it against XSS attack. This step is to be carried out manually by examining the codes extracted from Step 1 as explained in the followings.

In examining the input validation codes, the unprotected html-o-statements from some input variables are highly vulnerable to XSS attack through those variables. Hence, they require special attention in examining their input filtering codes. Next, invalid and valid input conditions of each remaining html-o-statement can be checked against required input formats or specifications to examine the adequacy for XSS defense. In examining the input filters, one needs to first examine how the pv-input filter for each html-o-statement is contributed to the XSS defense, and next determine its adequacy for XSS defense based on experience or using an up-to-date set of coding guidelines that avoid XSS vulnerability (e.g., XSS prevention rules from OWASP (2010)). Program slicing can also be used to aid the comprehension of invalid and valid input conditions, and pv-input filter through slicing on associated variables and statements.

Note that in addition to the statements related to the XSS defense, the extracted output may also include some other statements that do not serve any security purpose. Moreover, a pv-variable may not be vulnerable at all if it only references to input whose value can be defined only by the programmer (i.e., the database records or session variables read by the program may not contain user input data). Hence, one must study and comprehend the extracted statements to identify the portion that serves for XSS defense.

4 EVALUATION

4.1 Prototype Tool

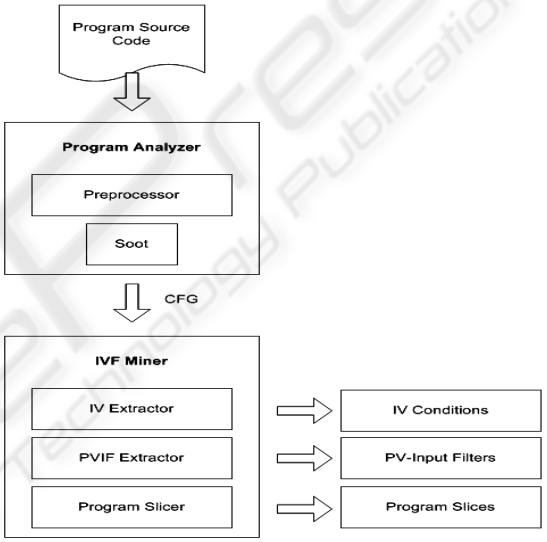

Figure 3: The architecture of XSSDE.

We have prototyped a tool called XSSDE (XSS Defense Extractor) through the use of Soot (Soot, 2008). The tool fully automates the extraction process in Step 1. Its architecture is shown in Figure 3. It consists of two major modules: a program analyzer, and an input validation and filtering miner (IVF miner). Program analyzer uses Soot's APIs to analyze Java programs. It takes the class files of a Java program as input and builds the CFG of the program for control flow and data flow analysis. IVF miner includes three sub-modules: an input validation extractor (IV extractor), a pv-input filter extractor (PVIF extractor), and a program slicer. For each html-o-node in a CFG, IV extractor extracts the set of valid and invalid input conditions of the html-o-node and the set of ordered pairs of input variable and input node such that the html-o-node is unprotected from XSS attack through each input variable. Similarly, for each html-o-statement in a

program, PVIF extractor extracts the pv-input filter for the html-o-statement. All the information extracted is printed in a report. Program slicer is a utility tool used for further comprehension of the extracted information in order to determine the adequacy of XSS defense implemented. It aids the comprehension of variables and statements involved in the information extracted through extracting a program slice according to a given criterion.

4.2 Experiment

To evaluate the proposed approach, we have compared our approach with Livshits and Lam's approach (Livshits and Lam, 2005). In the experiment, two postgraduate students applied our approach and two other students applied Livshits and Lam's approach on five open source systems obtained from GotoCode web site (GotoCode, n.d.). We have requested the two students to apply Livshits and Lam's approach analytically since their prototype tool is not available to us. In applying our proposed approach, the two students performed Step 1 through the use of the prototype tool XSSDE. As discussed in section 3, Step 2 is to examine the extracted output from Step 1 manually in order to determine its adequacy for XSS defense. To facilitate this step, we have provided the two students with a set of coding guidelines for XSS prevention which was obtained from the two reliable sources (OWASP, 2009; OWASP, 2010). In this evaluation, we aim to address the following two research questions:

RQ1. Is the code auditing approach really necessary despite existing static approaches?

RQ2. Is our approach effective in extracting all the XSS defense features implemented in programs and feasible in examining them to determine their adequacy?

To answer the questions, we analyzed the statistics of the experiment results shown in Table 1.

RQ1. Livshits and Lam's approach accounted for 27% (158/582) false positive rate (it produced 29% (12/41) false positive rate in their own evaluation in (Livshits and Lam, 2005)). Using a set of coding guidelines that avoid XSS vulnerability, both students applied the proposed approach and identified all the vulnerable cases accurately without any false positives. It was observed that majority of the actual vulnerable html-o-statements (424) arises from imperfect input filters used in the experimented systems. In this experiment, Livshits and Lam's approach did not produce any false negative cases as we assumed that a complete vulnerability specification is provided by user. In general, this is not possible. At least in this experiment, we observed that the compared approach produces significant false positives and our proposed approach really helps in identifying actual XSS vulnerabilities. To further study the usefulness of the proposed approach, we plan to compare it with more recent static analysis approaches which may produce lower false positive cases.

Table 1: Statistics of evaluation results.

Systems (LOCs)	Proposed Approach			Livshits and Lam's Approach		
	Total VC, IVC and pv-input filter extracted in LOCs	#vulnerable html-o-statements reported	#False positives	Data flow traces extracted in LOCs	#vulnerable html-o-statements reported	#False positives
Emp. Dir. (3035)	449	44	0	322	54	10
Bookstore (9551)	1608	143	0	1159	187	44
Events (3818)	497	37	0	356	65	28
Classifieds (5745)	732	63	0	575	89	26
Portal (8803)	1724	143	0	1202	194	51
Total (30952)	**4940**	**424**	0	**3590**	**582**	**158**

RQ2. Although Livshits and Lam's approach extracts data flow traces associated with each vulnerability point, clearly it missed out essential XSS defense features implemented. In contrast, we have confirmed that the proposed approach completely extracted all XSS defense features implemented. Of the total codes written, Livshits and Lam's approach and our proposed approach extracted 11.6% (3590/30952) and 16% (4940/30952) respectively. Though the proposed approach extracted more lines of codes (LOCs) including those that are not related to XSS defense, we observed that manual examination process is still very much feasible for these sizes of the applications experimented. In future work, we plan to test the feasibility of the proposed approach based on larger-sized web applications.

5 RELATED WORK

Many security researches to prevent XSS attacks are mostly dynamic approaches. There are two types of dynamic approaches; server-side detection and client-side detection. Server-side detection approaches generally set up a proxy between client and server, and check whether the input parameters in HTML request contain any malicious scripts (Kruegel and Vigna, 2003), or the input parameters are used in the scripts of HTML response (Ismail et al., 2004; Johns et al., 2008; Bisht and

Venkatakrishnan, 2008), or the scripts of HTML responses are actually intended by the application (Johns et al., 2008; Bisht and Venkatakrishnan, 2008).

Client-side detection approaches are proposed mainly to be used and deployed by the clients. Beep (Jim, Swamy and Hicks, 2007) provides the client's browser with security policies (e.g., a set of legitimate scripts and HTML content where malicious scripts may occur) and modifies the browser such that it is capable of preventing illegitimate scripts from being executed. Noxes (Kirda et al., 2009) acts as a personal firewall which detects potential XSS attacks based on filter rules generated automatically or manually by clients. Filter rules are basically the whitelist and blacklist of url links. Blueprint (Louw and Venkatakrishnan, 2009) takes over the browser's parsing decision on untrusted HTML contents to ensure that the resulting parse trees contain no script. Blueprint then embeds model interpreter in corresponding web pages such that the client's browser could reconstruct those parse trees. All the above approaches incur runtime overheads due to the use of proxy and interception of HTTP traffic. Some also introduce technical difficulties in setting up, configuring and deploying of their systems. Hence, they may not be useful for novice users and some who prioritize performance.

In software testing, there are input validation testing (IVT) approaches that could uncover some XSS vulnerabilities in the programs (Hayes and Offutt, 2006; Li et al, 2007; Liu and Tan, 2009). These approaches generate test cases with valid and invalid inputs either identified from specification (Hayes and Offutt, 2006; Li et al, 2007) or extracted from code (Liu and Tan, 2009). However, IVT approaches are not suitable for checking the adequacy of the input validation codes. Since even imperfect or incorrect input validation codes could detect some malicious inputs, IVT approaches may not reveal the weaknesses of those validation codes.

Works on the identification of XSS vulnerabilities in web applications are mainly based on static analysis techniques. They check whether tainted data reaches sensitive program points without being properly sanitized, using flow-sensitive, interprocedural and context-sensitive data flow analysis (Livshits and Lam, 2005; Jovanovic et al., 2006; Xie and Aiken, 2006). Pixy (Jovanovic et al., 2006) enhances the accuracy by aliasing. However, these approaches do not check the custom sanitization functions (either they conservatively assume that all those functions return unsafe data or user explicitly states the correctness of the functions). As a result, even Pixy produced 50% false positive rate (Jovanovic et at., 2006).

There are static analysis approaches which check the adequacy of sanitization functions (Balzarotti et al., 2008; Wassermann and Su, 2008). Based on string analysis techniques, they use blacklist comparison approach to check whether tainted strings may still contain the blacklisted characters at sensitive program points after passing through sanitization functions. However, Wassermann and Su's current tool cannot handle complex codes and some of PHP's string functions (Wassermann and Su, 2008). Saner (Balzarotti et al., 2008) requires dynamic analysis to identify the false positives produced by static analysis phase.

The similarity between the above static approaches and our approach is that the vulnerabilities identified by them may serve as pointers for auditing the XSS defense features implemented in the programs. However, they do not extract all the statements that contribute to the XSS defenses. The results of those static approaches show that false positives are common due to their conservative nature. Most of the above approaches learnt those false positive cases through manual inspection on identified vulnerabilities. Hence, it supports the main point of our proposed approach that manual code auditing is necessary. However, manual inspection on the whole system would be labor-intensive, costly, and error-prone. Therefore, we propose the code auditing approach that focuses on comprehensively extracting the security mechanisms implemented in code in order to facilitate the manual code audits.

In fact, the proposed approach and the above static approaches compliment each other. Because the tools implemented by most of the above static approaches automate the potential XSS vulnerability identification process and our XSSDE tool automates the XSS defenses extraction process, they could be used together for more efficient code auditing process.

6 CONCLUSIONS

Cross site scripting is one of the most common security threats to web applications. Although current dynamic analysis approaches are generally effective, users may not want to overcome their technical issues such as configuration, deployment and runtime overheads. Moreover, they also do not identify the XSS vulnerabilities in the applications. Existing static analysis approaches also lack

accuracy in identifying XSS vulnerabilities, and therefore they cannot avoid the code auditing requirement. Hence, we propose a novel approach for extracting XSS defense features implemented in code to facilitate both examination and auditing processes.

In future work, we plan to study on effective coding techniques for preventing XSS attack and generalize them as guidelines so that one could use it in examining the adequacy of extracted XSS defense features. We also plan to evaluate our approach against more recent or better approaches based on the experiments on both open source and industrial applications.

REFERENCES

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *S&P '08: Proceedings of the IEEE Symposium on Security and Privacy*, 387-401.

Bisht, P. and Venkatakrishnan, V. N. (2008). XSS-Guard: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA '08: Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 23-43.

GotoCode (n.d.). *Open source web applications*. Retrieved August 23, 2009, from http://www.gotocode.com

Hayes, J. H. and Offutt, J. (2006). Input validation analysis and testing. *Empirical Software Engineering*, 11, 493–522.

Ismail O., Eto M., Kadobayashi Y., and Yamaguchi S. (2004). A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *AINA '04: Proceedings of the 8th International Conference on Advanced Information Networking and Applications*, 145-151.

Jim, T., Swamy, N., and Hicks, M. (2007). Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th International conference on World Wide Web*, 601-610.

Johns, M., Engelmann, B., and Posegga, J. (2008). XSSDS: Server-side detection of cross-site scripting attacks. In *ACSAC '08: 2008 Annual Computer Security Applications Conference*, 335-344.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: a static analysis tool for detecting web application vulnerabilities. In *S&P '06: Proceedings of the IEEE Symposium on Security and Privacy*, 258-263.

Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N. (2009). Client-side cross-site scripting protection. *Computers & Security*, 28, 592-604.

Kruegel C. and Vigna G. (2003). Anomaly detection of web-based attacks. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communication Security*, 251–261.

Li, N., Wu, J., Jin, M. Z., and Liu, C. (2007). Web application model recovery for user input validation testing. In *ICSEA '07: 2nd International Conference on Software Engineering Advances*, 85-90.

Liu, H. and Tan, H. B. K. (2009). Covering code behavior on input validation in functional testing. *Information and Software Technology*, 51, 546-553.

Livshits V. B. and Lam M. S. (2005). Finding security errors in Java programs with static analysis. In *USENIX Security '05: Proceedings of the 14th Usenix Security Symposium*, 271-286.

Louw, M. T., Venkatakrishnan, V. N. (2009). Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *S&P '09: Proceedings of the 30th IEEE Symposium on Security and Privacy*, 331-346.

OWASP (May 14, 2009). *Reviewing Code for Cross-site scripting*. Retrieved January 10, 2010, from http://www.owasp.org/index.php/Reviewing_Code_for_Cross-site_scripting

OWASP (January 6, 2010). *XSS Prevention Cheat Sheet*. Retrieved January 10, 2010, from http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

Sinha, S., Harrold M. J., and Rothermel G. (2001). Interprocedural control dependence. *ACM Transactions on Software Engineering and Methodology*, 10, 209-254.

Soot (2008). *Soot: a Java Optimization Framework*. Retrieved February 12, 2009, from http://www.sable.mcgill.ca/soot/

Wassermann, G. and Su, Z. (2008). Static detection of cross-site scripting vulnerabilities. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, 171-180.

Xie, Y. and Aiken, A. (2006). Static detection of security vulnerabilities in scripting languages. In *USENIX Security '06: Proceedings of the 15th USENIX Security Symposium*, 179-192.