Information and Software Technology 54 (2012) 467–478

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Automated removal of cross site scripting vulnerabilities in web applications

Lwin Khin Shar *, Hee Beng Kuan Tan

School of Electrical and Electronic Engineering, Block S2, Nanyang Technological University, Nanyang Avenue, Singapore 639798, Singapore

ARTICLE INFO

Article history:
Received 18 February 2011
Received in revised form 1 October 2011
Accepted 20 December 2011
Available online 28 December 2011

Keywords:
Cross site scripting
Injection vulnerability
Character escaping
Encoding
Web security
Automated bug fixing

ABSTRACT

Context: Cross site scripting (XSS) vulnerability is among the top web application vulnerabilities according to recent surveys. This vulnerability occurs when a web application uses inputs received from users in web pages without properly checking them. This allows an attacker to inject malicious scripts in web pages via such inputs such that the scripts perform malicious actions when a client visits the exploited web pages. Such an attack may cause serious security violations such as account hijacking and cookie theft. Current approaches to mitigate this problem mainly focus on effective detection of XSS vulnerabilities in the programs or prevention of real time XSS attacks. As more sophisticated attack vectors are being discovered, vulnerabilities if not removed could be exploited anytime.
Objective: To address this issue, this paper presents an approach for removing XSS vulnerabilities in web applications.
Method: Based on static analysis and pattern matching techniques, our approach identifies potential XSS vulnerabilities in program source code and secures them with appropriate escaping mechanisms which prevent input values from causing any script execution.
Results: We developed a tool, saferXSS, to implement the proposed approach. Using the tool, we evaluated the applicability and effectiveness of the proposed approach based on the experiments on five Java-based web applications.
Conclusion: Our evaluation has shown that the tool can be applied to real-world web applications and it automatically removed all the real XSS vulnerabilities in the test subjects.

1. Introduction

Recent reports from OWASP [1] and CWE/SANS [2] reveal that cross site scripting (XSS) is one of the most common and serious web security flaws. It is a type of code injection vulnerability that enables attackers to send malicious scripts to web clients. It occurs whenever a web application references user inputs in its HTML pages (i.e., web pages sent by the application to its clients) without properly validating them. An attacker may embed malicious scripts via such inputs in the application's HTML pages. When a client visits an exploited web page, the client's browser not being aware of the presence of malicious scripts shall execute all the scripts sent by the application resulting in a successful XSS attack. The malicious script used in an XSS attack can be any kind of client side scripts (e.g., HTML, JavaScript, VBScript, and Flash) that can be interpreted by web browsers. XSS attacks may cause severe security violations such as account hijacking, data theft, cookie theft and poisoning, web content manipulation, and denial of service [3].

Popular real world XSS vulnerabilities include *Microsoft .NET Framework 2.0*'s vulnerability reported in 2006 [5]. It was caused by improper validation of HTTP request data allowing the attackers to manipulate data in web pages. If any web application is implemented using this framework, attackers may launch a variety of XSS attacks depending on the nature of the application. Recently, the web site of a giant telecommunication company *Vodafone* has been reported to contain an XSS hole that allows a fake survey form to be injected using crafted URLs such as https://surveys.vodafone.com/Checkbox/Survey.aspx?`surveyid=1234==<script>stealData()</script>` [6]. Its flaw is that *Vodafone*'s Survey.aspx page does not filter malicious characters such as "<script>". Hence, an attacker may exploit this flaw and steal the sensitive information submitted by *Vodafone*'s clients via the injected survey form.

To mitigate the threats posed by XSS attacks, several solutions have been proposed. They can be classified into defensive coding practices [3,4,7], input validation and XSS testing techniques [19–22], vulnerability detection techniques [23–33], and attack prevention techniques [34–39]. However, apart from defensive coding practices, none of the current techniques focuses on removing the XSS vulnerabilities (XSSVs) that exist in program source code. On the other hand, defensive coding practices could in principal ensure that a program is free from XSSVs. The known

* Corresponding author.
E-mail addresses: shar0035@ntu.edu.sg (L.K. Shar), ibktan@ntu.edu.sg (H.B.K. Tan).

0950-5849/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2011.12.006

defensive coding approach for effective prevention of XSSVs is to escape all the user inputs used in HTML documents according to the contexts in which these inputs are referenced (e.g., *JavaScript* context if the input is used in a JavaScript code; *HTML element* context if the input is used in an HTML element). Escaping (a.k.a encoding) is the process of transforming the characters that have special meanings to a client-script interpreter into the representations such that the special meanings are removed [3,4,7]. However, this method if performed manually is prone to human errors and hard to be enforced in existing web applications. Therefore, automation of this task would be beneficial.

This paper proposes an automated approach that statically removes XSSVs from program source code. This approach is different from existing approaches which mainly concern with either locating XSSVs in program source code or preventing real time XSS attacks. The proposed method consists of two phases: (1) XSSV detection and (2) XSSV removal. XSSV detection phase identifies potential XSSVs in the program source code using static analysis. XSSV removal phase first determines the context of each user input referenced in the identified potential XSSVs. It then secures the potential XSSVs by applying the appropriate escaping methods using an escaping library provided by ESAPI [8]. We also developed a tool called *saferXSS* to automate the proposed approach. Both the approach and the tool are targeted to the web applications written in Java language due to the frequent occurrences of XSS issues in Java-based web applications. However, the proposed idea can be easily extended to fit the syntax of other programming languages. Using *saferXSS*, we conducted experiments on five applications. Results show that the approach was effective in securing all the XSSVs found in the test subjects.

The major contribution of this paper is that it proposes an approach to identify and remove the potential XSSVs in web applications, presents a tool that automates the proposed approach, and evaluates it based on the experiments on five open source web applications. The remainder of the paper is organized as follows. Section 2 provides the background information on XSS injection techniques, XSS prevention rules based on proper escaping methods, and ESAPI's security mechanisms. Section 3 proposes our XSS vulnerability detection and removal approach. Section 4 evaluates the proposed approach and Section 5 reviews the current techniques that mitigate XSS attacks. Section 6 concludes the paper.

2. Background

In this section, we provide information on XSS injection, XSS prevention rules proposed by OWASP [4], and escaping APIs provided by ESAPI [8] which can be used to enforce OWASP's XSS prevention rules in vulnerable web applications.

2.1. XSS injection

XSS can be classified into three types: stored, reflected, and DOM-based. Stored XSS hole appears when a web server program stores an unrestricted user input in a persistent data store such as database and then the program accesses and references the stored data in a web page viewed by different users. This type of XSS is commonly found in forums, blogs, and other social networking sites. Reflected XSS hole appears when a server program references an unchecked data accessed from incoming HTTP request parameters in an immediate web page sent back to the user. This type of XSS is commonly found in error messages and search results. Both of these XSS types result from improper handling of user inputs in server side scripts. By contrast, DOM-based XSS hole appears when a client side script itself references a user input dynamically obtained from the DOM (document object model) structure without proper validation and thus, malicious script injected via DOM-based XSS needs not appear on server programs [9]. In this paper, only the issues of stored and reflected XSS shall be further discussed.

Regardless of different types of XSS, code injection techniques are the same. In an HTML document which is a combination of data and code, user inputs are often referenced as data. As such inputs are the cause of XSS injection, we may also address them as **untrusted data** depending on the context. In an HTML document, there are many contexts in which user input may be referenced. Different client side interpreters may also be used for different types of contexts. Therefore, to prevent XSS injection, it is important to be aware of the context in which user input is referenced and understand how an XSS injection is possible in that context.

As an illustration, Fig. 1 shows an example of a vulnerable web page through which multiple XSS attacks can be conducted via the user inputs referenced. As shown in line 5–7, the web page references an untrusted data as a text paragraph (i.e., untrusted_data4). If the data is `<script>hack();</script>`, the HTML output becomes `<p style=...><script>hack();</script></p>`. The tag "<script>" indicates the beginning of a JavaScript code and invokes the JavaScript interpreter. As a result, although the intended document structure is a block of text, it becomes a block of JavaScript code causing the JavaScript interpreter to execute the malicious function "hack()". In this case, the user input is referenced in the context of *HTML element* and the special character "<" is used to illegally switch into a code (in this case, JavaScript) context where all kinds of malicious scripts can be executed; thereby, causing an XSS scenario. Fig. 1 also shows other HTML contexts where user inputs are commonly referenced: *HTML attribute* (line 9), *JavaScript* (line 2), *Cascading style sheets* or *CSS* (line 5), and *URL parameter* (line 8). Similar to the above *HTML element* context example, the fitting special characters may be used for switching into a code context. For example, the *HTML attribute* value at line nine references `untrusted_data6` within the opening and closing quotes. This context can be broken out by injecting the quote " ' " into the untrusted data. In the scenarios where untrusted data is not quoted at all, many characters such as [space], ";", ",", and "<" can be used for code switching. There are also complex HTML contexts where user inputs may be referenced. They are worst-case scenarios in which no special character is required to cause context switching. For example, as shown in line 4, the web page references `untrusted_data2` as a value for the *HTML attribute* `src`. If the data is `http://www.hackersite.com/hack.js`, the output becomes: `<img src= 'http://www.hackersite.com/hackv.js'/>`. In this case, a JavaScript file "hack.js" from a hacker's web site is introduced within the original *HTML attribute* context and no special character is used. Some HTML attributes such as `src` and `href`, event handlers such as `onclick` and `onload`, and complex functions such as `expression` and `eval` are such complex contexts. No user input should be directly referenced in these contexts as there are simply many possibilities to inject malicious scripts via such an input.

2.2. XSS prevention rules

As discussed in Section 2.1, XSS injection is generally achieved by switching to a code context illegally from a data context by using special characters (e.g., "'", """, ";", "<") which are significant to a targeted client side interpreter. OWASP [4] specified systematic XSS prevention rules to follow to ensure that any user input referenced in an HTML output is only treated as a data. The rules condition that appropriate escaping mechanism be applied to the user input according to the HTML context in which the input is referenced. Escaping disables the effect of special characters con-

```
<HTML>
<script> var x='untrusted_data1' </script>
<BODY>
<img src= 'untrusted_data2'/>
<p style= "color:untrusted_data3">
untrusted_data4
</p>
<a href= 'untrusted_data5'> link </a>
<input type='hidden' name='id' value='untrusted_data6'/>
</BODY>
```

Fig. 1. Example of a vulnerable web page generated by a Java Servlet program. Bold words show the locations at which an untrusted data is referenced.

tained in user input and prevents them from invoking client side interpreters. Therefore, as long as user inputs are to be referenced in typical HTML contexts as data, XSSVs can be completely avoided by following OWASP's rules and there is also no harm in escaping the referenced untrusted data even if the HTML output is not actually vulnerable.

Hence, our proposed approach is built on following the rules defined by OWASP [4] to remove the XSSVs in server programs. We shall briefly review these rules in this section. Interested readers may refer to OWASP [4] for more detail.

- *Rule#0*: Do not reference user inputs in any other cases except the ones defined in Rule#1–Rule#5. As discussed in Section 2.1, 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 in Rule#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 directly referenced in any other cases. This rule is the most important among all XSS prevention rules as it implies a whitelist approach (blacklist approaches are commonly known as weak and insufficient).
- *Rule#1*: Use *HTML entity escaping* for the untrusted data referenced in an *HTML element*. For example, `<body><div>htmlEscape(untrusted_data)</div></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 escaping rules 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 `onclick`. 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='htmlAttrEscape(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)'''>`, where "javascriptEscape()" is the JavaScript escaping method, conforms to this rule. In this paper, we shall only discuss XSS injection using JavaScript. 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, `<table style=''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, `<a href='http://www.site.com?name=urlEscape(untrusted_data)'>` and `<img src='http://www.site.com?imgid=urlEscape(untrusted_data)'>`, where "urlEscape()" is the URL escaping method, conform to this rule.

Note that the last two rules, Rule#6 and Rule#7, of OWASP [4] are omitted by our proposed approach because those rules are not related to escaping of user inputs in server programs. They also do not conflict with the above Rule#0–Rule#5 used by our approach.

2.3. ESAPI's escaping APIs

ESAPI [8] implemented the escaping APIs that can be used to enforce the above XSS prevention rules in vulnerable web applications. As these APIs are used in our proposed approach and our implementation, we shall briefly review the ESAPI project in the following.

ESAPI is a security project that facilitates users to enforce security in both developed and developing web applications. The 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. As such, with ESAPI, users have a choice of implementing any of these mechanisms in their applications.

To utilize the security controls provided by ESAPI, users must first install ESAPI project into their applications. The documentation on ESAPI installation and configuration procedures can be found in ESAPI package downloadable from `<code.google.com/p/owasp-esapi-java/>`. The Java documentation on ESAPI's escaping/encoding APIs can be found in `<owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/Encoder.html>`.

Once ESAPI is installed, a user could secure an untrusted data by wrapping it with a proper escaping API before it is referenced in an HTML output. The proper API is to be determined based on the corresponding HTML context and the XSS prevention rules from Section 2.2. For example, if the context is *HTML element*, Rule#1 applies and thus ESAPI's *HTML escaping* API is to be used as shown in the following: `<div>`**`ESAPI.encoder().encodeForHTML`**`(untrusted_data)</div>`.

It ensures that the untrusted data is unable to cause context switching from its residing HTML element context. The appropriate APIs for the remaining Rule#2–Rule#5 are "encodeForHTMLAttribute()", "encodeForJavaScript()", "encodeForCSS()", and "encodeForURL()" respectively.

3. Proposed approach

The proposed approach consists of two major phases: XSS vulnerability detection and 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. Our approach strictly follows OWASP's XSS prevention rules [4] presented in Section 2.2. That is; it applies ESAPI's escaping mechanisms if and only if the case belongs to Rule#1–Rule#5. If the case belongs to Rule#0 (any other cases not belonging to Rule#1–Rule#5), the proposed algorithm provides two options to user in order to secure the concerned statement: (1) Lenient mode – it requests the user to input an appropriate sanitization method; (2) Strict mode – it unconditionally removes the untrusted data from the code location it is referenced. Therefore, our XSSV removal algorithm is sound and complete in terms of removing all the XSS vulnerabilities in the server programs. In the following, we provide the details of the approach.

3.1. XSS vulnerability detection

This phase is based on the taint-based analysis technique adapted from our previous work [10]. This section reviews the previous work and presents the method for extracting XSS vulnerability information from a server program.

The basic definitions of the control flow graph (CFG), such as control and data dependence, defined by Sinha et al. [11] shall be adopted in this paper. As defined in the previous work [10], in a CFG, a node x is **transitively data dependent** on a node y if there exists a sequence of nodes, $y_0 = y$, y_1, $y_2, \ldots, y_n = x$, in the control flow graph such that $n \geqslant 2$ and y_j is data dependent on y_{j-1} for all j, $1 \leqslant j \leqslant n$. 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.

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.

In this paper, 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, in our previous work [10], we have implemented the identification of pv-outs by tracking the flow of untrusted data between input nodes and HTML output nodes. 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.

As an illustration, in Fig. 2a, statement 1 and 2 define the variables `memID` and `pwd` respectively with the data from a direct input source—HTTP request GET. Statement 5 defines the variable `name` with the data from an indirect input source— Database. Statement 11 references the untrusted data through the variable `html`. Each node in the CFG shown in Fig. 2b represents each statement in Fig. 2a. Therefore, nodes 1, 2, and 5 in Fig. 2b are input nodes. Node 11 is a pv-out because it satisfies the third condition of the definition of pv-out.

Note that the term "potentially vulnerable" is used because the HTML output statement may not be actually vulnerable to XSS attacks if the untrusted data referenced is not controllable by external users. For example, in our approach, data read from indirect input sources is considered as untrusted although it may also be defined by the programmer or the database. However, there is no harm in securing the data that may be manipulated by an external user because our escaping approach does not affect the resulting HTML outputs or other program operations as long as the untrusted data is intended to be referenced as a data in an HTML document.

3.2. XSS vulnerability removal

This phase contains two major steps—HTML context discovery and secure source code replacement. The first step first identifies the statements at which the untrusted data referenced in an HTML output statement can be escaped without compromising intended HTML outputs and security aspects. Then it extracts the HTML document structure surrounding each untrusted data from the source code and identifies the HTML context using pattern matching. The required escaping mechanism for each untrusted data is then determined based on the context identified and the XSS prevention rules [4]. The second step generates secure code structures using ESAPI's escaping APIs as replacements for original code structures.

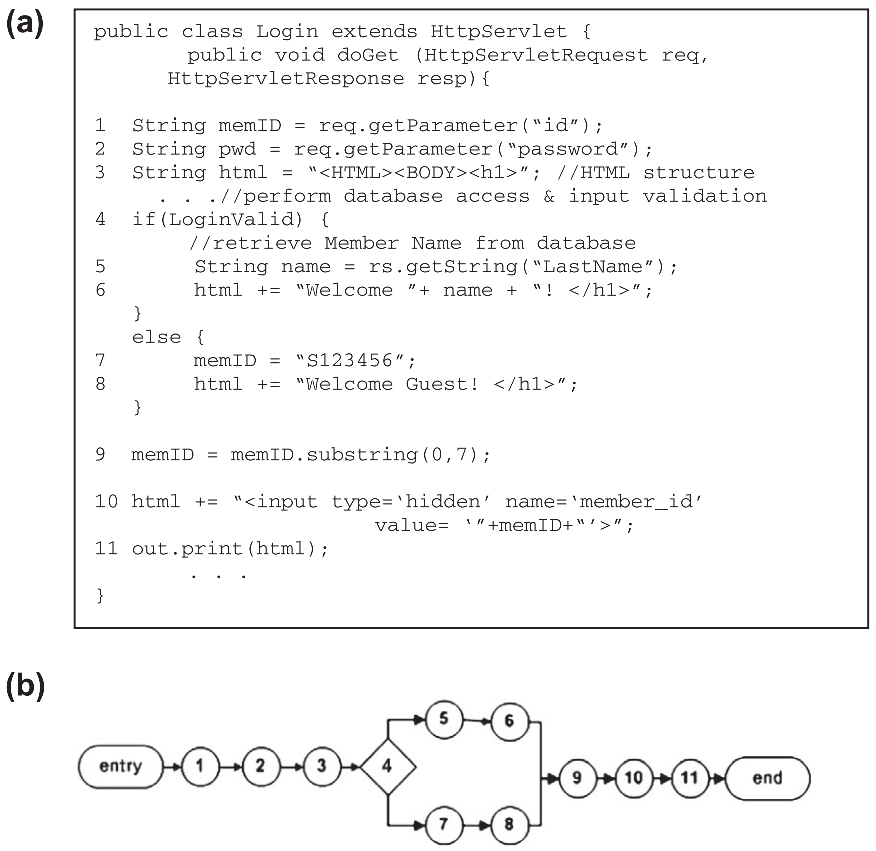
The algorithms are mainly based on data flow analysis and pattern matching. Therefore, static program analysis tools such as Soot [12] can be used to implement them. Next, we shall provide the detail of the two algorithms.

3.2.1. 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. However, in some scenarios, escaping should not be done in the vulnerable statement itself because the variable containing the untrusted data may also contain programmer-defined HTML document structures. Escaping may not be done in the input statements as well because the variables defined in the input statements may involve in more than one HTML context depending on different program paths. Therefore, for each pv-out, the algorithm first finds the statements at which the untrusted data can be properly escaped (we shall address such statements as **escaping statements** and the nodes representing them in a CFG as **escaping nodes**). Then, the algorithm identifies the HTML context by analyzing the HTML document structure surrounding the escaping statements. This technique is further explained in the following:

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.

- If o satisfies the first condition of the definition of pv-out, the algorithm marks o as *escape_stmt* and also marks the method that retrieves the untrusted data (e.g., `req.getParameter(‘‘input’’)`) as *to_be_ escaped*.
- If o satisfies the second condition of the definition of pv-out, there is an input node i and a variable v defined in i and used in o. The algorithm marks o as *escape_stmt* and also marks v as *to_be_escaped*.
- If o satisfies the third condition of the definition of pv-out, there is at least one sequence of nodes, $\{i = x_0, x_1, \ldots, x_n = o\}$, such that o is transitively data dependent on an input node i (note: there may be more than one sequence



(a)

```
public class Login extends HttpServlet {
       public void doGet (HttpServletRequest req,
      HttpServletResponse resp){

1  String memID = req.getParameter("id");
2  String pwd = req.getParameter("password");
3  String html = "<HTML><BODY><h1>"; //HTML structure
     . . .//perform database access & input validation
4  if(LoginValid) {
        //retrieve Member Name from database
5        String name = rs.getString("LastName");
6        html += "Welcome "+ name + "! </h1>";
   }
   else {
7        memID = "S123456";
8        html += "Welcome Guest! </h1>";
   }

9  memID = memID.substring(0,7);

10 html += "<input type='hidden' name='member_id'
                      value= '"+memID+"'>";
11 out.print(html);
        . . .
}
```

(b)

Fig. 2. (a) Code snippet of a sample Login Servlet program. (b) CFG of the program.

of nodes because untrusted data may flow from input node(s) to pv-out through different program paths). For each sequence of nodes, the algorithm tracks the flow of untrusted data from the input node i to the node x_j, $1 \leqslant j \leqslant n$; while tracking, if a node x_j performs a string operation which concatenates that untrusted data with another variable or a raw string data, the algorithm (1) computes the nodes on which x_j is (transitively) data dependent and extracts any raw string data found in those nodes; and (2) marks the node x_j as *escape_stmt* and the untrusted data (i.e., the variable containing the untrusted data) referenced in x_j as *to_be_escaped*, if the extracted data contains any HTML special character such as "<". If there is no such x_j exists, the algorithm simply marks o as *escape_stmt* and also marks the untrusted data referenced in o as *to_be_escaped*. This is because there is no node from i to x_n that could integrate any legal or programmer-defined HTML document structure into the untrusted data.

After an escaping statement is identified for a pv-out, the algorithm examines any raw string data used in the escaping node and applies pattern matching to extract any identifiers found in those data that can be identified as HTML document structure. For pattern matching, the algorithm uses an HTML pattern library which stores the patterns of HTML document structures. The document patterns are defined according to HTML 4.01 specification from W3C recommendation [13] (note: as XHTML 1.0 and HTML 4.01 are basically the same language in terms of the definitions of elements and attributes addressed in Rule#1–Rule#5 [14], our defined patterns could also be used for matching scripts written in XHTML). Any document structure extracted is matched against those patterns from the library and identified with an HTML context. If no identifier is found or the context is not recognized, the algorithm continues explore (1) the nodes on which the escaping node is (transitively) data dependent and (2) the HTML output 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 discussed in Section 2.2. 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 described in Section 2.2.

Note that there may be more than one escaping statement for a pv-out because (1) it may satisfy more than one condition of the definition of pv-out; (2) there may be more than one input node influencing the pv-out. The above pattern matching procedure is to be performed for each escaping statement.

As an illustration, Fig. 3 shows the pattern matching analysis performed for the pv-out in the Login Servlet program from Fig. 2. The sequences of nodes {1, 9, 10, 11} and {5, 6, 10, 11} shown in shaded color in Fig. 3 are the two sequences of nodes on which the pv-out, node 11, is transitively data dependent, and nodes 1 and 5 are the input nodes. According to the above HTML context discovery algorithm, for each sequence of nodes, the algorithm traverses the nodes starting from the input node and extracts any identifiers found. As shown in Fig. 3a, for the nodes {1, 9, 10, 11}, the HTML doc-

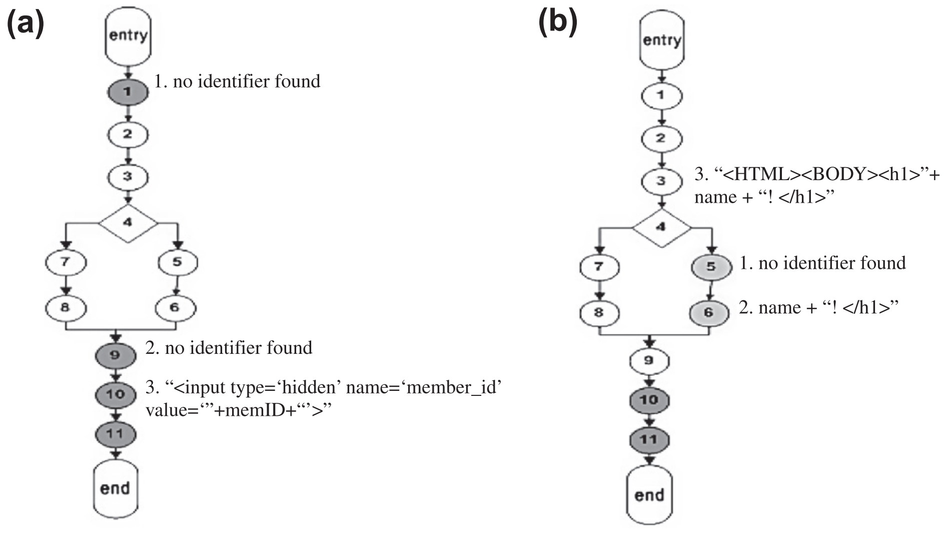


Fig. 3. Sequences of Nodes that are analyzed and HTML document structures collected from string objects while identifying the HTML contexts of untrusted data referenced in a vulnerable node. Both CFGs represent the Login Servlet program from Fig. 2a. (a) Nodes analyzed while tracking the flow of untrusted data from the input node 1 to the HTML output node 11. (b) Nodes analyzed while tracking the flow of untrusted data from the input node 5 to the HTML output node 11.

ument structure after analyzing node 10 is `''<input type='hidden' name='member_id' value=''' +memID+ ''' >''`. The algorithm marks node 10 as *escape_stmt* and the variable `memID` as *to_be_escaped* as the algorithm tracks that the untrusted data accessed in node 1 has been assigned to `memID`. The document pattern extracted at node 10 is then matched against the patterns from the library and it is identified as *HTML attribute context* which corresponds to Rule#2. Similarly, as shown in Fig. 3b, for the nodes {5, 6, 10, 11}, when the algorithm traverses node 6, it identifies some HTML special characters; thus it marks node 6 as *escape_stmt* and the variable `name` as *to_be_escaped*. During pattern matching step, after analyzing the escaping node, node 6, the algorithm continues to analyze the nodes on which the escaping node is (transitively) data dependent. Therefore, node 3 is analyzed next. From node 6 and 3, the document structure extracted is `''<HTML><BODY><h1>'' + name + ''! </h1>''`. It is then identified as *HTML element context* which corresponds to Rule#1.

3.2.2. Secure source code replacement

As an input, from the previous step, this algorithm receives the information of *escape_stmts*, *to_be_escapeds*, and *to_be_escapeds*' escaping rules and corresponding escaping methods. Firstly, 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 algorithm wraps the untrusted data referenced in *escape_stmt* with appropriate escaping APIs using the following three steps:

(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()`. The corresponding APIs for other escaping mechanisms are discussed in Section 2.3.
(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 algorithm provides two options to user. If the lenient option is chosen, the algorithm 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, the algorithm sets a default sanitization method which returns an empty string as *escape_api*. Secure source code replacement is then performed in the same procedure as the above steps 2 and 3 using this escaping API *escape_api*. For example, as discussed in Section 2.1, in Fig. 1, the untrusted data at line 4 is referenced in a complex HTML context which does not belong to any of the contexts stated in Rule#1–Rule#5. For such case, this algorithm in strict mode performs the following:

`<img src= '<%=saferXSS.emptyStrAPI(`**`untrusted_variable`**`)`**`%>`**`'/>`.

The resulting HTML output is:

`<img src= ''/>`.

Therefore, our algorithm in strict mode shall produce unintended HTML outputs for the cases belonging to Rule#0. However, as we discussed, 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 algorithms. And code modification required is very minimal because only objects containing untrusted data are wrapped with escaping API calls. To facilitate software maintenance and further security auditing purposes, the algorithm 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. 2, the algorithm will produce the output as shown in Fig. 4 (for brevity, only the necessary statements are shown). The modified statements are shown in bold.

4. Evaluation

We developed a prototype tool called saferXSS to implement the proposed approach. Using the tool, we evaluated the proposed approach on five open-source web applications. In the evaluation, we sought to answer the following questions: Does the approach compromise the required HTML outputs of the programs? Is the approach effective in removing the XSSVs of real-world

```
import org.owasp.esapi.ESAPI;
public class Login extends HttpServlet {
      . . .
   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.

applications? In the following, we discuss the implementation and evaluation of our proposed approach in detail.

4.1. Implementation

The prototype tool, saferXSS, was developed through the use of program analysis tool, Soot [12]. Fig. 5 shows the architecture of saferXSS and of the proposed approach. The tool consists of three modules; Program Analyzer, XSSV Detector, and XSSV Remover. It receives Java Servlet files as inputs. For each input program, program analyzer uses Soot's APIs to build the control flow graph (CFG) and stores the properties of each control flow node in global variables. XSSV Detector includes two major modules: data tracer and identifier. The two modules combine together to implement the XSSV detection phase discussed in Section 3.1. Data tracer traverses each node in the CFG and finds the HTML output nodes which reference untrusted data. Then, for each node found, identifier performs data dependency analysis to determine the nodes where escaping is to be performed. XSSV Remover consists of two major modules: context finder and code wrapper. These two modules implement the two algorithms discussed in Section 3.2. Code wrapper provides a user interface for user to set lenient or strict mode. In lenient mode, whenever the code wrapper encounters a vulnerable statement belonging to Rule#0, the interface shows the vulnerability information to user and requests the name of preferred sanitization/escaping method from user. As outputs, the tool produces modified Java Servlet programs and vulnerability report containing information of the potential XSS vulnerabilities found in the program. As a final step, user has to install ESAPI into the application and re-compile the modified programs.

4.2. Test subject

For evaluation, we selected five Java-based open source applications, *Events*, *Classifieds*, *Roomba*, *PersonalBlog*, and *JGossip*, as test subjects. *Events* and *Classifieds* are downloaded from GotoCode [16]. *Roomba*, *PersonalBlog*, and *JGossip* are downloaded from Sourceforge [15]. *Events* (3817 lines of code or LOC) is an event management system where users may participate or organize events. *Classifieds* (5744 LOC) is an online classifieds system where users may advertise various items or visit the system for online shopping. *Roomba* (3438 LOC) is a room booking system for small to medium-sized hotels. *PersonalBlog* (17,149 LOC) is an online blogging system where various users can publish their contents online. *JGossip* (79,685 LOC) is a forum system for discussion of various topics among users and it is a large-scale application. *PersonalBlog* and *JGossip* have also been used as benchmark applications in a related work [29]. The LOC counts do not include library classes. All these applications differ in functionality and usage. Yet, contributions or inputs from anonymous users are common in these applications. Therefore, XSS vulnerability is a serious security concern since a malicious user may easily trick many innocent users through exploiting an XSS hole.

4.3. Experiment

We conducted experiments on each test subject by performing the following procedure. First, we specified the directory of the subject's source folder for saferXSS, set the tool in strict mode, and ran it. The tool analyzed all the Java files and modified the programs with potential XSSVs. It also produced the XSS vulnerability report which contains information about modified program state-

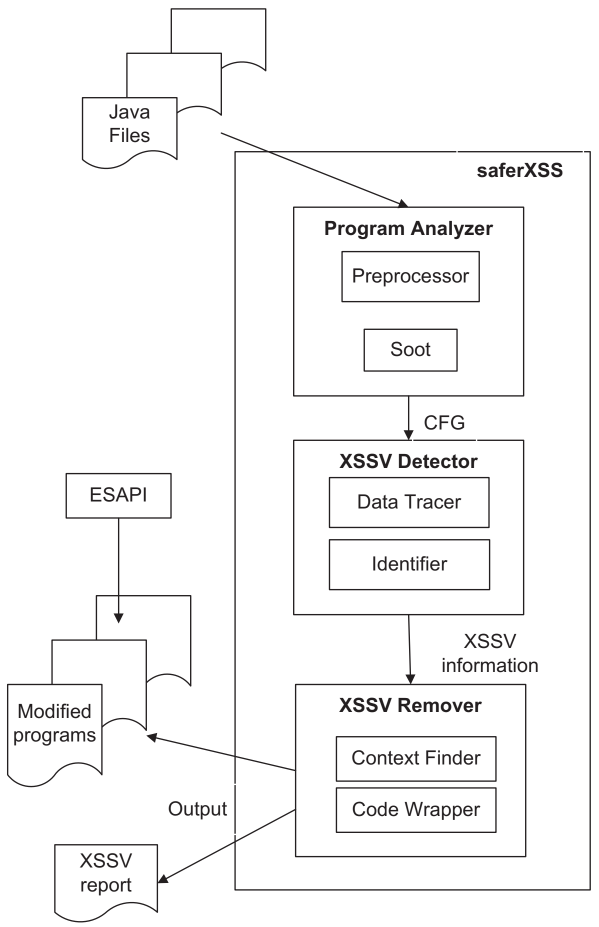


Fig. 5. Architecture of saferXSS.

Table 1
XSS vulnerabilities removed by saferXSS tool.

Test subject	#Potential XSSVs	#Modified statements	#Modified files
Events	81	81	11
Classifieds	123	123	10
Roomba	153	153	28
PersonalBlog	84	84	14
JGossip	312	312	40

Table 2
Real XSS vulnerabilities found in the test subjects and potential XSS vulnerabilities reported by saferXSS tool, which correspond to each HTML context type.

HTML context	#Real XSSVs/#Potential XSSVs				
	Events	Classifieds	Roomba	PersonalBlog	JGossip
HTML element	0/0	0/85	62/78	0/35	12/85
HTML attribute	9/28	9/17	42/45	0/12	12/37
JavaScript	0/0	0/0	2/2	0/0	0/3
CSS	0/0	0/0	0/0	0/0	0/83
URL	11/53	10/21	18/23	1/11	4/101
Rule#0	0/0	0/0	5/5	0/26	0/3
Total	20/81	19/123	129/153	1/84	28/312

ments (such as the one shown in Fig. 4). Independently, we also inspected the source code of each subject to confirm the potential XSSVs reported by the tool. Table 1 shows the results of the tasks performed by the tool. One potential XSSV corresponds to one program statement (i.e., a pv-out). Table 2 shows the statistics obtained from our code inspection and the tool. It shows the numbers of real XSSVs (inspected by us) and potential XSSVs (reported by the tool) corresponding to each type of HTML context where untrusted data is referenced. We observed that the real XSSVs present in the test subjects are mainly caused by improper input sanitization (*Classifieds*, *Events*), total absence of sanitization (*Roomba*), or failure to identify all the input sources (*PersonalBlog, JGossip*). Note that potential XSSVs include both real XSSVs and false positives.

Second, we installed ESAPI project into the modified subject, set up both the original and modified subjects on Eclipse,[1] and deployed them on Tomcat 6.0 server.[2] We also configured the database connections with MySQL 5.0[3] and ran the subject's SQL files on MySQL database to create the required tables.

Third, based on the XSS vulnerability report produced by the tool, we manually created a test suite, called functional test, in which a test case was formed for each modified program statement. This test suite was designed to test if the modified statements still produce the HTML outputs as intended. We then manually executed the test cases on both the original and modified subjects and compared the results (i.e., the resulting HTML documents). If the modified subject produces the same output as the original subject for a given test case, it is counted as *one test passed* (we did not consider the actual functional requirements of the subject). Otherwise, it is counted as *one test failed*.

Fourth, based on the real XSSV information inspected by us, we created another test suite, called XSS test, in which a set of test cases was formed for each real XSSV. Each test case set was constructed with a variety of XSS attack vectors designed to exploit a specific XSSV. The attack vectors were crafted based on the HTML context and escaping statement information reported by the tool. The attack patterns were also referenced from RSnake [17] and Kieżun et al.'s attack library [33]. They were crafted in such a way that a successful attack results in an alert message or a malformed web page. If no attack from a given set of test cases succeeds, it is counted as *one test passed*. If one of the attacks from the set succeeds, it is counted as *one test failed*. We again manually executed the test cases on both the original and modified subjects, and verified the results. As an illustration, some XSS attack patterns and the results of their executions on both the original and modified codes are shown in Table 3. The web pages shown in the first column are the places the attacks occurred. The attack vectors were injected into the input parameters shown in the third column. The injection may not take place in the same web page where the attack occurred depending on the stored and reflected XSS scenarios. For example, to expose the XSSV in the web page "bookings.step3" of *Roomba*, the attack vector "'; alert('XSSed');" was injected into the HTTP request GET parameter "firstname" in the web page "bookings.step1" where the data was then stored into the database.

Finally, we recorded the results of functional and XSS test cases executed on both original and modified test subjects. The results are listed in Table 4.

4.4. Result and discussion

4.4.1. Functional integrity

As shown in Tables 1 and 2, we observed false positives produced by the tool in all the test subjects (#Potential XSSVs – #Real XSSVs) because some of the statements are not exploitable by manipulating the values of user inputs. We found that some data retrieved from the database are actually defined by the program-

[1] http://www.eclipse.org
[2] http://www.tomcat.apache.org
[3] http://www.mysql.com

Table 3
Example of XSS attacks performed on original and modified test subjects.

Web page	HTML context	Attack vector	Original Code	Result	Modified Code	Result
Roomba/ viewTable	Rule#0 (URL inside JavaScript>	`chosenTable = '';!−''<XSS>=`	`<script>var newUrl =''viewTable.jsp?tableId='' +<%=chosenTable%></script>`	Mal-formed web page	`<script> var newUrl = ''viewTable.jsp?tableId='' +<%=saferXSS.emptyStrAPI (chosenTable)%> </script>`	No mal-formed web page
JGossip/jspf. messageForm	HTML element	`name=<SCRIPT>alert ('XSSed')</SCRIPT>`	`<td><%= request.getParameter (''name'')%></td>`	Alert message "XSSed"	`<td><%=ESAPI.encoder().encodeForHTML (request.getParameter(''name''))%></td>`	No alert message
Events/Login	HTML attribute	`querystring='';!−''<XSS>=`	`<inputtype='hidden' name ='querystring' value=<% =getParam(request, ''querystring'')%>>`	Mal-formed web page	`<inputtype='hidden' name='querystring' value =''+ESAPI.encoder().encodeForHTMLAttribute (getParam(request,''querystring'')%>>`	No mal-formed web page
Roomba/ bookings.step3	Java-Script	`firstname=';alert('XSSed');`	`<script> var firstname =<%=rs.getString (''firstname'')%>;</script>`	Alert message "XSSed"	`<script> varfirstname=<%=ESAPI.encoder(). encodeForJavaScript(rs.getString (''firstname''))%>;</script>`	No alert message
PersonalBlog/ deletepost	URL	`id='';<SCRIPT>alert ('XSSed')</SCRIPT>`	`<a href=''deletePost.do?method =executeFinish&postId=<%=id%>''>`	Alert message "XSSed"	`<ahref=''deletePost.do?method=execute Finish&postId=<%=ESAPI.encoder(). encodeForURL(id)%>''>`	No alert message

mer or the database (e.g., auto-increment primary key). Those data are not exploitable. Such false positives are expected because the XSSV detection phase of our proposed approach tends to be conservative (discussed in Section 3.1). As shown in Table 2, 5 cases from *Roomba*, 26 cases from *PersonalBlog*, and 3 cases from *JGossip* correspond to Rule#0. From our code inspection, we observed that the five cases from *Roomba* are real XSSVs whereas all the cases from *PersonalBlog* and *JGossip* are false positives. Regardless, all those 34 cases were secured with the tool's default sanitization method which returns an empty string. Therefore, as shown in the functional test results in Table 4, the tests corresponding to those cases failed for the modified test subjects. In a real scenario, a user may also choose the lenient mode; and based on the vulnerability information provided by the tool, he may decide whether to sanitize the untrusted data or to ignore the risk. For all other test cases, we confirmed that the resulting HTML documents of both the original and modified subjects were the same. Therefore, the results show that the proposed escaping procedure performed on the cases belonging to Rule#1–Rule#5 does not affect the intended HTML outputs of the programs.

4.4.2. Effectiveness

As the numbers of modified statements in Table 1 show, the saferXSS tool secured all the potential XSSVs identified by the tool. From our manual inspection, the potential XSSVs reported by the tool include all the real XSSVs found in the test subjects. As the XSS test results in Table 4 show, the attacks from XSS test suite successfully exploited all the real XSSVs in all the original subjects. None of the attacks was successful against the modified subjects. These results confirm that the real XSSVs inspected by us are actually exploitable. Quantitatively, the saferXSS tool removed 100% (197/197) of the real XSSVs in the test subjects with the precision of 26.2% (#real/#potential = 197/753). More importantly, this experiment demonstrates that the proposed approach is effective in completely removing all the real XSSVs.

4.5. Limitation

Among the three types of XSS attacks, our current approach does not prevent DOM-based XSS as it would require the analysis of client side scripts. Our current tool is only able to analyze server side scripts. If such analysis is possible in future work, the same XSSV removal approach proposed by us can be used to address DOM-based XSS. Also, our tool applies ESAPI's escaping APIs only when the HTML context corresponds to the contexts defined in Rule#1–Rule#5. Untrusted data referenced in any other contexts are unconditionally removed by the tool when set as strict mode.

Our approach does not track information flow across web pages. Thus the tool loses precision when an untrusted variable is first stored in a data store such as session objects and later referenced in an HTML output statement in a different web page. However, in order to prevent stored XSS, our proposed approach treats data accessed from such indirect input sources as untrusted and then indiscriminately applies escaping procedures to all untrusted data. As discussed in Section 3.1, this method causes no harm as long as the input is to be referenced as a data in an HTML document.

Our current approach is only targeted at Java-based web applications though its logic can be easily extended to other programming languages. Furthermore, our method modifies program source code instead of rewriting program bytecode. This is ineffective when the application source code is not available. However, our approach not only performs XSSV removal but also reports the statements modified and the statements that have security risks (in lenient mode) so that user can take further actions. As such, source code access is still required. It could though be future

Table 4
Results of functional and XSS test cases executed on original and modified test subjects.

Subject	Test suite	Original subject	Modified subject
Events	Functional	All passed	All passed
	XSS	20 tests failed	All passed
Classifieds	Functional	All passed	All passed
	XSS	19 tests failed	All passed
Roomba	Functional	All passed	5 tests failed
	XSS	129 tests failed	All passed
PersonalBlog	Functional	All passed	26 tests failed
	XSS	1 test failed	All passed
JGossip	Functional	All passed	3 tests failed
	XSS	28 tests failed	All passed

work to include this feature (bytecode rewriting) into our tool so that users could choose the preferred modification method.

5. Related work

Based on the way the XSS threat is mitigated, related approaches can be classified into three types— input validation and XSS testing, vulnerability detection, and attack prevention. These existing approaches focus on finding vulnerabilities present in applications or preventing XSS attacks with runtime monitors. By contrast, our approach focuses on removing XSSVs by using escaping mechanisms that prevent special characters contained in user inputs from invoking client script interpreters. This method is similar to Thomas et al.'s method [18] which removes SQL injection vulnerabilities by using prepared statements. However, their approach is not designed to remove XSSVs. Unlike securing a SQL statement with a prepared statement, the technique of securing an HTML output statement with an escaping API is more complex as it has to take into consideration the HTML context and the appropriate API to use.

5.1. Input validation and XSS testing

In software testing, both specification-based [19] and code-based [20,21] input validation testing approaches have been proposed. As input validation is often used as a key to enforce security, test cases that test the adequacy of input validation schemes could expose some XSSVs. However, these approaches are not suitable for finding all the XSSVs in the programs. And it is hard to determine the adequacy of test suite for the coverage of the XSSVs. Shahriar and Zulkernine [22] proposed a fault-based XSS testing approach which creates mutants for potential XSSVs using 11 mutation operators. Only test cases which contain adequate XSS attack vectors could induce different program behaviors between original and mutated program statements. However, their method is not yet practical as it requires users to identify all potential XSSVs and generate mutants by hand.

5.2. Vulnerability detection

These approaches are mainly based on static analysis techniques. Static approaches could in principal prove the absence of vulnerabilities. However, as they tend to generate many false alarms, later approaches incorporate dynamic analysis techniques to improve the accuracy.

5.2.1. Static taint analysis

These approaches track the flow of user inputs and check if any of them reaches sensitive program points (e.g., HTML output statements) without being properly sanitized. For tracking information flow, flow-sensitive, interprocedural, and context-sensitive data flow analysis techniques are used [23–25]. Livshits and Lam's approach [23] is based on points-to analysis using binary decision diagrams. Users are required to specify vulnerability patterns in a programming query language. Xie and Aiken's approach [24] is based on symbolic execution. Pixy [25] enhances the precision of data flow tracking by alias analysis. However, these approaches lose precision in the presence of custom sanitization functions. Some approaches conservatively assume that all those functions return unsafe data [25] or some request users to explicitly state the correctness of those functions [23,24]. Hence, false positives are inherent in this type of approaches.

5.2.2. Static string analysis

These approaches use formal languages to conservatively characterize the set of possible values a string variable may contain at sensitive program points [26,27]. Minamide [26] models the effects of string operations using finite state automata and language transducers, and checks the presence of "<script>" tags in untrusted data. Wassermann and Su [27] adapted Minamide's work and enhanced the detection of JavaScript contents in untrusted data by checking the possible values of untrusted data against the policies which specify various ways of invoking the JavaScript interpreter. However, these approaches still have limitations in modeling complex string operations such as character manipulation. Thus, these approaches also adopt overly conservative assumptions which often result in false alarms.

5.2.3. Combined static and dynamic taint analysis

These approaches adopt dynamic analysis phase to reduce the false positive rate inherent in static analysis phase. In Huang et al. [28], users are required to specify pre-conditions of sensitive functions (i.e., functions which contain HTML output statements) and post-conditions of sanitization functions. During runtime, these conditions are checked for conformance before executing any sensitive function. Martin and Lam [29] and Lam et al. [30] combine static analysis and dynamic monitoring to perform optimized information flow analysis that finds user-specified vulnerability patterns and apply model checking for generating attack vectors that expose real vulnerabilities. However, the effectiveness of these approaches depends on the completeness of vulnerability specification provided by user. Saner [31] checks the adequacy of sanitization functions motivated by the fact that existing static analysis approaches do not identify the faulty sanitization functions. Wassermann et al.'s approach [32] models the semantics of string operations based on Minamide's work [26], and performs concolic execution on PHP programs to collect path constraints and generate concrete test inputs that expose vulnerabilities. Similarly, Kieżun et al. [33] performs concolic execution to capture program path constraints and uses a string constraint solver to generate test inputs that explore various program paths. At sensitive program points, Kieżun et al. exercise two sets of test inputs—

one set contains valid inputs and another set contains XSS attack vectors. Then it checks the difference between the resulting HTML outputs.

In general, dynamic analysis-based approaches provide more accuracy than static analysis-based approaches but come at the cost of potentially complex frameworks to enable dynamic execution. Furthermore, concolic methods are known to suffer from state space explosion problem and thus, these methods may not cover all the code space (e.g., Kieżun et al. [33] only achieved maximum 50% of the code in their experiments). This weakness might result in false negatives. By contrast, although our method identifies potential vulnerabilities using static analysis-based approach, it automatically secures all identified potential vulnerabilities (with very minimal user interaction required for lenient mode) by inserting escaping routines in appropriate code locations.

5.3. Attack prevention

These techniques use dynamic monitoring systems, which are deployed on either server-side or client-side, to prevent real time XSS attacks.

In server-side methods, XSSDS [34] and XSS-Guard [35] set up a proxy between client and server, and check whether the input parameters in an HTML request become part of the client side scripts in the HTML response page and whether those scripts are actually intended by the application. Robertson and Vigna [36] proposed a web application framework that enforces strong typing of HTML documents so that intended document structures and inputs referenced in the documents can be separated and violations of intended document structures can be checked at runtime. Resin framework [37] provides programmers with interfaces for generating code assertions that define security policies while writing application code; and the framework checks for conformance of the defined security policies during runtime.

Client-side methods are mainly intended for clients. Beep [38] enhances the client's browser with the capability to detect malicious scripts based on the security policy provided by the client and prevent any malicious scripts from being executed. Similarly, Noxes [39] detects potential XSS attacks using the filter rules inferred from the web-browsing actions of the client.

In general, these approaches are effective at preventing real time attacks as they could intercept actual runtime values and check against security policies. However, these approaches are runtime-based and some approaches require programmers to develop applications in accordance with the requirements of the proposed runtime monitoring frameworks. By contrast, we provide a static program analysis method that removes XSSVs from applications before deployment.

6. Conclusion

In this paper, we presented a two-phase approach for finding and removing potential XSSVs in server programs. The first phase adopts a taint-based analysis approach to track the flow of user inputs into HTML output statements and identify potentially vulnerable statements. The second phase uses pattern matching and data dependency analysis to identify the HTML contexts in which the user inputs are referenced and the required escaping mechanisms that prevent code injection. Then it performs source code generation and replacement to secure potentially vulnerable statements with proper escaping APIs. Some of the existing XSS mitigation techniques are effective at detecting XSSVs or preventing XSS attacks; however they do not remove XSSVs. As more and more sophisticated attack patterns are discovered, vulnerabilities if not removed could be exploited anytime. We presented that the proposed approach is fully focused on removing XSSVs with minimal user intervention. We also developed the saferXSS tool that automates the proposed approach. In our evaluation, the tool was successful in removing all the real XSSVs found in five test subjects.

In future work, we intend to enhance both the current approach and the tool: (1) add client side script analysis in our proposed approach to prevent DOM-based XSS; (2) develop an analysis technique that tracks the flow of user inputs stored into persistent data structures and across web pages to accurately detect the influence of user inputs in HTML output statements; (3) discover up-to-date and new escaping mechanisms that could secure the untrusted data referenced in other HTML contexts not specified in Rule#1-Rule#5. Furthermore, we would like to explore the applicability of proposed approach in other security issues by making full use of ESAPI's capabilities.

References

[1] OWASP, November 2009, OWASP Top Ten project 2010. <http://www.owasp.org> (accessed January 2010).
[2] CWE/SANS, 2010, Top 25 Most Dangerous Programming Errors. <http://www.applicure.com/blog/cwe-sans-top-25-dangerous-programming-errors> (accessed June 2010).
[3] CWE, June 2010, CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). <http://cwe.mitre.org/data/definitions/79.html> (accessed June 2010).
[4] OWASP, June 2010, XSS (Cross Site Scripting) Prevention Cheat Sheet. <http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet> (accessed January 2010).
[5] US-CERT, Microsoft .NET Framework Contains a Cross-Site Scripting Vulnerability, October 2006. <http://www.kb.cert.org/vuls/id/455604> (accessed January 2010).
[6] </xssed>, May 2010, Vodafone.com XSS helps you trace unregistered "Pay As You Go" subscribers. <http://www.xssed.com/newslist> (accessed June 2010).
[7] A. Mueller, Cross Site Scripting (XSS), May 2009. <http://elegantcode.com/2009/05/28/cross-site-scripting-xss/> (accessed January 2010).
[8] ESAPI, OWASP Enterprise Security API, 2009. <http://www.owasp.org/index.php/ESAPI#tab=Project_Details> (accessed February 2010).
[9] A. Klein, July 2005, DOM based Cross Site Scripting or XSS of the Third Kind. <http://www.webappsec.org/projects/articles/071105.shtml> (accessed April 2010).
[10] 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.
[11] S. Sinha, M.J. Harrold, G. Rothermel, Interprocedural control dependence, ACM Trans Softw Eng Methodol 10 (2) (2001) 209–254.
[12] Soot, June 2008. Soot: a Java Optimization Framework. <http://www.sable.mcgill.ca/soot/> (accessed February 2009).
[13] W3C, 1999, HTML 4.01 Specification. <http://www.w3.org/TR/html401/> (accessed April 2010).
[14] W3C, 2002, XHTML 1.0 Specification. <http://www.w3.org/TR/xhtml1/> (accessed August 2011).
[15] Sourceforge, Open source website. <http://www.sourceforge.net> (accessed February 2009).
[16] GotoCode, Open source website. <http://www.gotocode.com> (accessed September 2009).
[17] RSnake, XSS (Cross Site Scripting) Cheat Sheet. <http://ha.ckers.org/xss.html> (accessed March 2010).
[18] S. Thomas, L. Williams, T. Xie, On automated prepared statement generation to remove SQL injection vulnerabilities, Inform. Softw. Technol. 51 (3) (2009) 589–598.
[19] J.H. Hayes, A.J. Offutt, Input validation analysis and testing, Empirical Softw. Eng. 11 (4) (2006) 493–522.
[20] H. Liu, H.B.K. Tan, Testing input validation in web applications through automated model recovery, J. Syst. Softw. 81 (2) (2008) 222–233.
[21] H. Liu, H.B.K. Tan, Covering code behavior on input validation in functional testing, Inform. Softw. Technol. 51 (2) (2009) 546–553.
[22] H. Shahriar, M. Zulkernine, MUTEC: mutation-based testing of cross site scripting, in: Proceedings of the 5th International Workshop on Software Engineering for Secure Systems (SESS'09), 2009, pp. 47–53.
[23] V.B. Livshits, M.S. Lam, Finding security errors in Java programs with static analysis, in: Proceedings of the 14th Usenix Security Symposium (USENIX Security'05), 2005, pp. 271–286.
[24] Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: Proceedings of the 15th USENIX Security Symposium (USENIX Security'06), 2006, pp. 179–192.
[25] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: a static analysis tool for detecting web application vulnerabilities, in: Proceedings of the IEEE Symposium on Security and Privacy (S&P'06), 2006, pp. 258–263.

[26] Y. Minamide, Static approximation of dynamically generated web pages, in: Proceedings of the 14th International Conference on World Wide Web (WWW'05), 2005, pp. 432–441.
[27] G. Wassermann, Z. Su, Static detection of cross-site scripting vulnerabilities, in: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), 2008, pp. 171–180.
[28] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, Securing web application code by static analysis and runtime protection, in: Proceedings of the 13th International Conference on World Wide Web (WWW'04), 2004, pp. 40–52.
[29] M. Martin, M.S. Lam, Automatic generation of XSS and SQL injection attacks with goal-directed model checking, in: Proceedings of the 17th USENIX Security Symposium (USENIX Security'08), 2008, pp. 31–43.
[30] M.S. Lam, M. Martin, B. Livshits, J. Whaley, Securing web applications with static and dynamic information flow tracking, in: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, 2008, pp. 3–12.
[31] D. Balzarotti, et al., Saner: composing static and dynamic analysis to validate sanitization in web applications, in: Proceedings of the IEEE Symposium on Security and Privacy, 2008, pp. 387–401.
[32] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, Z. Su, Dynamic test input generation for web applications, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'10), 2008, 249–260.
[33] A. Kieżun, P.J. Guo, K. Jayaraman, M.D. Ernst, Automatic creation of SQL injection and cross-site scripting attacks, in: Proceedings of the 31st International Conference on Software Engineering (ICSE'09), 2009, pp. 199–209.
[34] M. Johns, B. Engelmann, J. Posegga, XSSDS: server-side detection of cross-site scripting attacks, in: Proceedings of the Annual Computer Security Applications Conference (ACSAC'08), 2008, pp. 335–344.
[35] P. Bisht, V.N. Venkatakrishnan, XSS-Guard: precise dynamic prevention of cross-site scripting attacks, in: Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'08), 2008, pp. 23–43.
[36] W. Robertson, G. Vigna, Static enforcement of web application integrity through strong typing, in: Proceedings of the 18th USENIX Security Symposium (USENIX Security'09), 2009, pp. 283–298.
[37] A. Yip, X. Wang, N. Zeldovich, M.F. Kaashoek, Improving application Security with Data Flow Assertions, in: Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09), 2009, pp. 291–304.
[38] 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.
[39] E. Kirda, C. Kruegel, G. Vigna, N. Jovanovic, Client-side cross-site scripting protection, Comput. Security 28 (2009) 592–604.