# Research on Vulnerability Detection for Software Based on Taint Analysis

Beihai Liang [1,2], Binbin Qu [1,2], Sheng Jiang[1,2],Chutian Ye[1,2]
[1]School. of Compute Science, Huazhong University of Science & Technology, Wuhan, 430074,China
[2] Shenzhen Key Laboratory of High Performance Data Mining, Shenzhen, 518055, China
{beihai, bbqu, jwt, yevswang}@hust.edu.cn

***Abstract*—At present, Cross Site Scripting (XSS) vulnerability exists in most web sites. The main reason is the lack of effective validation and filtering mechanisms for user input data from web request. This paper explores vulnerability detection method which based on taint dependence analysis and implements a prototype system for Java Web program. We treat all user input as tainted data, and track the flow of Web applications, then we judge whether it will trigger an attack or not. The taint dependent analysis algorithm mentioned in this paper is used to construct the taint dependency graph. Next the value representation method of the string tainted object based on finite state automata is discussed. Finally, we propose the vulnerability detection method for the program. The experiment result shows that the prototype system can detect reflection cross-site scripting vulnerability well in those programs which don't have effective treatment for the user input data.**

***Keywords-XSS vulnerability; taint dependency graph; web security***

## I. INTRODUCTION

With the development of computer and Internet technology, Web applications in various fields are becoming increasingly popular, diversified and sophisticated. At the same time, the complexity of the program brings out more security problems. According to the Open Web Application Security Project (OWASP) application security vulnerability report, in 2010 and 2011, the top two security vulnerabilities were Injection Flaws and Cross Site Scripting (XSS).

At present, due to lack of effective verification and filtering mechanism for the Web request which contains the user input data, most Web sites exist XSS loophole. The attacker can launch an attack by constructing special input data.

Therefore, using static string analysis to excavate the vulnerability become an active research area. The critical of static string analysis is to identify all the possible values of the string expression at specified point of the testing program.

So far, many researchers have done lots of work about cross-site scripting vulnerability detection.

Paros Proxy tool[1] uses web crawler technology through a proxy to scratch the webpage and analysis them. It can be used to detect SQL Injection,XSS,CRLF and other common Web vulnerabilities. Because its inner test data can not be changed and the number of test data packet is limited, its false positives rate is high.

XSS-Me tool [2]is designed for XSS vulnerability, however, it can be only used for form test and single page analysis but not for the safety test with parameters in the URL link.

Xie Long [3] proposed a static detection method based on control flow analysis and data flow analysis. He analyzed the formation process for XSS vulnerability and designed a set of judgment conditions for the existence of XSS vulnerability, and got the control flow and data flow information by doing static analysis for the JSP source code.

Gary Wassermann [4] proposed a static detection method which based on string expression for tainted data flow. This method uses a regular language to describe the string at a specified program point to track tainted data.

Fang Yu [5] proposed an authentication method for string manipulation based on finite state machine in PHP program. He used finite state automata to represent the value set of a string variable and used an automatic operation to represent each string manipulation function.

This paper presents a vulnerability detection technique based on taint dependence analysis for Java Web program. We treat all user input as tainted data, and track the flow of Web applications, then we judge whether it will trigger an attack or not. Firstly we do static analysis for the program source code to generate the data dependency graph. Secondly we do the taint dependency analysis on the basis of the data dependency graph. Finally, we calculate all the possible values of the taint and validate them with the attack mode to detect vulnerability.

The remainder of this paper is organized as follows:

Section 2 gives the related definitions; section 3 gives the taint dependency analysis algorithm and construct the taint dependency graph; section 4 discusses value representation method for string tainted object based on finite state automata and gives the program vulnerability detection process; section 5 describes the design of vulnerability detection system for JAVA program; section 6 verifies the effectiveness of the detection method through experiment ; the end is the conclusion and prospect.

## II. RELATE DEFINITIONS

### A. *Data Dependency Analysis*

**Def 2.1 Control Flow Graph (CFG).** CFG = (N, E, n0, q)is a quadruple, in which N is a set of nodes in control flow graph, the node information is a statement for method M in program P ; $E \subseteq N \times N$ refers to the set of edges and reflects the control flow relationship between two statements in the program; $n0 \in N$ is the beginning statement of method M, which is the entry node of the control flow graph; q∈ N is the end statement of method M, which is the exit node of the control flow graph.

**Def 2.2 Defined-Clear Path(DCP).**For the node sequence p: $n_i, n_{i+1}, n_{i+2}, ..., n_j$ in CFG, if there is an edge for any ordered adjacent nodes <$n_i$,$n_{i+1}$>, we call P is a reachable path from $n_i$ to $n_j$. If there is no statement redefinition in path P for variable in node $n_j$, i.e. $\forall n \in P, \mathrm{DEF(n)} \cap \mathrm{USE(n_j)} = \varphi$ .We call path p is a DCP path from $n_i$ to $n_j$ in CFG.

**Def 2.3 Data Dependent (DD).**For two statements in CFG $s_i$ and $s_j$ which represented by two nodes $n_i$ and $n_j$ in Program P, only if the following two conditions are met: 1) the existence of a Defined-Clear path from $n_i$ to $n_j$ in CFG ; 2) variable collection $\mathrm{V} = \mathrm{DEF(n_i)} \cap \mathrm{USE(n_j)}$ , $\mathrm{V} \neq \varphi$ , statement $s_j$ data depends on statement $s_i$ by set of variables V (denoted by $\mathrm{s_j} \xrightarrow{\mathrm{V}} \mathrm{s_i}$ ).

The data dependency graph (DDG) is the graphical concept representation of the program data dependencies.

## *B. Taint Dependency Analysis*

A finite statement sequence $\pi :< s_1, s_2, s_{3,} ..., s_n >$ which is obtained by a topological in the CFG is a possible execution sequence. Each statement in the execution sequence $\pi$ performs an action (such as: get user input, execute judgment, perform calculations, access data, and display the results). The cross-site scripting vulnerability usually happens in jumping from one page to another. In this paper we focus on the above critical operation statements' security. The statement executed to perform critical operations is defined as the fragile sensitive point, which needs for vulnerability verification.

**Def 2.4 Vulnerability Sensitive Point (VSP)** .If a special statement S in the execution sequence of the program P performs critical operations (for example: page jump, executes scripts, etc.), the statement S may be exploited by attackers, then we call the statement S a fragile sensitive point in program P.

The taint dependency is a special kind of data dependency which reflects the way tainted data passes from the input to the fragile sensitive points, i.e.… a reflection of how the tainted data flow from the input to the fragile sensitive points under the program control.

**Def 2.5 Taint**. The user input data X in program P flows from the input interface to fragile sensitive point in the execution sequence, the variable flowed by the data X in the data flow path is called as taint.

**Def 2.6 Taint Dependence Graph (TDG)**.TDG is a directed graph G = <N,E>. N is a finite set of nodes, E ⊆ N×N is a set of directed edges. For each edge $< n_i,\ n_j > \in E$ , the value of $n_i$ is dependent on the value of $n_j$.

## III. TAINT DEPENDENCY ANALYSIS

Taint dependency analysis analyzes the data dependencies among each statement in data dependency graph reversely from fragile sensitive point for the purpose of constructing a taint dependency graph. The main steps are as follows:

1) get the data dependency graph according to the fragile sensitive points;

2) set tainted objects in fragile sensitive point as the initial taint;

3) generate the corresponding node depending on the type of taint;

4) treat the current taint as the data-dependent edge information and get the pollution sources;

5) parse the tainted object in the pollution sources;

6) analyze each taint.

In summary, the taint dependency analysis algorithm is shown in Algorithm 1.

```
Algorithm 1 :taint dependency analysis algorithm
Input1: fragile sensitive point (VSP)
Input2:data dependency graph (DDG)for the fragile
        sensitive point
Output: taint dependency graph (TDG)

 1： initialize taint analysis queue called taintQueue, and
the pollution sources collection which called
taintSourceSet is empty
 2： taintQueue.add( tainted objects in VSP)
 3： while(taintQueue $\neq \phi$ )
 4：   taint = taintQueue.getnext()
 5：    switch(type of the tainted object)
 6：      case：constant
 7：         generate new ordinary nodes (constant)
 8：       case：references
 9：          generate new ordinary nodes
10：           if(parameter references)
11：              generate parameters reference node, add the
                   dependency information
12：             else
13：             stmts：= treat the taint as the edge
           information ,take dependent statement from
           DDG
14：                 taintSourceSet.addAll(each statement in the
                   expression in stmts)
15：             end if
16：         case：expression
17：             generate temporary ordinary node
18：             taintSourceSet.add(taint)
19：      end switch
20：      for each taintSource ∈ taintSourceSet
21：          if(taintSource is a method call expression)
22：              if(taintSource is a input interface method
                      calls)
23：                  Generate uninitialized node, and add the
                         dependency information
24：              else
25：                  generates an operation node and add the
                       dependency information
26：                  taints：=taint parsed from taintSource
27：                  taintQueue.addAll(taints)
28：              end if
29：        else
30：            taints：= taint parsed from taintSource
31：            taintQueue.addAll(taints)
32：        end if
33：      end for
34： end while
```

Algorithm 1 starts from the fragile sensitive point. It analyzes the taint spread source in data dependency graph reversely. A constant ordinary node which no longer belongs to any pollution source is generated at Line 7; parameter reference node is generated at Line 11, the node's data dependency relies on the input parameter values , and we don't analyze the pollution source any more in the algorithm ; The uninitialized nodes generated by the Line 23 due to the method call from the input interface denotes that the node data is a non-trusted user input, which can be any value and no longer depend on other pollution sources . Algorithm 1 contains two nested while and for loops and generates a node in every cycle at least. Each node corresponds to a data object or a method call. If the total number of data objects in the data dependency graph from fragile and sensitive point is n. The time complexity of the algorithm 1 is O (n).

## IV. TAINTED STRING EVALUATION AND VULNERABILITY DETECTION METHOD

In order to describe the value of taint at the fragile sensitive point, this section focuses on the representation of the tainted string value and vulnerability detection method.

### *A. Tainted String Value Representation*

Static analysis method can not determine the specific value during the runtime of the program. But in the context of the program control flow and data flow graph, the possible value of the string is a determined finite set. That means the possible value of the string is determined during the runtime. So we can calculate the string variable value set due to the context of the program in static analysis. The context mentioned in this paper means taint dependencies.

According to the formal language theory, a string value set can be abstracted into a finite automaton. A finite automaton represents a language which contains a set of strings. During static analysis process, the value set of the tainted string can be represented by a finite automaton..

In this paper we descript the taint value in the taint dependency graph by using the automata operation library developed by Anders Møller etc[6] .

### *B. Vulnerability Detection Method*

The essence of vulnerability detection is to verify whether the value set of tainted string at the fragile sensitive points can match some special attack modes. Assume that a fragile and sensitive point exists in the execution sequence $\pi :< s_1, s_2, s_{3,} \ldots, s_n >$ of program P and this point could match a attack mode, this reflects that the malicious user data in the execution sequence has not been effectively cleaned-up and the program can be attacked at this point. Therefore, the key of vulnerability detection is how to get string value of fragile sensitive point tainted object.

This paper presents a vulnerability detection method based on taint dependency analysis. We construct the taint dependency graph from the tainted string at the fragile sensitive point to the taint input source on the basis of static analysis . We calculate the value set of the root node (the taint at the fragile sensitive point) in the taint dependency graph by using the automata operation provided by the automata operation library mentioned above. Next we take the intersection of the value set of the root node and the attack mode. If the intersection is not empty, it shows that the fragile sensitive point could match the attack mode successfully, and then we say the vulnerability exists. The process is shown in Fig 1.

## V. PROTOTYPE SYSTEM

In this paper, we propose the prototype design of vulnerability detection system for Java (JVDS).Relying on the dependency analysis for program source code, we construct the taint dependency graph and represent the value set of the tainted string by using finite state automata, then match the automaton for string value with the automaton for attack mode to verify whether the program has the safe handling against user input data. The exact design of JVDS is shown in Fig2.

The JDVS mainly includes the following functional blocks:

1) Static analysis. First abstract syntax tree of the input Java source code file will be generated by the preprocessor, lexical and grammar parser. Then we traverse the abstract syntax tree to transform the code information into equivalent code intermediate representation (IR).

2) Dependency analysis. we traverse the IR for the control flow as well as data flow analysis to collect data dependency information and generate the data dependency graph for the program.

3)Taint analysis. We analyze the tainted data pollution source at the fragile sensitive point for each vulnerability sensitive point according to the data dependency graph. Generate the taint dependency graph from fragile sensitive point to pollution source.

4)Vulnerability detection engine. We collect fragile sensitive

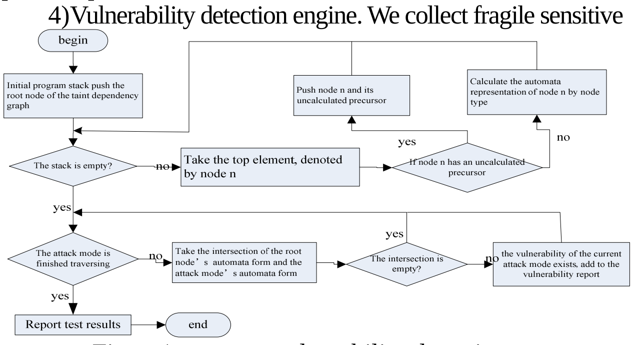

begin
Initial program stack push the root node of the taint dependency graph
Push node n and its uncalculated precursor
Calculate the automata representation of node n by node type
yes
no
The stack is empty?
no
Take the top element, denoted by node n
If node n has an uncalculated precursor
yes
yes
The attack mode is finished traversing
no
Take the intersection of the root node's automata form and the attack mode's automata form
The intersection is empty?
no
the vulnerability of the current attack mode exists, add to the vulnerability report
yes
Report test results
end

Figure1 program vulnerability detection process

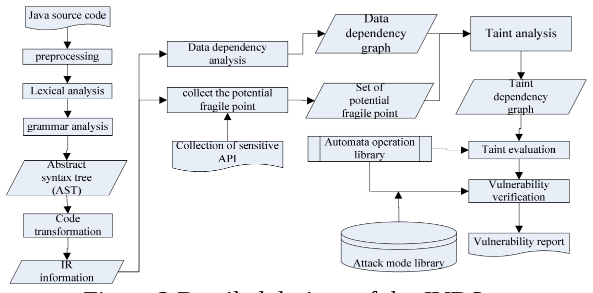


Figure 2 Detailed designs of the JVDS

points by traversing the IR and matching the sensitive API calls. Then we use automata operation library to evaluate the taint dependency graph at the fragile sensitive point. By matching with the attack mode, we implement the verification process for each fragile sensitive point's vulnerability and generate the vulnerability report.

5)Attack mode library, which is designed for the management and configuration of the attack mode.

## VI. EXPERIMENT ANALYSIS

The experiment is designed to analyze the correctness and validity of the prototype system.

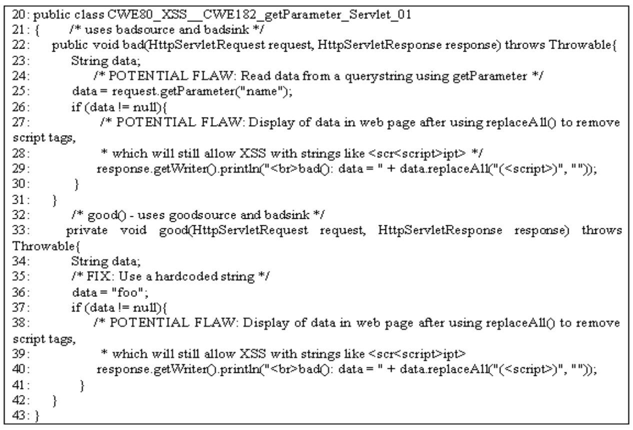
The main code of test case is shown in Fig3. The case code is excerpted from CWE182_getParameter_Servlet_01 in CWE80_XSS vulnerability public set of test case which belongs to CWE. Function bad() with loopholes and function good() with no loopholes. The functionality of the case code is getting the echo after processing the user's input data. Function bad() gets the user's input in line 25 and echoes back after calling the replaceAll function to filter the input data in line 29. The 29th line of the function bad() calls the function "java.io.PrintWriter.println" to response to the user's browser by the object for responding to user requests. The browser displays information to the user after parsing the data returned by the response object, if the response data returned embedded malicious script, the browser will execute the malicious script. So the function bad() may include a reflective cross-site scripting vulnerability, line 29 is a fragile sensitive point statement. The taint dependency graph is shown in Fig4.

The system found two XSS cross-site scripting vulnerability fragile sensitive points, one of which is vulnerable. The weak point is the statement in line 29 in the class named CWE80_XSS__CWE182_getParameter_Servlet_01 within the package called testcases. The matching attack mode is " .*\\<[Ss][Cc][Rr][Ii][Pp][Tt] .*\\>.* ", <script> is the tag of embedded script in HTML language. The function bad( ) does not have effective filtration treatment for the data ,so the malicious user can launch an attack by embedding a script which contains <script> tags. The experiment result shows that the system can detect reflective cross-site scripting vulnerability effectively in the program which doesn't have the effective treatment for the user input data.

## VII. CONCLUSIONS

Cross Site Scripting vulnerability exists in most Web sites. The main reason for its appearance is the lack of effective validation and filtering mechanism for user input data in Web request.

In this paper, we proposed the concept of taint dependency graph and the taint dependency analysis algorithm for Java program. Besides, we discussed the representation for the value set of the tainted string object based on the finite state automata and combined taint analysis and attack pattern matching rules to achieve the program vulnerability detection method. The experiment result shows that the prototype system can detect reflection



```
20: public class CWE80_XSS__CWE182_getParameter_Servlet_01
21: {       /* uses badsource and badsink */
22:     public void bad(HttpServletRequest request, HttpServletResponse response) throws Throwable{
23:         String data;
24:             /* POTENTIAL FLAW: Read data from a querystring using getParameter */
25:         data = request.getParameter("name");
26:         if (data != null){
27:              /* POTENTIAL FLAW: Display of data in web page after using replaceAll() to remove
script tags,
28:              * which will still allow XSS with strings like <scr<script>ipt> */
29:             response.getWriter().println("<br>bad(): data = " + data.replaceAll("(<script>)", ""));
30:         }
31:     }
32:         /* good() - uses goodsource and badsink */
33:       private void good(HttpServletRequest request, HttpServletResponse response) throws
Throwable{
34:         String data;
35:         /* FIX: Use a hardcoded string */
36:         data = "foo";
37:         if (data != null){
38:             /* POTENTIAL FLAW: Display of data in web page after using replaceAll() to remove
script tags,
39:              * which will still allow XSS with strings like <scr<script>ipt>
40:             response.getWriter().println("<br>bad(): data = " + data.replaceAll("(<script>)", ""));
41:           }
42:     }
43: }
```

Figure 3 Cross Site Scripting vulnerability case code

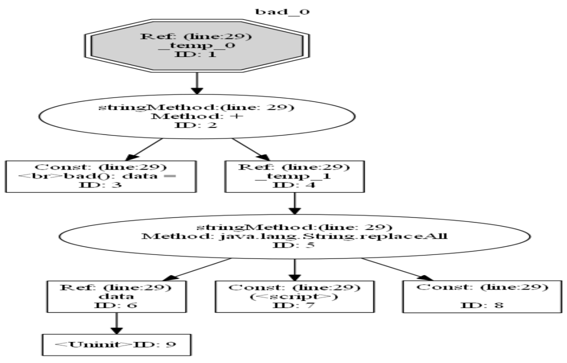

bad_0
Ref: (line:29) _temp_0 ID: 1
stringMethod:(line: 29) Method: + ID: 2
Const: (line:29) <br>bad(): data = ID: 3
Ref: (line:29) _temp_1 ID: 4
stringMethod:(line: 29) Method: java.lang.String.replaceAll ID: 5
Ref: (line:29) data ID: 6
Const: (line:29) (<script>) ID: 7
Const: (line:29) ID: 8
<Uninit>ID: 9

Figure 4 TDG of function bad()

cross-site scripting vulnerability well in those programs which don't have effective treatment for the user input data.

The following work is as follows: 1) Do the further study for reverse taint analysis techniques based on the method call dependence, improve taint dependency graph 2) Expand the prototype system which supports for the vulnerability detection for various programming languages .

## ACKNOWLEDGMENT

This research is supported by Shenzhen Key Laboratory for High Performance Data Mining with Shenzhen New Industry Development Fund under grant No.CXB201005250021A

## REFERENCES

[1] Paros, http://www.parosproxy.org/index.shtml. 2009

[2] XSS-Me, http://www.securitycompass.com/exploite.tml. 2009

[3] Xie Long. Research and Implementation of JSP cross-site scripting vulnerabilities static detection technology: [MS].Guangzhou: Zhongshan University Librarian, 2011

[4] G.Wassermann, Zhendong Su. Static detection of cross-site scripting vulnerabilities. In: Proc. 2008 ACM/IEEE 30th International Conference on Software Engineering. Leipzig , 2008: 171-180

[5] Fang Yu, T.Butan et al. Symbolic String Verification: An Automata-based Approach. In: Proc. of the 15th International SPIN Workshop on Model Checking of Software. Los Angeles, 2008: 306-324

[6] A.S.Christensen, A. Møller, M.Schwartzbach. Precise Analysis of String Expressi- ons. In: Proc.of 10th International Symposium, SAS San Diego.2003:1-18