Runtime Protecting System for Java Applications with Dynamic Data Flow Analyzing

Bo Feng, Miao Zhang, Guoai Xu, Xinxin Niu, Zhengming Hu
Information Security Center, State Key Laboratory of Networking and Switching Technology/ National Engineering Laboratory for Disaster Backup and Recovery
Beijing University of Posts and Telecommunications
Beijing, China

***Abstract* — With the rapidly development of the computer and internet technology, the security problem of software is more and more concerned. In this paper we present a runtime protecting system for java applications with dynamic data flow analyzing. We proposed the runtime software protecting algorithm and discussed the design and implementation of the runtime protecting system for java applications. At last, we compared runtime protecting system with static code analysis and analyze the performance of our implementation.**

Keywords- Runtime Protecting; Dynamic Data Flow; Security

I. INTRODUCTION

With the rapidly development of the computer and internet technology, the problem of software security is more and more concerned. At present, there are many researches about detecting and defending software vulnerability. Vulnerability is about a program's defects or errors, which may be used to damage the confidentiality, integrality, and availability of the program as well as the whole system the program stays in,[1] which is the chief reason why malicious code, Trojan and network invade can bring grate threat to computer system. Vulnerable software in your system may lead to system crash, data losing, and personal information leaking or even serious economic losses.

With the proposal of the concept of cloud computing, more and more services are provided in form of web applications which should be pay more attention to security. Cross site scripting (or XSS for short) is the most prevalent and pernicious web application security issue. It allows attackers to execute script in the victim's browser, which can hijack user sessions, deface web sites, insert hostile content, conduct phishing attacks, and take over the user's browser using scripting malware [2]. XSS vulnerabilities are commonly exploited in the form of worms on popular social or commercial websites, such as MySpace, Yahoo!, Orkut, Justin.tv, and Twitter. In 2007, the largest XSS worm, Samy, infected over 1 million MySpace user profiles in less than 20 hours [3].

Static code analysis is a solution to resolve this problem. Static code analysis examines code in the absence of input data without running the code and can detect potential security violations [4]. Static code analyzers are widely used in many software companies but they also have some limitations, which will be discussed later.

In this work we present a software runtime protecting system using dynamic data flow analyzing to resolve the security problem in the software development in the dynamic analysis perspective. The implementation for the java applications is discussed then.

II. DYNAMIC DATA FLOW ANALYZING

Static code analysis has some limitation in the vulnerability detection because the static analyzer does not know the execution state of the program and can not exactly determine the execution path of the code. Therefore, every static analyzer reports false positives. For example, some static analyzer may not exactly determine whether there is

```
public boolean login() {
  Connection conn = getConnection();
  String name = getName();
  String password = getPassword();
  String sql = "select * from users where name=?";
  if(!"guest".equals(name)) {
    sql += " and password='" + password + "'";
  }
  PreparedStatement ps =
conn.preparedStatement(sql);
  ps.setString(1, name);
  ResultSet rs = ps.executeQuery();
  return rs.next();
}
```

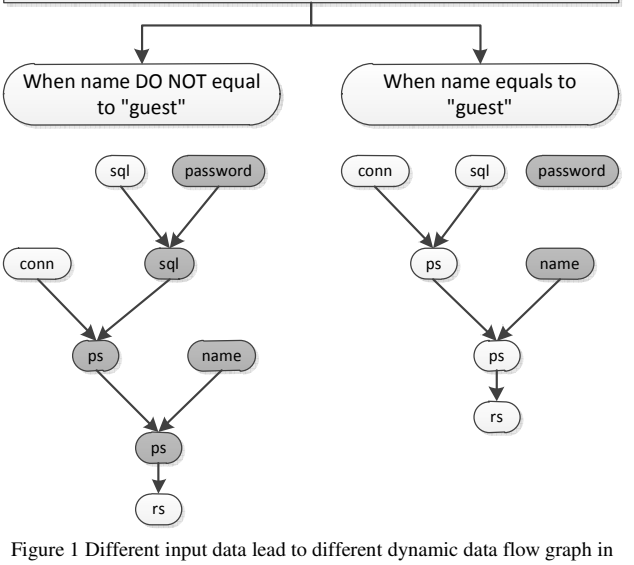


Figure 1 Different input data lead to different dynamic data flow graph in dynamic data flow analyzing

978-1-4244-5824-0/$26.00 ©2010 IEEE

SQL Injection vulnerability or not in the code given in the Figure 1, because the unsafe string concatenation is conditional executed determined by the value of the variable "name" which static analyzer do not know.

So, we try to resolve this problem by dynamic data flow analyzing. When software executed, it obtains data from user input, transmits and calculates them following some specific logic, then outputs the final result. In order to analyze the data flow during the software execution, we use Dynamic Data Flow Graph (DDFG) to describe the dynamic dependencies among variables at program runtime [5].

DDFG is the result of collecting and analyzing the variable state changes and dependencies among them at the runtime of the program. A vertex in the graph implies one of the states of the variables defined in the program. An edge from vertex v_i to vertex v_j implies the state v_j is directly depends on the state v_i. For example, the states of variables a, b and c are s_a, s_b and s_c initially; after the statement "c = a + b" executed, the value of the variable c is changed and it comes to a new state s_c' which depends on s_a and s_b. With dynamic data flow analyzing, we can analyze the program according to its execution state. Figure 1 shows the different DDFGs of the sample code snippet when variable "name" in different values.

Base on dynamic data flow analyzing, we can introduce runtime software protecting algorithm in the following section.

III. RUNTIME SOFTWARE PROTECTING ALGORITHM WITH DYNAMIC DATA FLOW ANALYZING

It is often said "Never trust user input!" According to the report of The Open Web Application Security Project at 2007 [2], the top three of the most critical web application security vulnerabilities are XSS, Injection Flaws and Malicious File Execution. All of these three are directly related to lack of user input validation, and the other vulnerabilities also depend on input of the program more or less.

The input data of the program can be a button click event on the web page, data from file or database, or anything else. Input makes a bridge for user and program but provide some chance for the attackers at the same time. To protect software at runtime, user input is the very key point.

There are four kinds of operations with input data we should focus on in a program:

- User input operations: The operations fetch data from user. For example, getting the command line parameters of the program, getting some information from the user submitted form, reading some data from a local file or from network.
- Input checking operations: The operations validate the user input data and remove the malicious content from it.
- Critical operations: These operations are sensitive to their arguments. If they use the unchecked user input data as the argument directly or indirectly, there may be some vulnerabilities in the software. The database query operation is a typical example.

In the runtime software protecting software algorithm, we tag the user input data as "unchecked". The tag can propagate when data dependency occurs during the execution of the program. The algorithm cab described as follows.

Let I, C and X are the predefined sets of user input operations, input checking operations and critical operations. In order to protect software at runtime, we need to intercept the invocation of the operations which are in I, C or X. Let o_c is the current executing operation of the program. For any operation o of the program, $R(o)$ is the result data of o and $P(o)$ is the set of parameters passed to o. Assume that we can use function *tag*() to tag the data during the program execution, *clearTag*() to clear the tag and *isTagged*() to check whether data has been tagged.

If $o_c \in I$, then *tag*($R(o_c)$).

Else if $o_c \in C$, then *clearTag*($R(o_c)$).

Else if $o_c \in X$, if there is any $p \in P(o_c)$ makes *isTagged*(p) is true then o_c is dangerous and should be blocked and logged to protect the software. Or else, o_c can be executed as normal.

Figure 2 illustrates the DDFGs during program execution and process of the runtime software protecting

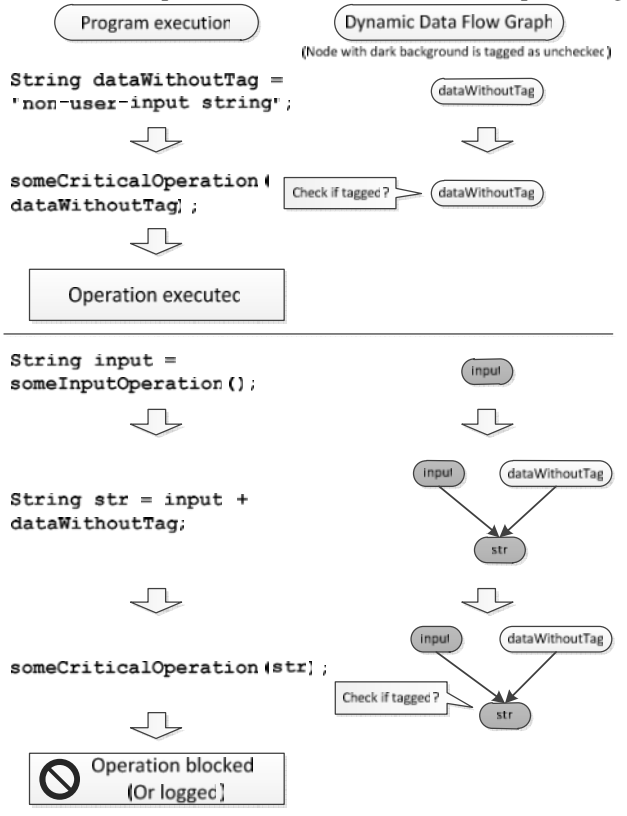


Figure 2 Process of the runtime software protecting algorithm and the DDFGs during processing

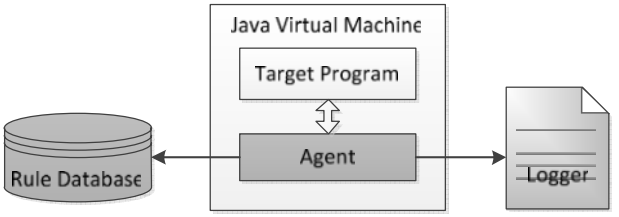


Figure 3 Structure of runtime protecting system for java applications

algorithm.

IV. IMPLEMENTATION FOR JAVA APPLICATIONS

To implement a runtime software protecting system for java platform applications, we need an agent which startup along with the target software we want to protect, tracing the execution of the software, tagging the unchecked data, blocking the malicious operations and record these attacks with a logger for later analysis. A rule database which defines many rules of data tagging and checking is also important for our implementation.

So, a runtime software protecting system for java platform applications may has structure which illustrated in the Figure 3. The gray parts are what we need to work on.

A. *Rule Database*

As we discussed in section 3, there are three kinds of operations we should focus on in a program. Therefore, we need to pre-define some rules about each existent attacking method in our rule database, which contains the three kinds of operations associated with the attack, for the agent to use in the dynamic analyzing. For example, in a Java web application, the following code exposure a SQL Injection flaw because of the concatenation of SQL string and unchecked data "name":

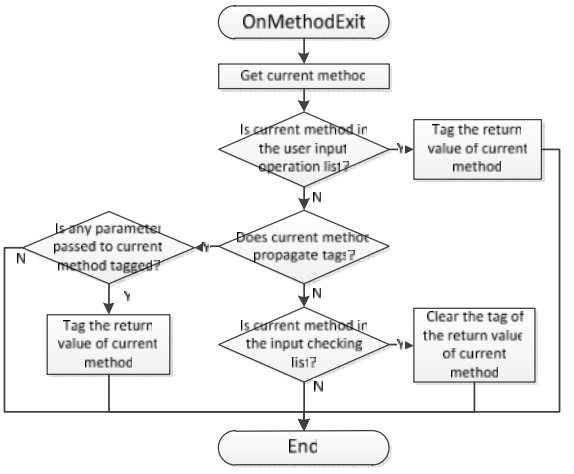
```
String name = request.getParameter("name");
String sql =
   "SELECT * FROM user WHERE userId=' + name + '";
PreparedStatement prepStmt =
   con.prepareStatement(sql);
```

Against SQL injection vulnerability, we can formulate a rule in accordance with Table 1.

In a Java web application, the getParameter() method of HttpServletRequest class is often used for retrieving the user input data from web form or URL parameters. The data fetched by this API may have potential dangerous and

Table 1 Rules for SQL injection vulnerability

Category	Java APIs
User input operations	HttpServletRequest.getParameter(String name)
Input checking operations	PreparedStatement.setInt(int index, int x) PreparedStatement.setString(int index, String x) And other similar methods in PreparedStatement class…
Critical operations	Statement.execute(String sql) Connection. prepareStatement(String sql)



should be tagged as unchecked.

Figure 4 The method entry event handler of the implementation

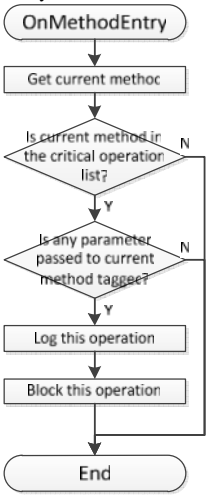


Figure 5 The method exit event handler of the implementation

Variables passed as arguments to PreparedStatement objects will automatically be escaped by the JDBC driver. So pass the unchecked data to setXXX() methods of PreparedStatement class is safe and the tag should stop its propagation.

However, if the unchecked data is passed to Statement.execute() or Connection.prepareStatement() directly or indirectly, there must be a SQL Injection flaw in this program and we should log it or even block this operation.

So far, we have specified a rule for the SQL injection attack. For other kinds of attacks, existing or appearing in the future, we need to analyze behavior of them, and define more rules to enrich the rule database.

B. *Agent*

Java programs are compiled into a form called Java bytecodes which can be executed by Java Virtual Machine (JVM). To the JVM, a stream of bytecodes is a sequence of instructions. Dynamic data dependencies of a Java program

is directly concerned with the JVM execution sequence of the bytecodes. However, if we trace the execution of the JVM instructions at the program runtime, the just-in-time compiler (JIT) of the JVM will unable to work effectively and the efficiency of the program execution will be greatly reduced.

Most of input-related vulnerabilities are due to lack of string object validation, and dependencies between objects often are caused by method invocation. Therefore, we may reduce the precision of the trace by focusing on the methods execution at runtime to implement an more effective agent basing on Java Virtual Machine Tool Interface (JVMTI). JVMTI is a native programming interface which provides both a way to inspect the state and to control the execution of applications running in the JVM. As a client of JVMTI, the agent can be notified when some specific events occurs, such as METHOD_ENTRY and METHOD_EXIT. And we can also use Object Tag API to tag the unchecked input data.

The implementation of the agent can be described as the Figure 4 and Figure 5. We register handlers to the method entry and exit events to trace the method invocation of the target program. After the user input operations executed, agent will tag theirs return value as "unchecked". The tag will be passed on along with the data dependency, but it will be removed after the invocation of the input checking operations. If the unchecked data is directly or indirectly used by the critical operations, then this invocation will be logged or even blocked by agent.

V. PERFORMANCE ANALYSIS

Comparing with static code analysis, the advantages of runtime protecting system can be expressed in the following aspects:

- Static analysis often suffers from lots of false positives which require the developer has more knowledge in security and more experience with the static analysis tool. Runtime protecting system traces the execution of the program, analyzes the program according to its execution state, logs the vulnerabilities and blocks the attacks against them only when they occur at runtime. So runtime protecting system does not have the problem of false positive.
- Static analysis tools need program source code to analyze with. But source code of a program is not always available in some situation. Our implementation of runtime protecting system for java applications can protect software without its source code, and it's useful for providing security protecting to the old-age applications which are difficult to maintain.

We test our implementation on a simple Java web application which contains SQL Injection and XSS vulnerabilities. Table 2 is the scan result of a web vulnerability scanner working on this application. We compare the vulnerability count of the target application before and after applying our implementation of runtime protecting system. After the application is protected, we manually perform attacks against these vulnerabilities, and all of attacks are blocked and logged exactly. Our implementation successfully protects the target application against attacks.

Table 2 Comparison of the vulnerability count before and after applying runtime protection

Vulnerability name	Before protection	After protection
SQL Injection	7	0
Cross site scripting(XSS)	8	0

VI. CONCLUSIONS

This paper presents the design and implementation of a runtime protecting system for java application. This system can increase the security of an existent java application at the cost of slightly execution efficiency reduce. In the development environment, runtime protecting is complementary to the static analyze, and can help developer finding and fixing potential security problems in program. And in the production environment, it can protect applications against attacks, and it is especially useful for the left from past and hard to maintain ones. To amend and improve the rule database and a better implementation of system is the aim of our future research.

ACKNOWLEDGMENT

This work is supported by National 863 (No. 2009AA01Z439), National Natural Science Foundation of China (No. U0835001), National S&T Major Program (2009ZX03004-003-03)

REFERENCES

[1] G. Zhao, H. Chen, "Data-flow based analysis of java bytecode vulnerability", in Proceedings of the 2008 the Ninth international Conference on Web-Age information Management (July 20 - 22, 2008).

[2] Open Web Application Security Project (OWASP), "The ten most critical web application security vulnerabilities", 2007, http://www.owasp.org/index.php/Top_10_2007

[3] XSS Worm, Wikipedia, http://en.wikipedia.org/wiki/XSS_Worm

[4] N. Ayewah, D. Hovemeyer, "using static analysis to find bugs," IEEE Software, pp. 22-29, September/October, 2008.

[5] B. Feng, M. Zhang, "Dynamic data flow graph-based software watermarking", in IC-NIDC 2009 Conference, 2009.

[6] D. Baca, K. Petersen, "Static code analysis to detect software security vulnerabilities - Does experience matter?," in 2009 International Conference on Availability, Reliability and Security, 2009

[7] B. Livshits, "Improving software security with precise static and runtime analysis", 2006.

[8] Java Virtual Machine Tool Interface Reference, http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html