A Practical J2EE Application Static Analysis Method Based Upon Taint Propagation

JianJun Hu[a], Qiaoyan Wen[b], DaiFei Guo[c]

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, 100876, China
Siemens Ltd., China

Abstract. Currently security audit/review for binaries is an upcoming method used to detect security vulnerabilities. In this paper we describe an efficient security audit method based on the java binaries. This method can The method in this invention can greatly reduce false positives and provides an efficient solution for automated secure auditing on binaries by only checking the exploitable security flows, especially for the scenarios which source codes are not available.

1 Introduction

Combined with java and web technology, J2EE Web applications are widely deployed in many large enterprises. These applications provide online support for enterprise business process, e-commerce activities, and information security is a crucial requirement.

According to OWASP[1] top ten and CVE[2] statistics, "Input Validation" is one of the most severe and pervasive security problems in web applications. Many famous security vulnerabilities, such as "SQL injection", "Cross-Site Scripting", "Path Traversal", "Command Injection", are caused by inefficient input validation.

For example, if programmer want to launch an operating system command, they should prepare the command string and call function exec() with the string as input. However, the command string may be controlled by the malicious user to run any OS commands, such as delete a file or open a port, etc. This kind of security vulnerability is named as "Command Injection".

There are two common approaches to address web application security issues, runtime scanning and static code review.

Runtime scanning based on web scanners [3] try to enumerate all the application data entrance point, feed them with multiple attack strings and analyze the response of the application for hints of security vulnerabilities [5]. It has several shortcomings:

-If the applications are complicated, it is hard to enumerate all the data entrance point, either by automatic crawling or manual inspection.

-Applications may have customized error message, or the inherent logic is not simply "fetch-and-show". Therefore, a lot of application vulnerabilities cannot be identified simply by request-and-response.

-If applications implement filtering mechanism that needs special crafted attack vectors to bypass, scanner cannot craft special attack vector and detect the possible vulnerabilities [5].

-The efficiency is low. Many commercial web application security scanners maintain a large database of attack vectors and try all these vectors one by one on all the data entrance points.

-The scanning may be dangerous, since scanner will test all the data entrance point, some of them may damage the application data.

Code review [6] has many advantages over the runtime scanning. All the data entrance point can be checked, the inherent application logic and filtering mechanism is recognized, and the checking process is absolutely safe [7]. However, there are several considerations about the code review:

-The efficiency: There are many kinds of security vulnerabilities and if the application is complicated, pure manual inspection is very inefficient.

-The intellectual property: The source code is intellectual property of software providers and customers always cannot get the source code to perform code review checks.

-Version problem, sometimes the source codes don't match the deployed program because of bad version management.

In this paper we propose a new approach to review based on the binaries. The review can be done with the help of automatic tools thus enhance the efficiency. Also the review is based upon the complied code, instead of source code, so there is no problem about the intellectual property.

2 Overview

The basic idea is very simple: we maintain a database of all the dangerous functions of java fundamental libraries, and

[a]hujianjun@bupt.edu.cn, [b]wqy@bupt.edu.cn, [c]Daifei.Guo@siemens.com

check whether the input of these functions can be tainted by user input.

2.1 Taint propagation introduction

Suppose we have a sample java servlet code as following:

```
private void list (HttpServletRequest request,
HttpServletResponse response)
{
  String user = request.getParameter("user");
  String password =
request.getParameter("password");
  try{
        String query = "SELECT Username, UserID,
Password
                                    FROM Users
WHERE
                  username ='" + user + "' AND
                  password ='" + password + "'";
        stmt.executeQuery(query);
  }
  …
}
```

An experienced programmer may find the SQL injection vulnerability in the above code. The taint propagation in the above code can be illustrated in Figure 2.1:

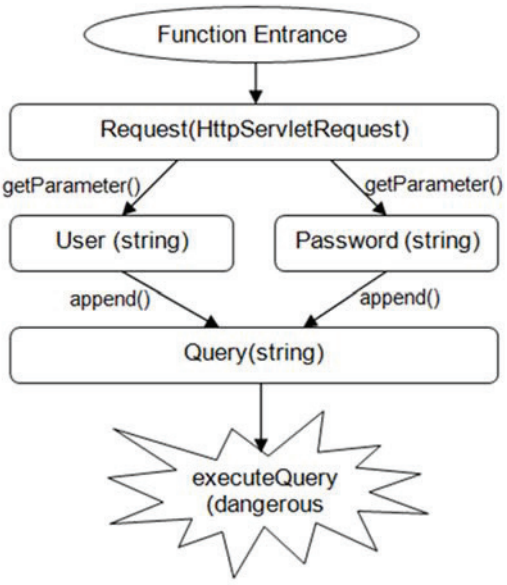

Figure 2.1 SQL injection control path

The "query" is a string variable. It is concatenated with two string variable, "user" and "password", which in turn comes from the result of calling sub-routine "request.getParameter()". Note that java language will compile string concatenation as append(). Byte code analysis will only see this function instead of concatenation.

Figure 2.2 illustrates the process of Taint Propagation.

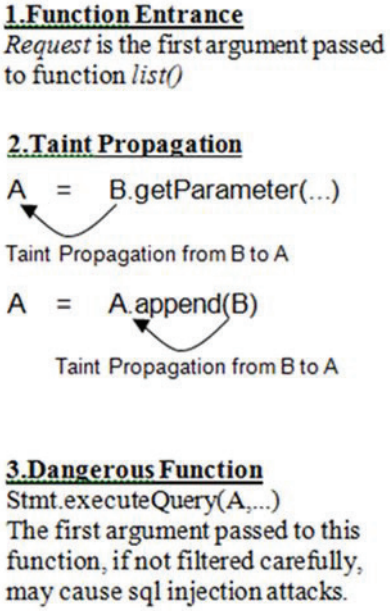

Figure 2.2 Taint Propagation process

3 Procedure

The taint propagation analysis in single function involves three key steps:

3.1 Function entrance

At the entrance of each function, we recode all the arguments with their order and type. For example, request is the first argument, of type "class HttpServletRequest".

Since java is strong typed language, if the argument type is long, int, double, etc, attacker can't inject any malicious content, so we record these arguments as UNTAINTABLE. We record other typed arguments as TAINTED.

For the entire local variable declared in the function body, we will record them as UNTAINTED. For the type of long, int, double, we will record them as UNTAINTABLE.

Now we have a table of all the variables to facilitate the following analysis, and we use a value to record the state of each variable.

Value	Meaning
-1	Untaintable
0	Untainted
1	Tainted by the first argument of the function
2	Tainted by the second argument of the function
4	Tainted by the third argument of the function
8	Tainted by the fourth argument of the function
…	…

3.2 Taint propagation

For each java byte code, we will inspect the effect of taint propagation. Since java byte code is formal language intended to be interpreted by JVM, it can easily been interpreted by our programs.

For example, declare a constant string will be compiled as LDC, when our program met with LDC, we will add a string variables tagged as "UNTAINTED".

For each sub-routine function, we will analyze arguments of the function. The taint will be propagated between the input arguments and the output argument. The detailed steps are as follows:

1) For all the input arguments,(it may include the instance class itself, because all the non-static function in java language belongs to class instance, this instance will be treated as the first argument of the function), analyze their type, if they are not int, long, double , etc, they will be tainted by all other TAINTED input arguments.

2) Analyze the function return type, if the type is taintable, it will be tainted by all the TAINTED input arguments.

Taint propagation is simulated by bitwise OR. For example: variable A's value is 4 and B's value is 10, after taint propagation, A and B all becomes 14, that means A and B are both tainted by the second, third and fourth argument in the function entrance.

Example 1: String A = A.Append(String B), This function will be compiled as String A = Append(String A, String B) .

(A, B) Before the call	(A, B) After the call
(1,0)	(1,1)
(0,0)	(0,0)
(4,1)	(5,5)

Example 2: long A = Func (String B, int C, class D)

(A, B, C, D) Before the call	(A,B,C,D) After the call
(-1,1,-1,0)	(-1,1,-1,1)
(-1,0,-1,4)	(-1,4,-1,4)
(-1,4,-1,2)	(-1,6,-1,6)

Since every function has specific inherent logic, the output may be tainted by only one of the input argument and often there is no propagation among different input arguments. We need to build the "diffusion mode" of each function to reduce the false positives and make the analysis more accurate. The "diffusion mode" can be built by firstly define all the "diffusion mode" of basic java functions, and then make diffusion analysis of the whole applications.

3.3 Dangerous Function

After the taint propagation, we will check all the sub-routine function, and check whether there is any dangerous function.

We will maintain a dangerous function list, for example:

```
<pattern>
  <className>java.sql.Statement</className>
  <methodName>executeUpdate</methodName>
  <signature>.</signature>
  <vulnParamIndex>2</vulnParamIndex>
  <category>SQL_INJECTION</category>
  <rootcause>.</rootcause>
</pattern>
<pattern>
  <className>java.sql.Statement</className>
  <methodName>executeQuery</methodName>

<signature>(Ljava/lang/String;)Ljava/sql/ResultSet;</
signature>
  <vulnParamIndex>2</vulnParamIndex>
  <category>SQL_INJECTION</category>
  <rootcause>.</rootcause>
</pattern>
```

"VulnParamIndex" stands for the position of argument, that if tainted by user input, will cause security problems. "className", "methodName" and "signature" are used to match the exact function. Many functions have multiple form of argument list declaration, so we simply use "." to match all of them.

In the example above, the executeQuery is a dangerous function, if the second argument is tainted, we will record this as a finding.

We can extend our signature database with other famous third party libraries. For example, for the famous ORM library hibernate2, we have the following signatures:

No	Dangerous Function	Argument Position
1	Net.sf.hibernate.Session.delete	1
2	Net.sf.hibernate.Session.iterate	1
3	Net.sf.hibernate.Session.find	1
4	Net.sf.hibernate.Session.createQuery	1
5	Net.sf.hibernate.Session.createSQLQuery	1

Also user can define their customized dangerous functions.

3.4 Taint propagation in multiple functions

Large application always has layered architecture, complex dataflow and well encapsulation. The main body of the application may only call the encapsulating function without calling dangerous function in our predefined database directly. So the analysis result of each function must be correlated together to construct the full attack path. For example:

```
Func1(String call1arg1, String call1arg2){
  …
  Func2(o, m, call1arg1 + call1arg2, n);
}
Func2(String call2arg1, int call2arg2, String call2arg3, int
call2arg4){
  …
  Func3(call2arg3);
}
Func3(String call3arg1){
  …
  Stmt.executeQuery(call3arg1);
}
```

The Func2 encapsulating Func3, which encapsulating dangerous function executeQuery(). If all the other code only call Func2 without direct calling stmt.executeQuery, such as Func1, and the argument call1arg1 can be tainted by user inputs, we may fail to recognize Func1 as a security violation.

The solution is simple: we will analyze the application recursively and populate our dynamic list of dangerous function.

For the example code before, the check process is as follows:

1) The first run, it finds Func3 with vulnerable argument 1, and adds Func3 to dangerous function list;

2) After the first run, the list is dirty and updated, so it needs to run again and recheck the application.

3) The second run, it finds Func2 with vulnerable argument 3, and adds Func2 to dangerous function list;

4) After the second run, the list is dirty and updated, so it needs to run again.

5) The third run, it finds Func1 with the vulnerable argument 1 and 2, and adds Func1 to dangerous function list;

6) …

7) The list is not dirty, the check is finished.

8) From the list, the full attack path is constructed, that is Func3 Func2 Func1.

Note that Func1, Func2, Func3 does not necessarily need to be in the same java class.

By above solution, we provide a practical and effective method to detect these possible vulnerabilities in J2EE web applications. It has been implemented as an independent application. It successfully finds all the vulnerabilities detected by manual inspection, and with great efficiency.

4 Conclusion

The method described in this paper can greatly reduce false positives and provides an efficient solution for automated secure auditing on binaries by only checking the exploitable security flows. And it greatly improves the efficiency of code review process.

Since it will simulate the data flow and function calling network, it is theoretically zero false negative. We have tested it on several realworld applications, and it successfully covers all the vulnerabilities found by other means. This solution also has the advantages such as no need for source codes, and it will construct the full attack path for the identified vulnerabilities.

Acknowledgements

We are grateful to a number of readers whose comments have substantially improved the paper, and the anonymous reviewers. Thanks especially to Beijing university of Posts and Telecommunications and Siemens Ltd., China.

References

1. OWASP: The Open Web Application Security Project, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
2. CVE: Common Vulnerabilities and Exposures, https://cve.mitre.org/.
3. Elizabeth Fong, Vadim Okun, “Web Application Scanners: Definitions and Functions”, hawaii international conference on system sciences, 2007
4. Marco Vieira, Nuno Antunes, Henrique Madeira, “Using web security scanners to detect vulnerabilities in web services”, dependable systems and networks, 2009
5. Nidal Khoury, Pavol Zavarsky, Dale Lindskog, Ron Ruhl, “An Analysis of Black-Box Web Application Security Scanners against Stored SQL Injection”, international conference on social computing, 2012
6. Tor Stalhane, Cat Kutay, Hiyam Alkilidar, Ross Jeffery, “Teaching the process of code review”, australian software engineering conference, 2004
7. Panagiotis Louridas, “Static code analysis”, IEEE Software, page 58-61, 2006