The Vulnerability Analysis of Java Bytecode Based on Points-to Dataflow

Tang Hong, Zhang Lufeng, Chen Hua, Zhang Jianbo

Abstract**—**Today many developers use the Java components collected from the Internet as external LIBs to design and develop their own software. However, some unknown security bugs may exist in these components, such as SQL injection bug may comes from the components which have no specific check for the input string by users. To check these bugs out is very difficult without source code. So a novel method to check the bugs in Java bytecode based on points-to dataflow analysis is in need, which is different to the common analysis techniques base on the vulnerability pattern check. It can be used as an assistant tool for security analysis of Java bytecode from unknown softwares which will be used as extern LIBs.

Keywords**—**Java bytecode, points-to dataflow, vulnerability analysis

I. Introduction

TODAy more and more Java applications have been used on the Internet. For example, many enterprise applications are developed under the J2EE framework[1]. The reason why people choose Java is not only for its cross platform ability, but also for its rich library support. The developers can use these library functions to design and develop their own products quickly and easily.

However, some unknown security bugs may exist when too many external components collected from the Internet have been used as LIBs in the software by the developers. For example, the SQL injection[2] bug may comes from the components which have no specific check for the input string by users. Some of these components have been provided with source code, but others have been only given in the form of bytecode. The users can take a security check on the source code to confirm the bugs while it is difficult to do so to avoid security risks without provided source code.

A novel method is introduced in this paper to check the bugs in the Java bytecode based on points-to dataflow analysis. This method has the following advantages:

- It does not require for source code disclosure.
- It can be used more freely for its independence of those specific predefined patterns.
- Its analysis process is global-wide, not just restricted to local fields or variables. So the analysis is more powerful.
- It describes the whole process that how the program exploits the bug with the points-to dataflow. Therefore, it is more helpful for the developer to secure the code from the essence of exploited vulnerability.

Hong Tang, Lufeng Zhang, Hua Chen, Jianbo Zhang are with the Beijing Institute of System Engineering, Beijing, 100101 9702# CHN (phone: +86-13521120770; e-mail: tanghone@ gmail.com)

II. Related Work

The common vulnerability analysis tools for Java language are static analysis tools for source code. Knizhnik has developed Jlint for bugs searching [3], which is focusing on inconsistencies and synchronization problems by analyzing data flow and building the lock graph. Jlint runs fast especially for large projects. Livshits has developed LAPSE (Lightweight Analysis for Program Security in Eclipse) [4]. LAPSE is focusing on the analysis of Java J2EE applications for common types of security vulnerabilities. PMD [5] can scan Java source code for empty try/catch/finally/switch statements and unused local variables, parameters and private methods as well as wasteful String/String Buffer usages. Furthermore PMD can be integrated with many IDEs such as JDeveloper, Eclipse, Jbuilder and etc.

The main technique used to check the bugs of Java bytecode is vulnerability pattern analysis. FindBugs [6] is one of such tools, which has been widely used. It has some functional patterns such as null pointer check, type check for abstract set, etc. The users can discover some bugs in the bytecodes to reduce the security risks with the help of FindBugs. However, it has two shortcomings: The first one is that the number of FindBugs' pattern is limited, which means that the users have no idea about the bugs that outside these patterns; The second one is that the patterns are mainly focusing on the local field of program, and they cannot handle the check on global scope.

III. The Essence of Exploited Vulnerability

Vulnerability is the bug or flaw of software system which can be exploited by attackers from outside. It depends on three basic conditions: the first is the channel to be used by outside attackers, which refers to the malicious input data; the second is the risky point which could lead to security breach of the system, which also refers to the execution of the malicious input data; the third is the connection between the malicious input data from outside attackers and the risky point, which is the data propagation path from outside to the operation point. Figure 1 points out the essence of exploited vulnerability.

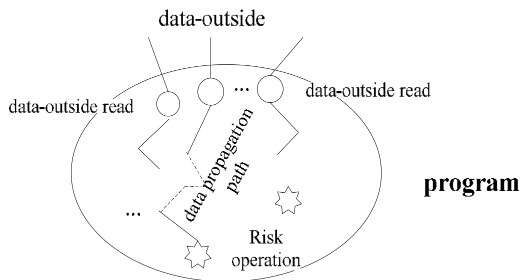

Fig 1. Essence of exploited vulnerability

Therefore, if the analyzers want to analyze the vulnerability of the program, they can start from the receiving data from outside, and meanwhile check the security related internal operation. If there are some close relations between the input data and the corresponding operation objects, it may indicate certain vulnerability of the program. However, it is difficult to confirm this kind of connections. Fig. 1 shows the data propagation path from the data-outside read point to the risk operation. It is the most difficult part in the analysis, which is how to trace the path inside the target program. In this paper the way is found by improving the common points-to dataflow analysis technique.

IV. Improving points-to dataflow analysis

Generally, the people can use points-to dataflow analysis to describe the connections between the variables in the program, for example, which variables will be affected by the same allocation instruction. Many researchers have done good work on points-to dataflow analysis[7-10]. Anderson has began points-to dataflow analysis in C language firstly [11]. Lhotak has developed SPARK[12] to analyze the points-to dataflow in Java language. The results of these analyses have been used to optimize the developing program. Some improvements have been made on the basis of SPARK to meet the requirement of our vulnerability analysis in this paper.

Our points-to dataflow analysis have been divided into two stages: pointer assignment graph (PAG) construction, and points-to set propagation. The following subsections describe the improved pointed-to dataflow analysis.

A. PAG construction

The pointer assignment graph consists of three types of nodes:

- Data-outside reading nodes represent sites reading the data from outside in the source program.
- Simple variable nodes represent local variables, method parameters and return values, and static fields.
- Field dereferences nodes represent field access expressions in the source program; each is parameterized by a variable node representing the variable being dereferenced by the field access.

The nodes in the pointer assignment graph are connected with four types of edges representing the pointer flow, corresponding to the four types of constraints imposed by the pointer-related instructions in the source program.

TABLE I
THE FOUR TYPES POINTERS ASSIGNMENT GRAPH EDGES

	Reading	Assignment	Field store	Field load
Inst	a : dst := read(data -outside)	dst := src	dst.f := src	dst := src.f
Edge	a → dst	src → dst	src → dst.f	src.f → dst
Rules	a → dst /(a □ pt(dst))	(src → dst) (a □ pt(src)) /(a □ pt(dst))	(src → dst.f) (a □ pt(src)) (b □ pt(dst)) /(a □ pt(b.f))	(src.f → dst) (a □ pt(src)) (b □ pt(a.f)) /(b □ pt(dst))

In this table, a and b denote data-outside reading nodes, src and dst denote variable nodes, and src.f and dst.f denote field dereference nodes. Depending on the parameters to the builder, the pointer assignment graph for the same source code can be very different, representing various levels of precision required by the points-to dataflow analysis.

B. Propagation of points-to set

After the pointer assignment graph has been built, the following step is to propagate the points-to sets along the edges according to the rules shown in Table 1. The following statements also show the rules for the propagation.

- Data-outside reading instruction means that if a points to dst, a belongs to the points-to set of dst. (rule 1)
- Assignment instruction means that if src points to dst and a belongs to the points-to set of src, then a belongs to the points-to set of dst. (rule 2)
- Field store instruction means that if src points to the field f of dst, and a belongs to the points-to set of src, and b belongs to the points-to set of dst, then a belongs to the points-to set of b.f. (rule 3)
- Field load instruction means that if the field f of src points to dst, and a belongs to the points-to set of src, and b belongs to the points-to set of a.f, then b belongs to the points-to set of dst. (rule 4)

According to these rules, we can construct the whole points-to flow of the program from the beginning to the end.

C. Results

With the result of the points-to analysis, the variable of the program has the points-to set whose elements are the instructions that read data from outside. So suppose there are two variables *v1* and *v2*, the points-to set of them are *pt(v1)* and *pt(v2)*. If $pt(v1) \cap pt(v2)$ is not empty, that means *v1* and *v2* all have the connections with some instructions reading data from outside. Let's move forward, suppose variable *vout* is the result of the reading instruction, and *vn* is the parameter of the risk method, then if $pt(vout) \cap pt(vn)$ is not empty, that means the data from outside has a propagation path to the parameter of the risk method, and the points-to flow is the a propagation path.

V. Analysis the vulnerability of Java bytecode

From previous discussion, it shows that users can analyze the vulnerability of the program by searching the instructions of input data from outside whose results are considered as *vout*,

and by searching the risk methods in the program whose parameters are considered as *vn*. With the help of points-to dataflow analysis, users can find out whether $pt(vout) \cap pt(vn)$ is empty or not. If it is not empty, it is concluded that there is a bug in the program which can be exploited by attackers, and some special protections have to be taken for the target bytecode in addition.

A. Analysis strategy

The following part defines the strategies of the vulnerability analysis, which includes the entrance point of input data from outside, the risk operation method, and the rules of points-to dataflow analysis. We use <*Ir, Pr, Tr*> to represent the strategies respectively.

Ir is the set of functions of Java language reading the input data from outside, which comes from APIs of JDK. In the future we can include some APIs from the other components. The below table specifies the set.

TABLE II
DATA-OUTSIDE READING FUNCTION

From user	args[], UI text input, console text input
From network	get url information
From environment	get environment variables□get system information
From file system	read file name□read the content of file

On some cases with low level security requirement, it is unnecessary to add all the items in the above table to the analysis strategy as Ir. You may just pick some functions such as “get url information” for you need to analyze the network part of the program.

Pr is the set of the points-to dataflow analysis rules, which mainly includes the context of the rules of Subsection 3.3. In the future we may add some other rules to meet the propagation.

Tr is the set of risk methods, which include local file operations, database operations, process operations, some risk native methods such as strcpy and etc. The table is shown below.

TABLE III
RISK OPERATION

Database operation	Database create, search, modify, delete, add and so on
File operation	File read, delete, create, copy, write
Risk native method	Risk system call and risk native method from DLL or lib
Network operation	Net connect, data send and receive

As *Ir*, for different levels of the security analysis, we can form different *Tr* by choosing different subset of table 3.

So the strategy depends on the requirements of the analysis. Based on the experience of analysts, different strategies can be adopted for different tasks in order to improve the speed and correctness.

B. Analysis model

Based on the definition of the analysis strategy<*Ir, Pr, Tr*>, the vulnerability analysis model <*Id, Pd, Td*> can be built. Id is the set of variables that denotes the results of security check on the input data from outside under *Ir* rule. Td is the set of variables that denotes the parameters of the risk methods. A set *PId* can be built by the way of points-to dataflow analysis whose element *pid* is the points-to set of the variable among the Id. A set *PTd* can also be built by the way of points-to analysis whose element *ptd* is the points-to set of the variable among the Td. When the element of *PId* crossing the element of *PTd*, its result can be collected to form a set *Pd*. If all the element of *Pd* is empty, that means the program is secure. The figure below shows the model.

.

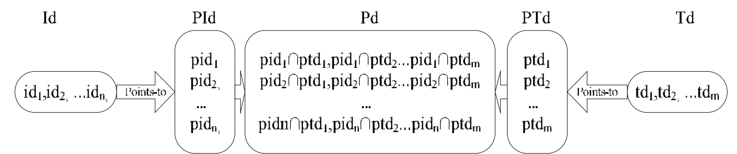

Fig. 2. The vulnerability analysis model base on points-to dataflow analysis

C. Analysis procedure

In order to analyze Java bytecode, the .class files need to be converted into the jimple files first. It can be easily done by open source tools such as Soot framework[12]. After that the code of the jimple files are scaned line by line to search the variables denoting the data-outside and the variables as the risk method parameters. Those are *Id* and *Td* described above. Then the points-to dataflow analysis of the target codes must be executed as described in Section 3. At last, the analysis model in Section 4.2 is used to get the risk data points-to set *Pd*, which shows the result of the vulnerability of the program. The analysis procedure is shown in the following figure.

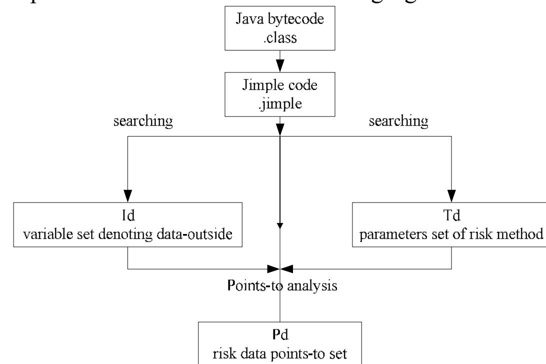

Fig 3. Aanalysis procedure

VI. TEST AND EVALUATION

A code segment as a simple example which is used to test the vulnerability analysis model presented is showed in Fig. 3. On the top part, a1-a2 are the main functions while on the below b1-b2 are the extern class A used in a1-a2. On the left a1-b1 are the source codes which describe the tasks of the program while on the right a2-b2 are the jimple format codes which are our target codes translated from the bytecode of the source codes on the left. Our analysis has been carried on the jimple codes.

Firstly, the program is scaned to search Id and Td. Line 1 is the point of reading data-outside and line 22 is the risk method

of calling native risk function. So the Id is {args} and the Td is {s3}.

Then the points-to dataflow analysis begins line by line as follows:

Pt(args)={1} (rule 1)
Pt(temp$0)={1} (rule 2)
Pt(temp$1)={}
Pt(s1)={1} (rule 2)
Pt(s2)={1} (rule 2)
Pt(temp$2)={}
Pt(a1)={}
Pt(this)={}
Pt(s)={1} (rule 1)
Pt(this.<main.A: java.lang.String> f)={1} (rule 3)
Pt(s3)={1} (rule 4)
Pd={Pt(args) ∩Pt(s3)}={{1}}.

The element of Pd is {1}, which is not empty. So there is vulnerability in the code, which can be exploited by attacker. If "`this.<main.A: java.lang.String f> = s` " is replaced with "`this.<main.A: java.lang.String f> = "hello"`" in the line 19. Some analysis tools still regard line 11 as a bug because of its calling of risky native method.

However, there is a different opinion. Here is the points-to dataflow analysis:

Pt(this.<main.A: java.lang.String> f)={}
Pt(s3)={} (rule 4)
Pd={Pt(args) ∩Pt(s3)}={{ }}.

That means there is no vulnerability of the program which shows the fact.

```
String s1 = args[0];
String s2 = s1;
A a1= new A();
a1.setF(s2);
String s3 = a1.f;
a1.printf(s3);
```

a1

```
...
1  args := @parameter0: java.lang.String[];
2  temp$0 = args;
3  temp$1 = 0;
4  s1 = temp$0[temp$1];
5  s2 = s1;
6  temp$2 = new main.A;
7  specialinvoke temp$2.<main.A: void <init>()>();
8  a1 = temp$2;
9  virtualinvoke
       a1.<main.A: void setF(java.lang.String)>(s2);
10 s3 = a1.<main.A: java.lang.String f>;
11 virtualinvoke
       a1.<main.A: void printf(java.lang.String)>(s3);
```

a2

```
public class A {
  public String f;
  static {
    SystemloadLibrary("printf.dll");
 }
   public void setF(String s){
   f = s;
 }
 public native void printf(String s);
}
```

b1

```
12 public class main.A extends java.lang.Object{
13  public java.lang.String f;
                          ...
14  public void setF(java.lang.String){
15   main.A this;
16   java.lang.String s;
17   this := @this: main.A;
18   s := @parameter0: java.lang.String;
19   this.<main.A: java.lang.String f> = s;
20   return;
21  }
22 public native void printf(java.lang.String);
                          ...
   }
```

b2

Fig 4. Example codes for check tests

There is a benchmark tool nist from SAMATE (Software Assurance Metrics And Tool Evaluation) [14], which is used to test the tools that collect the common security faults in Java program. This analysis model is able to find out all security faults in it.

VII. CONCLUSIONS

In this paper a novel method has been described to analyze vulnerability in Java bytecode. It is different to the common analysis techniques based on the vulnerability pattern check. The analysis model can be used as an assistant tool for security analysis of the bytecode for unknown software, especially before the unknown component can be used as extern LIBs. Users can use the model to check the vulnerability in order to add specific protections. In the future the method will be improved continually by complete the details of the analysis strategy. Java language is still growing up and many software companies are continue supplying different components. It is the challenge for us to keep the completeness of analysis strategy.

REFERENCES

[1] Ed Roman and Rickard Oberg, The Business Benefits of EJB and J2EE Technologies over COM+ and Windows DNA, 1999, The Middleware Company
[2] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), pages 174–183, Long Beach, CA, USA, Nov 2005
[3] Jlint: a security tool for checking Java source code to find bugs , http://artho.com/jlint/
[4] lapse: security analysis tool for J2EE applications, http://suif.stanford.edu/~livshits/work/lapse/
[5] pmd: a security tool for checking Java source code to find bugs , http://pmd.sourceforge.net/
[6] findbugs: a security tool for checking Java code to find bugs , http://findbugs.sourceforge.net/
[7] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In Proceedings of PLDI'94, pages 242–256, 1994
[8] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In Proceedings of PLDI'01, pages 24–34, 2001
[9] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In Proceedings of PASTE'01, pages 73–79, 2001
[10] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In Static Analysis 9th International Symposium, SAS 2002, volume 2477 of LNCS, pages 180–195, 2002.
[11] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, May 1994. (DIKU report 94/19).
[12] Ondˇrej Lhot´ak. SPARK: A Flexible Points-to Analysis Framework for Java. Montreal: McGill University, 2003.
[13] Soot: a Java optimization framework. http://www.sable.mcgill.ca/soot/.
[14] SAMATE test cases. http://www.samate.nist.gov/SRD /view.php