2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

Detecting Vulnerable Java Classes Based on the Analysis of Java Library Call Graph

Hyosung Park[1], Chulwoo Park[1], SangBong Yoo[1], Kichang Kim[1]
[1]Information and Communication Engineering Department
Inha University
Incheon, Korea
E-mail : kchang@inha.ac.kr

***Abstract*—Java Virtual Machine relies on the SecurityManager class to prevent illegal system resource access by external Java code (e.g., Java Applet). The SecurityManager checks the access right of all Java classes in the call stack when a system resource is requested and rejects the request if any of these classes does not have the proper access right. But researchers have found a number of vulnerable Java system classes that allow user programs to bypass the SecurityManager. The identification of these vulnerable system classes is important in order to strengthen the security of Java Virtual Machine. However, finding them manually in the huge space of Java libraries is not trivial task. In this paper, we suggest a systematic technique of identifying vulnerable Java classes in a given Java library through the analysis of the call graph. We observe that there is a number of Java methods that force the SecurityManager omit the full examination of the Java call stack for performance issue and that these target methods are exploited through the vulnerable Java classes to enable the bypassing of the SecurityManager. Based on this observation, our technique enumerates all Java classes that call these target methods, analyzes the call chains and successfully detects vulnerable classes.**

Keywords-component; Java; security; call graph; SecurityManager; call stack

I. INTRODUCTION

When an Internet browser downloads and executes a Java applet, the SecurityManager of the Java Virtual Machine becomes activated [1, 6]. Once activated, the SecurityManager will check the permission whenever the applet accesses a system resource. However, since an applet can access the system resource via another set of system classes, the SecurityManager checks the permission of all the classes in the call stack at the time of system resource access. Checking all classes in the call stack guarantees strong safety for the Java system, however examining the permission mode of all classes in the call stack for every system access slows down the system considerably.

To resolve this performance issue, Java system allows a fast permission checking routine for some frequently used system methods such as forName, getField, getDeclardField, getMethod, getDeclaredMethod, findStatic, and findVirtual [2, 3]. For these methods, the SecurityManager doesn't examine the entire call stack but scan only some fixed number of immediate caller classes of these methods. If these caller classes all satisfy the permission requirement the system allows the resource access.

For example, "forName" method forces the SecurityManager to check only the immediate caller for permission checking. If this immediate caller was a system class, the SecurityManager will pass "forName" to proceed to load a new class without further permission checking[1]. This means if we can find some system class that accepts a class name and calls "forName" for us, we can load any class by calling "forName" indirectly through this system class. Because this is dangerous, all system classes that invoke "forName" are supposed to perform security checking by themselves, and those system classes that do not perform proper security checking become vulnerable classes.

Researchers have found a number of such vulnerable classes, e.g. AverageRangeStatisticImpl, ClassFinder, RMIConnectionImpl, etc., through manual examination of Java libraries [7]. However, identifying vulnerable classes in Java libraries manually is very expensive and error-prone. In the mean time new Java libraries are constantly being developed and used in many situations including IOT(Internet Of Things) environment [8]. In this paper, we suggest a systematic way of identifying these vulnerable classes. Given a set of Java libraries, our system can produce a list of candidate vulnerable classes and with further processing it can reduce this list down to one that contains actual vulnerable classes. Our technique first identifies dangerous system methods such as "forName", "getField", etc. which can allow bypassing of the SecurityManager with the help of some vulnerable classes. Then it examines the Java call graph to find all methods that can allow general user classes to access these dangerous methods. The classes that contain these methods but do not provide proper security checking become vulnerable classes. Our technique has been implemented and tested extensively for a set of Java system libraries. We were able to rediscover most of the pre-known vulnerable classes.

The rest of the paper is organized as follows. Section 2 surveys related researches. Section 3 explains the basics of security system in Java VM and an example of a vulnerable class that can break it. Section 4 provides a control flow graph

[1] "forName" is a method used to load a new class, and the SecurityManager is supposed to issue error message when a user class tries to load a restricted class such as "sun.awt.SunToolkit" with "forName".

978-1-5386-7975-3/18/$31.00 ©2018 IEEE
DOI 10.1109/Cybermatics_2018.2018.00311

to show how a vulnerable class is exploited. Section 5 describes the suggested technique to detect the vulnerable classes. Section 6 shows experimental results. Finally, Section 7 gives a conclusion and future research direction.

II. RELATED RESEARCHES

Most of Java vulnerable classes have been identified in the technical reports by Security Explorations [10, 12, 13, 11, 15]. History Based Access Control System [5] analyzes the behavior of a Java application and performs security check. It consists of two components, AM (Application Monitor) and PDP (Policy Decision Point), and can dynamically monitor dangerous system resource access. This technique fortifies the security of Java VM system, however it does not provide a way of identifying which classes are responsible for the security violation. Reynolds [9] uses Alloy Analyzer [4] to analyze the security of Java Byte code. He builds a Java VM model and analyzes it to identify any security vulnerability. His technique can identify security flaws in the Java VM model but not vulnerable Java classes themselves.

Zhao et al. [14] analyzes the security of Java Byte code via a Data-flow graph which shows the flow of input data starting from the input point until it reaches a point where it can affect the security of the system. This technique can tell the user whether a given input data for a given class can cause security problem to the system but not whether a particular class can cause security concern in general.

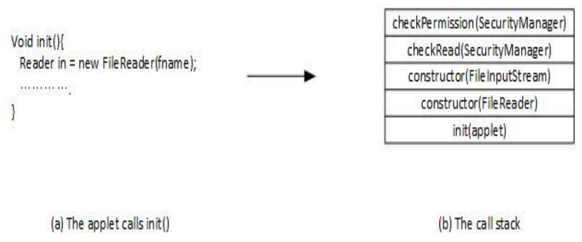

Void init(){
Reader in = new FileReader(fname);
....
}
checkPermission(SecurityManager)
checkRead(SecurityManager)
constructor(FileInputStream)
constructor(FileReader)
init(applet)
(a) The applet calls init()
(b) The call stack

Figure 1: Accessing a system resource and the corresponding call stack

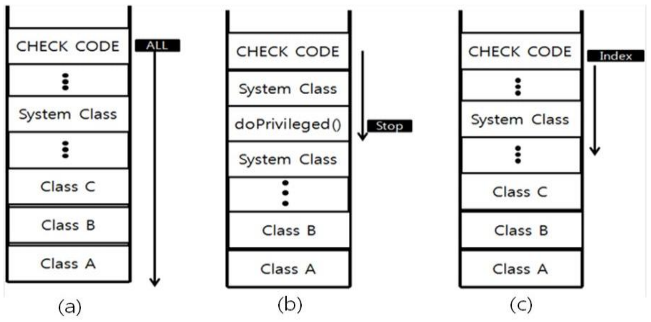
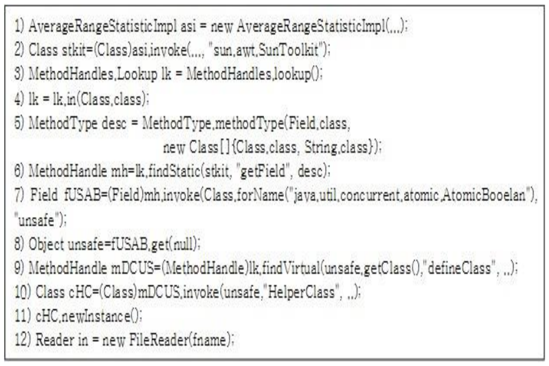


Figure 2: Three ways of call stack examination

III. JAVA SECURITYMANGER and VULNERABLE SYSTEM CLASSES

Java SecurityManager uses a call stack to perform security checking for an applet's system resource access. Figure 1 shows an example of a call stack[2] when the user applet calls init() which in turn calls the constructor of FileReader class. To instantiate FileReader class, the user applet needs a file permission, which a regular user applet doesn't have, and the SecurityManager can detect this violation and blocks the access. However, since the permission checking is done at the last stage of this call sequence (checkPermission method in Figure 1), the SecurityManager checks the file permission of all classes in the call stack and rejects the request if any of the classes does not have the right permission. This is to make sure that even if the Java applet has the file permission the system should reject the request if any of the subsequent methods invoked by init() does not have the proper permission. As mentioned before checking all the classes in the call stack for all system resource access is too costly and Java system allows abbreviated stack examination for some frequent system calls. Figure 2 shows three ways of call stack examination. We assume some method in Class A asks for a system resource and it invokes a sequence of system calls until the SecurityManager performs a security check on this request. "CHECK CODE" at the top of each stack denotes "checkPermission" method called by the SecurityManager. The stack examination in (a) is the most typical one where the SecurityManager checks the permission of all classes in the stack. The stack examinations in (b) and (c) are the abbreviated ones where the SecurityManager stops scanning the stack when it finds doPrivileged() method in the middle (the case of (b)), or when it has scanned some pre-determined "Index" number of methods (the case of (c)).

[2] In the figure, the top of the call stack shows the most recent method called, and the corresponding class of each method is shown inside a parenthesis.



```
1) AverageRangeStatisticImpl asi = new AverageRangeStatisticImpl(...);
2) Class stkit=(Class)asi.invoke(...., "sun.awt.SunToolkit");
3) MethodHandles.Lookup lk = MethodHandles.lookup();
4) lk = lk.in(Class.class);
5) MethodType desc = MethodType.methodType(Field.class,
                              new Class[]{Class.class, String.class});
6) MethodHandle mh=lk.findStatic(stkit, "getField", desc);
7) Field fUSAB=(Field)mh.invoke(Class.forName("java.util.concurrent.atomic.AtomicBooelan"),
"unsafe");
8) Object unsafe=fUSAB.get(null);
9) MethodHandle mDCUS=(MethodHandle)lk.findVirtual(unsafe.getClass(),"defineClass", ..);
10) Class cHC=(Class)mDCUS.invoke(unsafe,"HelperClass", ..);
11) cHC.newInstance();
12) Reader in = new FileReader(fname);
```

Figure 3: Attack code

The reasoning behind the partial stack examination in (b) is that doPrivileged() is one of the very restricted methods owned by privileged system classes usually not accessible from general user classes and therefore it should be safe to bypass further security checking. The case in (c) is more general way of avoiding full stack examination. Some frequently called system methods can indicate the depth of call stack examination, indicated as "Index" in the figure, such that the SecurityManager stop checking at that depth. For example, Class.forName() has Index=1 while Class.getMethod() has Index=2 and they both force the SecurityManager to perform only partial stack examination.

System methods that avoid full stack examination by indicating "Index" value are dangerous ones since user classes can call these methods via some other system classes. For example, by calling "forName" through some system class, the applet can avoid the full security check of the SecurityManager and can load some restricted class. Because of this potential problem, all system classes that can invoke these dangerous methods (e.g. forName, getMethod, etc.) should perform some proper security check by themselves before passing on the user's request. And if they fail to do the proper security checking, they become the vulnerable classes.

A skeleton of attack code that exploits these vulnerable classes is shown in Figure 3. In Figure 3, two vulnerable classes/methods are exploited: invoke() method of AverageRangeStatisticImpl class[3] (in Step 2) and findstatic() method of Lookup class[4] (in Step 6). These two steps allow the attacker to gain access to the "getField" method of the "sun.awt.SunToolkit" class, a restricted method of a restricted class. Step 2 is possible because asi.invoke(...,"sun.awt.SunToolkit") calls forName("sun.awt.SunToolkit"), and forName() loads a restricted class sun.awt.sunToolkit without complaining since forName is one of the dangerous methods mentioned above that let the SecurityManager check only the immediate caller in the call stack for the permission checking. In this case the immediate caller class is AverageRangeStatisticImpl which is a system class and has the proper permission. "forName" assumes the caller class has performed all necessary security check, thus there is no need for additional checking by the SecurityManager. The "invoke" method of AverageRangeStatisticImpl, unfortunately, does not perform necessary security checking. Step 6 proceeds without security blocking similarly. Once the attacker has "getField" method of "sun.awt.SunToolkit", he/she can obtain "unsafe" field of AtomicBoolean class (Step 7) and get the MethodHandle of "defineClass" (Step 9). Using this method handle, the attacker can load his own class, "HelperClass", as a system administrator, which will set the SecurityManager to NULL when instantiated. After this, the same FileReader(fname) shown in Figure 1(a) will now work with no security exception (Step 12).

IV. ACCESS PERMISSION CHECK FLOW CHART

Researchers have found and reported a set of vulnerable classes and we observe most of them can be expressed graphically in a control flow chart during access permission checking process. Figure 4 shows the flow chart for three dangerous methods: forName, getField, and getDeclaredField. At the top of the figure, we can see four entry points: forName() with three arguments, forName() with one argument, getField(), and getDeclaredField(). At the bottom of the figure we can see two exit points: ERROR and SUCCESS. ERROR node means the call exits with security exception raised, and SUCCESS node means the call exits with successful security passing.

The figure shows that there are three dotted lines that reach SUCCESS node. Solid lines that reach SUCCESS node represent normal full call stack scanning paths while dotted lines represent shortened ones. The figure shows that all of the four dangerous methods have shortened paths to SUCCESS node when the caller class loader is NULL. This is because these four methods require fast call stack scanning (type (c) in Figure 2 with Index=1), and the SecurityManager stops stack examination at the first stack component which is the immediate caller of the current method. Figure 5 shows additional flow charts for getMethod, getDeclaredMethod, findStatic and findVirtual which are also dangerous methods because they force the SecurityManager to shorten the call stack examination. Again there are dotted lines in the figure that bypass the full security checking. As mentioned before, this omission of security checking puts Java system in danger, but is necessary for system performance. All system methods

[3] This vulnerability is registered as CVE-2012-5076.

[4] This vulnerability is registered as CVE-2012-1726.

that allow bypassing of full security checking are dangerous, and it is the responsibility of the corresponding system classes to perform proper security checking by themselves before allowing the user classes to invoke these methods. The classes/methods that fail to do this become vulnerable classes/methods.

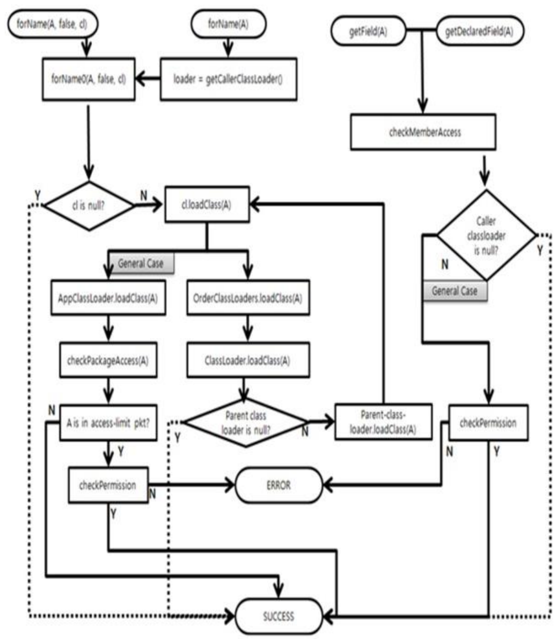
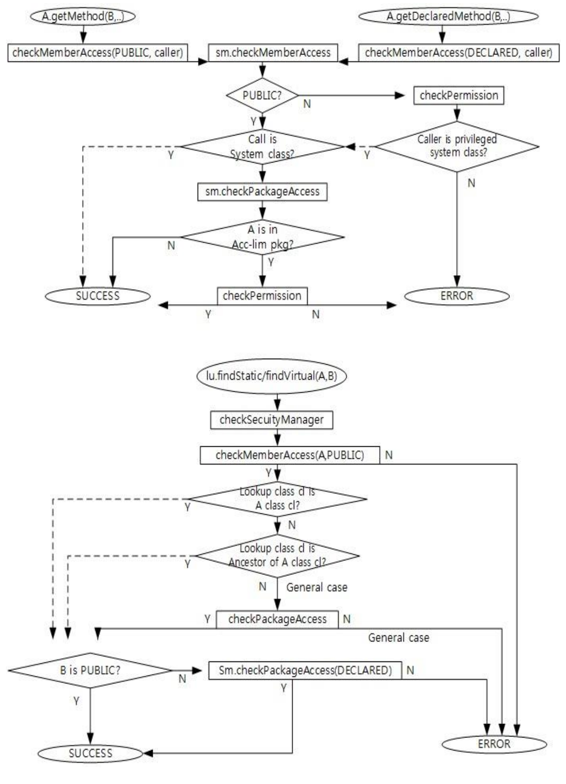


Figure 4: Access Permission Check Flow Chart for forName, getField and getDeclaredField



A.getMethod(B,..)
A.getDeclaredMethod(B,..)
checkMemberAccess(PUBLIC, caller)
sm.checkMemberAccess
checkMemberAccess(DECLARED, caller)
PUBLIC?
checkPermission
Call is System class?
Caller is privileged system class?
sm.checkPackageAccess
A is in Acc-lim pkg?
SUCCESS
checkPermission
ERROR
lu.findStatic/findVirtual(A,B)
checkSecuityManager
checkMemberAccess(A,PUBLIC)
Lookup class cl is A class cl?
Lookup class cl is Ancestor of A class cl?
General case
checkPackageAccess
B is PUBLIC?
Sm.checkPackageAccess(DECLARED)
SUCCESS
ERROR
Y
N

Figure 5: Access Permission Check Flow Chart for getMethod, getDeclaredMethod, findStatic and findVirtual

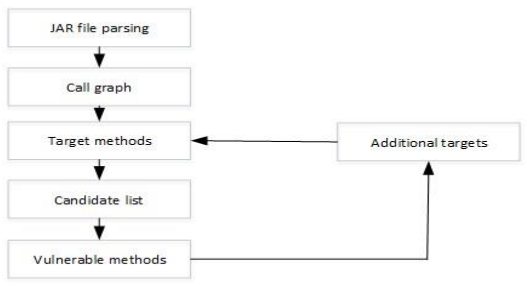


Figure 6: Identifying vulnerable classes/methods

V. SYSTEMATIC IDENTIFICATION OF VULNERABLE CLASSES

To identify vulnerable Java classes, we start from the set of dangerous system methods: forName, getField, getDeclaredField, getMethod, getDeclaredMethod, findStatic, findVirtual, etc. All Java classes that can invoke these methods are potentially vulnerable unless they provide proper secure checking process. Therefore, we build a call graph from the given Java library and analyze all the paths leading to the dangerous methods. The system classes that are accessible by external user classes and contain methods located on one of these paths will form a candidate list. We examine this list and identify those classes that do not have proper security checking process.

The identification steps are shown in Figure 6. In Step 1, the given JAR file will be parsed. We can extract all classes from a given jar file with com.sun.org.apache.bcel.internal.classfile package. JavaClass contains getConstantPool method with which we can get member methods of the corresponding class. In Step 2 a call graph will be built. All methods in all classes in the given jar file are extracted and represented as nodes in the call graph. Typical jar file contains a huge number of methods, and we use HashMap to implement a call graph instead of Adjacency Matrix. In Step 3, target dangerous methods are initialized. The target methods will be updated through the feedback as shown in the figure. We selected a set of target methods which are dangerous enough so that a system class without proper security measure may allow the user class to access the system resource through them.

In Step 4, we analyze the call graph and extract candidate method list. We search the call graph to collect the classes/methods that can invoke one of the target methods. Since a typical Java library contains a huge number of classes/methods, collecting all relevant classes/methods is not realistic. We need some filtering rules to limit our search. Table 1 shows the filtering rules we used.

Rule 1, 2, 3 and 4 are for filtering out methods that are not directly accessible from user class. Rule 5 removes methods that do not accept arguments in the form of String, method, or class since most of attacks involve passing class/method name as a string or passing method/class objects themselves. Rule 6 removes those methods that check whether the current SecurityManager is active because these methods can block attacking code with null SecurityManager. Rule 7 removes methods that check the class loader in the current context which represents the class loader of the original application. By checking the original class loader these methods can detect illegal access to system resources via some system classes. Rule 8 removes methods that require full call stack scanning. Table 2 shows the partial list of candidate list. For each target, the table shows the number of candidate methods at various

Table 1: Filtering rules

no	Rule
1	Remove methods of restricted classes
2	Remove methods of private classes
3	Remove “private” methods
4	Remove protected methods of “final” class
5	Remove methods whose input parameter type is not String, nor Methods, nor Class
6	Remove methods that call java.lang.System.getSecurityManager
7	Remove methods that call java.lang.Thread.getContextClassLoader
8	Remove methods that call java.lang.class.checkMemberAccess

Table 2: Number of candidate methods at various depth (partial list)

Target Method	Number of candidate methods at various depth
java.lang.class.forName	1(13) 2(47) 3(109) 4(192) 5(161)
java.lang.Class.getDeclaredField	2(4) 4(9)
java.lang.Class.getDeclaredMethod	2(4) 3(11) 4(6) 5(6)
java.lang.Class.getField	1(1) 2(4) 3(1)
jala.lang.Class.getFields	2(1) 3(1) 4(4)
java.lang.Class.getMethod	1(5) 2(14) 3(15) 4(113) 5(15)
java.lang.Class.getMethods	1(3) 2(4) 3(10) 4(2) 5(20)
java.lang.ClassLoader.getSystemClassLoader	1(7) 2(15) 3(56) 4(195) 5(206)
java.lang.reflect.Method.invoke	1(18) 2(20) 3(34) 4(138) 5(34)
....	....

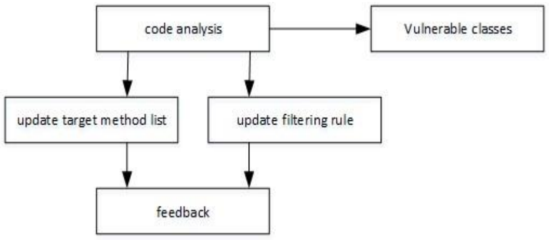


Figure 7: Code analysis and feedback

depths. The number of candidate methods at each depth is shown inside parentheses. For example, target "java.lang.class.forName" has candidate list of size 13 (at depth 1), size 47 (at depth 2), etc. As depth grows the number of candidate methods also grows as expected.

The final step in identifying vulnerable classes/methods is code analysis and feedback. Code analysis will result in a list of vulnerable classes and at the same time it may produce additional target methods and filtering rules. When a method of a class is analyzed as a vulnerable method, the method itself can be added to the target method list if external user class can call it in order to access system resource. On the other hand when a method is found to be non-vulnerable, the system calls invoked by this method to check security can be included in the filtering rules. Figure 7 shows this process. Additional methods and filtering rules will be fed back to the algorithm in Figure 6 to produce an expanded candidate list. Below are some of the methods that are added during this process.:
sun.reflect.misc.forName;
com.sun.corba.se.impl.util.JDKBridge.loadClass;
java.net.URLClassLoader.findClass

VI. EXPERIMENTAL RESULTS

We have tested the performance of our system in terms of prediction accuracy and the computation time to achieve it. To measure prediction accuracy, we have tested our system against a set of Java libraries that contain 25 pre-known vulnerable classes.

Table 3: A list of target methods detected and their depth in the call graph (partial list)

pre-known vulnerable classes	detected target methods and their depth
com.sun.beans.finder.ClassFinder	JL.Class.forName(2) JL.ClassLoader.getSystemClassLoader(2)
com.sun.beans.finder.FieldFinder	JL.Class.getField(1)
javax.management.modelmbean.Descriptorsupport	getSystemClassLoader(4)
javax.management.remote.rmi.RMIConnectionImpl	JS.AccessController.doPrivileged(2)
java.awt.Toolkit	JS.AccessController.doPrivileged(7)
com.sun.beans.finder.MethodFinder	JL.Class.getMethod(1)
CSOG.gmbal.ManagedObjectManagerFactory	JS.AccessController.doPrivileged(1)
CSOGESI.AverageRangeStatisticImpl	JL.reflect.Method.invoke(1)
CSOGESI.BoundaryStatisticImpl	JL.reflect.Method.invoke(1)
CSOGESI.BoundedRangeStatisticImpl	JL.reflect.Method.invoke(1)
CSOGESI.CountStatisticImpl	JL.reflect.Method.invoke(1)
CSOGESI.RanageStatisticImpl	JL.reflect.Method.invoke(1)
....	....

[5] In the table, CSOG stands for com.sun.org.glassfish, CSOGESI for com.sun.org.glassfish.external.statistics.impl, JL for java.lang, JS for java.security, and JLI for java.lang.invoke.

Table 3 shows the list of pre-known vulnerable classes in the first column[5]. The second column shows the target methods found by our system. It also shows the depth, inside parentheses, at which the vulnerable classes are detected. Out of 25 classes, 22 were detected correctly with 88% accuracy. Three following vulnerable classes are undetected:

java.beans.XMLDecoder,
com.sun.org.glassfish.gmbal.util.GenericConstructor,
com.sun.beans.finder.ConstructorFinder.

XMLDecoder allows user classes to access the system resource via java.lang.Class.forName, however there is a Java Native Method in the call chain reaching "forName" and our system cannot continue searching into the native libraries yet. GenericConstructor is omitted in the candidate list because of filtering rule 5, in Table 1, which filters out those methods that do not accept string or class/object input argument. GenericConstructor class has a vulnerable method "create" which simply uses a string member variable whose value is determined in the constructor. If our system deletes filtering rule 5, we can catch this vulnerability, however it will increase the size of the candidate list tremendously as shown later. Lastly, ConstructorFinder was missing because our target method list is not complete yet. Our system should add more target methods and filtering rules through the feedback shown in Figure 9.

To measure computation time, we have measured the time for call graph building and the time to search the deepest vulnerable method. Table 4 shows the partial result. We have experimented with 7 popular Java libraries (rt.jar, web logic.jar, com.bea.core.utils_2.0.0.0.jar, com.bea.core.weblogic.security.wls_1.1.0.0_6-2-0-0.jar, glassfish_jaxws.rt_2.0.0.0_2-2-5.jar, ws.databinding_2.0.0.0.jar, and javax.servlet_2.0.0.3_3-0.jar) and with 20 target methods. Total time for call graph building was about 31 seconds. Searching vulnerable methods takes longer as the depth increases. The longest depth was when the target was java.lang.Class.getMethods, and the time taken was about 6 minutes. Since the number of target methods is limited, the total searching time should be within manageable amount.

We have also examined the effectiveness of our filtering rules by observing the ratio of the number of remained candidate methods after filtering. Table 5 shows the results. For candidate methods found up to depth 5, the average filtering ratio was 94.47% meaning our filtering rule removes 94.47% of the methods.

Table 4: Target methods and searching time for the deepest vulnerable methods (partial list)

Target method	Time	Depth
java.lang.Class.forName	29.92ms	24
java.lang.ClassLoader.getSystemClassLoader	45.03ms	30
java.lang.Class.getMethod	215.78 ms	38
java.lang.Class.getDeclaredMethod	1.88ms	14
....	....	.....

Table 5: Effectiveness of filtering rules (partial list)

	number of candidate methods (after filtering / before filtering)				
target method	depth 1	depth 2	depth 3	depth 4	depth 5
forName	13/1489	47/1602	109/2444	192/2606	161/3244
getDeclaredField	0/23	4/27	0/31	9/76	0/176
getDeclared Method	0/70	4/100	11/137	6/214	6/400
getField	1/41	4/66	1/88	0/104	0/106
....	.....	.....	.....	.....	.....
filtering ratio per depth	79.63 %	96.69 %	94.16 %	90.79 %	96.27 %
ave. filtering ratio	94.47%				

VII. CONCLUSIONS

The identification of vulnerable Java classes/methods in the context of Java mobile code has been pursued by numerous researchers, but the process was mainly of manual inspection because of the complexity of the Java security checking system. This paper suggests a systematic technique to expose vulnerable Java classes/methods existing in a given set of Java libraries. The technique consists of three steps: selecting target methods, building call graph and collecting candidate vulnerable methods, and extracting vulnerable methods based on filtering rules. The paper shows a list of target methods and the flow graph that shows how these target methods bypass the full security checking. It explains the process of building call graph and candidate list. And then it enumerates a set of filtering rules. We have performed an extensive experimentation on our system with a set of popular Java libraries. Our system was able to identify 88% of pre-known vulnerable methods within less than 17 minutes, and our filtering rules were able to reduce the size of candidate list by 94%.

ACKOWLEDGMENTS

This study was supported by research grants from Inha University and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (grant no: 2018019591).

REFERENCES

[1] Dean, D., Felten, E. W., Wallach, D. S., Balfanz, D., “Java security: Web browsers and beyond”, Internet besieged: countering cyberspace scofflaws, ACM Press, 1997.

[2] Goichon, F., Salagnac, G., Frenot, S. “Exploiting Java code interactions”, Technical Report RT0419, 2011 (https://hal.inria.fr/hal-00652110, 2011).

[3] Gong, L., Mueller, M., Prafullchandra, H., Schemers, R., “Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development,” Proceedings of USENIX Symposium on Internet Technologies and Systems, Dec. 1997.

[4] Jackson, D., “Alloy website”, http://alloy.mit.edu

[5] Martinelli, F., Mori, P. (2007). "Enhancing Java security with history based access control", Foundations of security analysis and design IV, 2007.

[6] Oaks, S. "Java security", O'Reilly Media, Inc. 2001.

[7] Oh, J. W., "Recent Java exploitation trends and malware", Black Hat USA, 2012.

[8] Oracle, "The Internet of Things: Manage the Complexity, Seize the Opportunity", http://www.oracle.com/us/solutions/internetofthings

[9] Reynolds, M. C., "Lightweight modeling of java virtual machine security constraints", Lecture Notes in Computer Science, vo. 5977, 2010.

[10] Security Explorations. "Security Vulnerabilities in Java SE Technical Report", April 2012.

[11] Security Explorations, "Google App Engine Java security sandbox bypasses", December 2014.

[12]Security Explorations, "Security vulnerabilities in Oracle Java Cloud Service", January 2014.

[13] Security Explorations, "Security vulnerabilities in Oracle Database Java VM", June 2014.

[14] Zhao, G., Chen, H., Wang, D., "Data-flow Based Analysis of Java Bytecode Vulnerability", Proceedings of Web-Age Information Management, 2008.

[15] WhiteHat Security, "Application Security Statistics Report,", info.whitehatsec.com, May, 2017.