Contemporary Engineering Sciences, Vol. 7, 2014, no. 24, 1383 - 1388
HIKARI Ltd, www.m-hikari.com
http://dx.doi.org/10.12988/ces.2014.49170

Java Vulnerability Analysis with JAPCT: Java Access Permission Checking Tree

Hyo-Seong Park

Inha Univ. Information & Communication Engineering, Korea

Young-Chan Lim

Inha Univ. Information & Communication Engineering, Korea

Chul-Woo Park

Inha Univ. Information & Communication Engineering, Korea

Luna Clout

RDIT, Korea

Ki-Chang Kim

Inha Univ. Information & Communication Engineering, Korea

Abstract

Java Security Manager is automatically enabled when the web browser downloads a Java applet. Java Security Manager monitors the behavior of the applet and raises an exception and blocks further process when it tries illegally to access system classes or methods. In order to test the legality of the access request, Java Security Manager steps through a tree-like decision path. In this paper, we analyze various kinds of malicious Java applet code and classify them according to the

path they take to avoid the blockage by Java Security Manager. The result of this classification is JAPCT (Java Access Permission Checking Tree). We believe JAPCT will greatly enhance our ability to understand Java security vulnerabilities in the past and to predict possible security problems in the future Java Virtual Machine.

Keywords: Java Security, Security Manager, Access Permission

1 Introduction

Java is easy to program and provides intuitive web interface. As the Internet grows, these properties of Java made it spotlighted as suitable programming language for the Internet environment. Most of Web browsers activate Java Security Manager in order to execute Java applet in safe environment [1] [2]. When the Java bytecode is executed, Java Security Manager protects system by generating exceptions for non-authorized actions [3] [4]. But there are various paths through which we can bypass the permission check code of Security Manager when we use some vulnerable Java system classes. In this paper, we analyze Java API classes that have vulnerabilities as reported in the literature, and classify them according to specific patterns. Also we analyze Security Manager and show how it protects the local system. Finally we suggest JAPCT (Java Access Permission Checking Tree) to express various Java permission checking path and to show where the vulnerabilities exist in the path. Section 2 of this paper explains how to disable the Security Manager and how to classify vulnerabilities - we assume the readers have some knowledge on Java Security Manager. In section 3, we describe two different kinds of access permission checking: class access permission checking and method access permission checking. We explain JAPCT in section 4, and conclusion in section 5.

2 Disabling Security Manager

Once the Security Manager is activated, user classes cannot deactivate it, but there are some ways to deactivate the Security Manager using vulnerabilities of System classes [5] [6]. In most cases, these vulnerabilities arise from the lack of security checking process in the vulnerable Java API classes. The security checking path is omitted to enhance the execution speed of the classes and most of time it is safe [7], but there exist cases where this omission becomes fatal when combined with the vulnerabilities of other classes [6]. System.getSecurityManager method verifies the status of Security Manager. When Security Manager is not activated, the return value is NULL. When it is activated, the return value is any

value other than NULL. It needs two steps to deactivate Security Manager. The first step is to load a restricted class and the second step is to use the method of the restricted class to disable the Security Manager. It can change the return value of the System.getSecurityManager method to NULL using some classes that can deactivate the Security Manager of the loaded restricted class.

3 Vulnerability Classification

Vulnerability classification is split into two large parts. The first is the vulnerability to access restricted classes, and the second is the vulnerability to access the methods of the restricted classes. The usual method we can use to load general classes is Class.forName. The third parameter of forName(String name, Boolean initialize, ClassLoader loader) method is the ClassLoader of the class that has called forName method. The ClassLoader of a user class cannot load a restricted class, but the ClassLoader of a system class can load any restricted class. If we can find a system class that calls forName method we can use it to load restricted classes. The techniques to access the method of restricted classes are classified into three cases: using getMethod method, using getDecalredMethod, and using Lookup class which is an inner class of MethodHandles class.

4 Java Access Permission Checking Tree

The vulnerabilities classified in Section 3 can be expressed in the form of Access Checking Tree. There are two kind of access checking tree: class access checking tree and method access checking tree.

4.1 Class Access Checking Tree

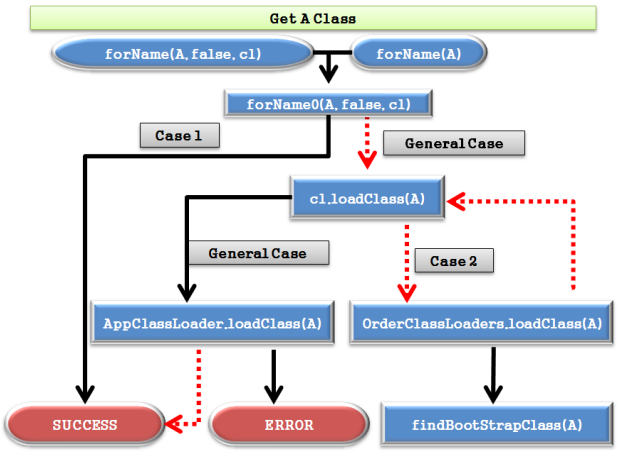

Fig. 1 Class Access Checking Tree

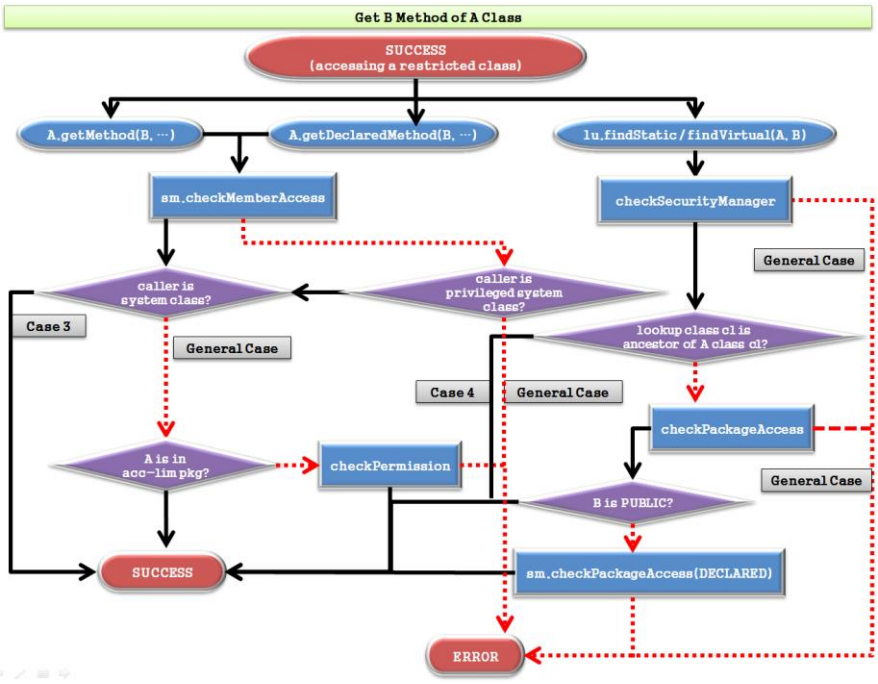

Fig. 2. Method Access Checking Tree

Fig. 1 refers to Class Access Checking Tree. At the branch point, it divides into two lines - normal and dotted line. In case of vulnerability, we marked 'CaseN' and in normal case 'General case'. forName(A), the 1-argument forName(), and forName(A, false, cl), the 3-argument forName(), both call forName0(A, false, cl), where "cl" is the classloader of the caller. forName0() in turn is handled by load_instance_class(A, cl, ...). load_instance_class(A, cl, ..) is the function that checks the access permission of the caller for class "A". It bypasses the security checking when "cl" is NULL because "cl" represents the caller classloader and is NULL when the caller is a system class. Since a system class should be able to load any class, the java system bypasses the security check for a system class.

"Case 1" in Fig. 1 represents the vulnerable path. Simply passing NULL in the third argument of forName(A, false, cl) will not bypass the security check since forName0() itself checks whether the caller is a system class or not when "cl" is NULL. However when forName(A) is called by a system class, JVM assumes the caller has the right permission and invokes forName0() with NULL in the third argument, effectively bypassing the security check. Therefore any system class that calls forName(A) and can be called by a regular user class is a vulnerable class. The user can use a simple user class that calls one of these vulnerable system classes passing the restricted class name in "A" such as "sun.awt.SunToolkit", which, once obtained, can be used to perform various tasks

allowed only for the administrator. The researchers have found 4 such system classes: AverageRangeStatisticImpl in com.sun.org.glassfish.external.statistics.impl package, findClass in com.sun.beans.finder.ClassFinder package, getDefaultToolkit in java.awt.Toolkit package, and invoke() in java.lang.invoke.MethodHandle package.

"Case2" in Fig. 1 is another vulnerable path with regard to forName(). When "cl" in forName0(A, false, cl) is not NULL, it is usually AppClassLoader and this class loader calls "checkPackageAccess(A)" to check the access permission for class name "A". Restricted classes such as "sun.awt.SunToolkit" are included in the access-limited package and are rejected here. However if "cl" is "OrderClassLoaders", the java system checks the parent class loader again and bypasses the security check if the parent class loader is NULL. We can call "new OrderClassLoaders(NULL, cl)" to pass NULL in the parent class loader. However, calling OrderClassLoaders directly from a user class will cause a security exception. We need to find a system class and method that calls it as a privileged class. RMIConnectionImpl is such a class and we can use this class to bypass the security check again.

4.2 Method Access Checking Tree

Fig. 2 shows Method Access Checking Tree. The 'SUCCESS' in the start point means accessing a restricted class was successful. The ‘SUCCESS’ in the below means getting a method is successful. There are three ways of approaching to a method of restricted classes as shown in Section 3: via getMethod(), getDeclaredMethod(), or findStatic()/findVirtual(). "getMethod()" allows us to bypass the security check when the caller is a system class. Case 3 shows the path where getMethod() fails to block illegal access properly. "getDecalredMethod()" also allows us to bypass the security check when the caller is a privileged system class. Fig. 2 shows the failure of this method in Case 3. "findStatic()" or "findVirtual()" let us pass the security test when the lookupClass is a system class. Case 4 shows the path where the vulnerability in Lookup class is used to obtain a method.

5 Conclusion

New Java vulnerabilities are being published but its principles are similar as before in that they bypass Security Manager of the Java system. In this paper, we have analyzed the inner-working mechanism of Security Manager to expose the vulnerabilities of Java API and have designed JAPCT to capture the necessary security checking steps that existing vulnerable Java API classes have missed. JAPCT Also shows the process of checking access permission to classes and methods

allowing us to identify the paths that have potential vulnerabilities.

Acknowledgements. This work was supported by Inha University and a National Research Foundation of Korea grant funded by the Korean Government.

References

[1] Scott Oaks, ″Java Security 2nd Edition″, O′REILLY′, pp.21-112, May 24, 2001.

[2] Wallach, Dan S., and Edward W. Felten. "Understanding Java stack inspection." Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on. IEEE, 1998.

[3] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2", 1997.

[4] Drew Dean, Edward W. Felten, Dan S. Wallach, Dirk Balfanz, "Java Security: Web Browsers and Beyond, February 24, 1997.

[5] Li Gong, "Java 2 platform security architecture, October 2, 1998.

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

[7] Almut Herzog, Nahid Shahmehri, Performance of the java security manager, 18 August 2004.

Received: August 27, 2014