ISSN(Online) :2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

An Appraisal to Secure JAVA Programming

Vamsi Krishna Myalapalli[1], Sunitha Geloth[2]

Open Text Corporation, Mind Space IT Park, Hitec City, Hyderabad, India

ABSTRACT: Security engineering in a program or application exhibits a crucial part, as it prevents or minimizes security holes. Application developers should secure the application code before being deployed or used in a productionenvironment.As such the proposed model is envisioned to assist the JAVA programmers to secure and enhance the JAVA based application(s). This paper elucidates miscellaneous techniques to escalate JAVA application program security and can serve as asecurity engineering and overhauling tool for the JAVA application programmers. Our experimental denouements designate that security bugs are reduced and maintainabilityis enhanced.

KEYWORDS: JAVA Security; JAVA Tuning; JAVA Optimization; JAVA Best Practices; JAVA Security Engineering; Robust JAVA; Invulnerable JAVA.

I. INTRODUCTION

The Java programming language together with its runtime environment is well known to provide a lot of security features for applications written in Java and to the environment, in which Java applications are deployed. Java prevents a programmer from writing code that could produce buffer−overflows or overwrite data unintentionally. Errors that may occur from freeing memory are not possible; as such tasks are delegated to an automatic garbage collection mechanism. Moreover, as elementary data types are strictly defined, Java source code is compiler independent. Already at compile time, access to variables and methods can be checked and type checks are possible for widening casts of classes, interfaces and objects. Byte-code (class files) that has been generated by a correctly implemented Java compiler is therefore safe against its misuse to attack someone running such code in a (correctly implemented) runtime environment.

When it comes to enforce security for a running application and its environment, the Java runtime environment (JRE) with its Java virtual machine (JVM) plays a major role. Its first duty is to check untrusted byte code (class files) for compliance with the Java virtual machine specification, a task that is not necessary for trusted code. These checks guarantee the protection against all sorts of attacks that might corrupt the memory of a process running untrusted byte-code. The JVM also complements the Java compiler with those checks that can only be performed at runtime, like boundary checks or type checks when applying narrowing cast operators. JRE allows configuration and enforcement of detailed policies which encompass the exact permissions to be granted to a particular application for accessing system resources. When policies are configured it is possible to mandate that code has to be signed by an accepted authority in order to be granted specific permissions. Hence code signing (including the tools necessary to actually get some code signed) is another aspect of "Java Security".

II. BACKGROUND AND RELATED WORK

Java security relies heavily on the execution model as specified in the Java language specification and in the Java virtual machine specification. In particular it relies on byte-code verification, class loading mechanisms, securitymanagers and access controllers.

High Performance JAVA Programming [1] and Minimizing Impact on Java Virtual Machine via JAVA Code Optimization [2] explained optimizing JAVA programs, hence contributing to tune JAVA based applications. High Performance C Programming [3] explained tuning programming statements in logical way. An Appraisal to Optimize SQL Queries [4] and High Performance SQL [5] explained several methodologies to optimize and secure SQL queries.

ISSN(Online) :2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

This paper is aimed at proposing secure techniques, which could serve as benchmarking tool for ensuring zero bugor vulnerabilities in JAVA applications.

III. PROPOSED MODEL

Secure programming is not possible without obeying some programming practices.The following are holistic techniques we have implemented and ensured secure and robust JAVA applications.

1) ***Garbage Collector:***

Design classes, such that the corresponding objects are of reasonable size, and instantiateobjects only at the time they are really needed.

Mark (large) objects for garbage collection as soon as possible.This can be done by setting all references of such object(s) to null.

One of Java's prominent language features is the fact, that memory allocation and destruction is managed by the Java runtime execution engine. Nevertheless, memory consumption can be monitored and to some extend influenced by an application. If we are concerned about memory consumption, we should allocate only as much memory as necessary, and give the garbage collector the opportunity to free memory by setting references to unneeded objects to null. On the other hand, it is not good to call System.gc(), as the garbage collector is well optimized to work on its own. If a more sophisticated interaction with the garbage collector is needed, (e.g. when programming memory) sensitive caches, classes from the package java.lang.ref can be employed.

Exceptions #(2-3):

2) ***Never Code try/catch Block with an Empty Catch Block:***

Java provides a comfortable framework for error handling with its build−in exceptionclasses. But to not run into erroneous situations that bypass undetected or that are difficultto trace back, exceptions should never be silently catched. Even if we do not ever expectany exception to occur in the try block, it is safe programming practice to add some code tosignal an exceptional case in any case, for example by adding some call to theprintStackTrace() method. This is particularly important if we catch general exceptions(class Exception) as otherwise a RuntimeException would be catched unnoticed.

3) ***Use a finally block to Release Resources that have been claimed in a Try block:***

If system resources are allocated in a try block, it has to be guaranteed that these resourcesare released if an exception occurs. This can be done in catch or finally blocks. If we use thecatch blocks to release claimed resources, we have to take care to release the resources inthe try block as well. Using finally blocks for such purposes unifies and simplifies the releaseof resources.

Serialization and Deserialization#(4-7):

Java provides a standardized mechanism to serialize objects by implementing the Serializableor Externalizable interface. Serialization may be used for lightweight persistence and forcommunication via sockets or RMI.

4) ***Do not Directly Serialize Variables that contain Sensitive Data:***

While being serialized, an object is outside of the control of the JVM and might bevulnerable to modification or inspection. The serialization of variables can be prohibited bydeclaring them as "private transient". On the other hand, if sensitive data has to be serialized,this data (or the whole serialized object) should be encrypted.

5) ***Implement Checks in the readObject method of Serializable Classes:***

The readObject method of a serializable class is effectively a public constructor, which may have a lot of parameters, whose values are all coming from a serialized object. It is secure programming practice to assume that these parameters might have been modified by an adversary or just by some system failure. Hence, validity checks of these parameters have to be implemented in a serializable class's readObject method to guarantee that the de-serialized object is in a consistent and valid state.

ISSN(Online) :2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

6) Use the transient keyword for fields that contain Direct Handles to System Resources (suchas file handles):

It would be bad programming style to use handles to system resources after deserialization,as a modification of the serialized object by an attacker (or just by some system failure)would result in improper access to resources after deserialization. Hence, from the start, it isgood programming practice to avoid serialization of such handles.

7) When Writing Own Serialization methods, do not use DataOutput orDataInput Methods as they may Expose Byte Arrays to Malicious Code:

If defining serialization methods for some classes, attention should be paid not to exposeinternal (mutable) variables in such a way that they could be modified by malicious code. Inparticular the write methods for byte arrays from the ObjectOutputStream class should beavoided, as these methods could be overridden by an attacker (simply by extendingObjectOutputStream). (Note however that the standard serialization mechanism using thefinal method writeObject has its own implementation to serialize byte array membervariables.)

Java Native Interface#(8):

8) Use only Well Evaluated Native Libraries.

Java provides the native programming interface to interoperate with applications and libraries written in other programming languages. Coding to the JNI may open program to vulnerabilities inherent to applications/libraries written in other programming languages. The native applications/libraries should be written under some programming guidelines, specific to the particular programming language. Evaluation of these applications/libraries has to be executed very cautiously.

Security Critical Modules and Tasks#(9-12):

9) Separation of Security Critical Modules:

a) Identify security critical modules/tasks during the specification and design phase.

Identifying security relevant tasks, such as the generation of PIN numbers, in early phases ofthe software development process is mandatory in order to define clean interfaces to suchfunctionalities. Furthermore, having identified such tasks/modules, implementation andreview of such modules can be delegated to software engineers with appropriate skills.

b) Regularly review all code to check whether security critical methods are only implementedin distinguished packages/classes.

To assure some security standard of the overall product it is mandatory that all securitycritical tasks are delegated to dedicated modules. Some poor ad hoc implementation of sucha task may ruin the overall security goals completely.

10) Cryptographic and Security Relevant Packages:

a) Using well established or evaluated packages for security critical algorithms and services.

b) Do not implement some proprietary cryptographic or security relevant routines.

It is generally good practice to only use well established and investigated algorithms forcryptographic purposes. Proprietary algorithms have very often been the reason forhaving security holes.

11) Sensitive Data:

a) Do not store some secrets in the code (like pins or secret keys for encrypting user input).

For anyone who may obtain a copy of the code, secrets compiled into the code are NO secrets! This is particularly true for Java bytecode, which can easily be decompiled.

Obfuscation tools may provide some means to complicate the analysis of a program or to impede others to simply reuse the code, but it does not help to hide secret data.

b) Take special precautions, when processing sensitive user input data like PINs or passwords to clear such data as soon as possible.

It is good programming practice to store sensitive data in an mutable object, say some array of bytes or chars and to clear the data (setting the entries of the array to 0), as soon as the sensitive data has been

ISSN(Online) :2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

processed. (Note: Setting the reference to the object to null, does not remove the data immediately from the heap. To the contrary this transfers control to the garbage collector, who might not need to release the corresponding memory at all.)

c) Take care not to echo sensitive data to the UI.

When users have to input some sensitive data, it is good practice to not echo the keystrokes, either by not displaying any characters or by displaying some fixed char for every keystroke (often a '*'). For example in a Swing GUI application we may use javax.swing.JPasswordField for such a purpose. Unfortunately there is no appropriate method contained in the java.io package, that allows turning echoing off, when input data is read from the command line. (A particular bad example of an application that echoes sensitive data is the keytool application that ships with the JDK.)

12) Pseudo Random Data:

Never use java.util.Random for the generation of random numbers in security relevantcontexts.It is not at all safe to use java.util.Random forthe generation of random numbers in such security critical applications, as the sequence ofnumbers returned by java.util.Random is predictable, once any of its values has beenobserved.

Security Mechanisms#(13-16):

13) Byte-Code Verification:

a) Take into account that the set of trusted classes does depend on the deployment of anapplication (in particular on the version of the JVM).

As bytecode verification is an integral part of the enforcement of the security of a Javaapplication, it is important do know what classes are checked, when running a Javaapplication. To speed up performance, virtual machine implementation does not check"trusted" classes. Unfortunately it is not specified what classes are "trusted" classes.An important verification that can be performed at runtime is to check access restrictions(for example denying access to private members of other classes). But such checks are notalways performed by a JVM. To enforce such checks for virtual machines shipped withJDKs the −verify option has to be used.

b) Make sure that application runs in an environment, where access restriction is checked by the byte-code verifier (if necessary).

14) Policies and Security Managers:

Supply policy files or implement a SecurityManager class, such that applications run in a sandbox with minimal necessary permissions granted.

It is good practice to supply most strict security confinements for Java applications. This makes it more difficult for an attacker to misuse the application. Furthermore, the user will usually not know what permissions have to be granted for a particular application. The security confinements can be supplied by an appropriate policy file or by an implementation of a subclass of the SecurityManager class. Supplying a policy file should usually suffice. But note that the permissions granted to the application ultimately then depends on the correct deployment of the policy files.

15) Privileged Code:

a) Keep privileged code blocks as short as possible.

b) Do not allow parameters to privileged blocks that could be misused by untrusted code to gain access to some resources.

Generally, when the default security manager is enabled and a method accesses some resource, the access controller checks, whether all code traversed by the execution thread up to that point has the appropriate permissions. Using the doPrivileged methods from the AccessController class allows circumventing these checks, i.e. permission of calling the method is granted only on the basis of the permissions given to the code for the method in question. In other word untrusted (possible malicious) code calling the method containing the privileged block gains the privileges of the called method while executing this call. Therefore it is important to program privileged blocks very carefully.

ISSN(Online) :2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

16) Packages:

Ship sealed JAR files.

Packages can be protected from insertion of untrusted classes and from access of package private members by untrusted code. There are two alternatives to enforce such protections. One possibility is to use the "package.definition" and "package.access" properties in the java.security configuration file. The other possibility is to ship sealed JAR files.

17) Input Data Validation:

a) Always validate input to public methods.
b) Provide utility methods for input data validation and transformation purposes.
c) Consider to automatically generate input data validation methods.

18) Performance Optimization:

Optimize performance only after profiling.

Programs written in Java are not as performant as programs written in C/C++ or inassembly code. But even if performance may be an important issue, it should not beoveremphasized in the initial programing phase. Otherwise, the code may get quite obscureand difficult to maintain and review. When the program can be tested and profiled, code thatis responsible for performance bottlenecks can be identified and optimized.

19) Assertions and Debug Traces:

a) Adding debug traces to the code.

During software development and testing phase(s) assertions and debug traces may provide valuable hints about the location and reason for bugs occurring in the running program. Also statistics and profiling information may be included in debug traces. Debug traces not only help a lot during the development phase, but for example also do provide means for quality engineers to follow the program execution path and allow a first analysis when it comes to narrow down a bug. In general thorough testing, which is an integral part in the production of secure software, will not be possible without sufficient trace information.

b) Use the technique of "Dead Code Elimination" to build debug and release versions.

Some final static variables should be used to flag whether assertions should be enabled and what trace level has to be used. As an example see the following class DebugConstants and its usage in the class Demo.

Using global static variables, allows generation of different versions of the byte code, by simply changing the values of the static variables and recompiling the code. The final release version may not contain any trace generating code at all, making the code as small as possible and hiding information that may provide an attacker with valuable hints about the execution of the code.

20) Mutable Objects:

a) Make objects immutable whenever possible.

Immutable objects are objects whose state (value of member variables) cannot be changedafter construction. It is a good (safe) coding rule to reduce the number of modifiablevariables as much as possible. Obviously, immutable classes are optimal with respect to thisprinciple. Furthermore, following the principle of designing and using immutable objects,also is of advantage with respect to synchronization issues.

b) Never save references to mutable objects in some member variable, if such a reference isa parameter of a public constructor or method.

A necessary condition for programming secure software is the implementation of clearinterfaces and strict data encapsulation. The possibility to change member variables of anobject from the outside does represent a severe violation of this basic rule. Hence, ifreferences to mutable objects are given as parameters to a public constructor or method, theconstructor/method should never directly copy any such reference to its member variables.

If the value represented by the referenced object has to be stored, a newly constructed copy(a clone) of this object has to be constructed first, to finally store a reference to this clone.

c) Never return references to mutable member objects from public methods.

ISSN(Online) :2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

Again, the possibility to change member variables of an object from the outside doesrepresent a severe violation of secure programming principles.
If the value of a member object has to be returned, a deep copy (clone) of this object has tobe constructed first, to finally return a reference to this clone.

21) Static Variables:

a) Deter using non−final public static variables.

While final static variables are treated as constants (already at compile time), values ofnon−final public static variables can be changed from everywhere during runtime, making itimpossible to guarantee a consistent state of such variables. Furthermore such values couldas easily be manipulated by some attackers code, that exploits application. Therefore theaccess scope of non−final static variables should be declared to be private, and public getand set methods should be defined to access such variables. The set methods have toimplement checks to guarantee a consistent state of the corresponding variables.

b) Make sure to recompile all classes if some final static variable has been changed.

Values of final static variables are substituted already at compile time. Hence, if some classuses a static final variable that has been changed; this class must be recompiled in order toreflect the changes in the byte-code.

22) Inheritance:

a) Make classes and methods final whenever possible.

Be aware that non final classes and methods could be extended/overridden by some non-anticipated subclasses. This may lead to unstable code or may even open up some possibility to an attacker to misuse the application by extending some classes. While inheritance is an integral aspect of Java as an object oriented programming language, it contradicts secure programming principles (principle of least privileges).

b) Program by contract using the template method design pattern.

When implementing some public method, we should "program by contract", that is we should first check all preconditions (input validation), then implement the essential routines, and finally check post-conditions (if necessary) before returning. To guarantee that all implementations of a overridden method perform identical precondition and post-condition checks, the following template method design pattern should be used: Declare public methods as final methods in the base class. Implement the precondition and post-condition checks in the base class. Then, to do the essential routines, let the method call some protected auxiliary method. It is this auxiliary method that can then be overridden in some derived classes.

c) Do not call non−final methods from constructors.

Initialization of classes, interfaces and objects can raise some subtle questions. To guarantee a well-defined state after initialization of classes and objects, some rules should always be followed. In particular, calling non−final methods from initializing code may introduce unforeseeable dependencies of the initial state of the class with subclasses that override the called method.

23) Access of Variables and Methods:

a) Declare variables and methods to be private whenever possible.

As it is good (safe) programming practice to always limit access as much as possible (principle of least privileges), everything should be declared to be private by default.

b) Make public access to variables only possible via accessor methods (get/set methods).

Making variables private and limit access only via accessor methods allows differentiating between reading (get) and writing (set) access rights. Furthermore this defines a central point, where checks can be implemented, before allowing access/modification of the variable's value.

24) Minimise the Number of Permission Checks:

Java is primarily an object-capability language. SecurityManager checks should be considered a last resort. Perform security checks at a few defined points and return an object (a capability) that client code retains so that no further permission checks are required.

ISSN(Online) :2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

25) Design APIs to Avoid Security Concerns:

It is better to design APIs with security in mind. Trying to retrofit security into an existing API is more difficult and error prone. For example, making a class final prevents a malicious subclass from adding finalizers, cloning, and overriding random methods. Any use of the SecurityManager highlights an area that should be scrutinized.

26) Avoid Duplication:

Duplication of code and data causes many problems. Both code and data tend not to be treated consistently when duplicated, e.g., changes may not be applied to all copies.

27) Restrict Privileges:

Despite best efforts, not all coding flaws will be eliminated even in well reviewed code. However, if the code is operating with reduced privileges, then exploitation of any flaws is likely to be thwarted. The most extreme form of this is known as the principle of least privilege. Using the Java security mechanism this can be implemented statically by restricting permissions through policy files and dynamically with the use of the java.security.AccessController.doPrivileged mechanism.

28) Establish Trust Boundaries:

In order to ensure that a system is protected, it is necessary to establish trust boundaries. Data that crosses these boundaries should be sanitized and validated before use. Trust boundaries are also necessary to allow security audits to be performed efficiently. Code that ensures integrity of trust boundaries must itself be loaded in such a way that its own integrity is assured.

29) Evade Exposing Constructors of Sensitive classes:

Construction of classes can be more carefully controlled if constructors are not exposed. We have to define static factory methods instead of public constructors. Support extensibility through delegation rather than inheritance. Implicit constructors through serialization and clone should also be avoided.

30) Prevent Constructors from Calling Methods that can be Overridden:

Constructors that callingoverridable method(s) give attackers a reference to this (the object being constructed), before the object has been fully initialized. Likewise, clone, readObject, or readObjectNoData methods that call overridable methods may do the same. The readObject methods will usually call java.io.ObjectInputStream.defaultReadObject, which is an overridable method.

31) Avoid Dynamic SQL:

Dynamically created SQL statements including untrusted input are subject to command injection. This often takes form of supplying an input containing a quote character (') followed by SQL. For parameterised SQL statements using JDBC, use java.sql.PreparedStatement or java.sql.CallableStatement instead of java.sql.Statement. It is better to use a well-written, higher-level library to insulate application code from SQL. When using such a library, it is not necessary to limit characters such as quote ('). If text destined for XML/HTML is handled correctly during output, then it is unnecessary to disallow characters such as less than (<) in inputs to SQL.

32) Super-class vs. Sub-class Behavior:

Subclasses do not have the ability to maintain absolute control over their own behavior. A superclass can affect subclass behavior by changing the implementation of an inherited method that is not overridden. If a subclass overrides all inherited methods, a superclass can still affect subclass behavior by introducing new methods. Such changes to a superclass can unintentionally break assumptions made in a subclass and lead to subtle security vulnerabilities.

ISSN(Online) :2319-8753
ISSN (Print) : 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 9, September 2015

IV. CONCLUSION

The techniquescould serve as best practices for JAVA programmers in the form of benchmark as well as White box testing for questing security bugs.JAVA secure programming plays a key role in the programming microcosm, due to the reason that JAVA is used as programming language in many of the commercial application development environments. Subsequently, enforcing these techniques comprehensively minimized the rate of bugs and thus security is enhanced.

REFERENCES

[1] Vamsi Krishna Myalapalli and Sunitha Geloth, "High Performance JAVA Programming", IEEE International Conference on Pervasive Computing, Pune, India, January 2015.
[2] Vamsi Krishna Myalapalli and Sunitha Geloth, "Minimizing Impact on Java Virtual Machine via JAVA Code Optimization", IEEE International Conference on Energy System and Application, Pune, India, October 2015.
[3] Vamsi Krishna Myalapalli, Jaya Krishna Myalapalli and Pradeep Raj Savarapu, "High Performance C Programming", IEEE International Conference on Pervasive Computing, Pune, India, January 2015.
[4] Vamsi Krishna Myalapalli andMudduButchi Shiva, "An Appraisal to Optimize SQL Queries", IEEE International Conference on Pervasive Computing, Pune, India, January 2015.
[5] Vamsi Krishna Myalapalli and Pradeep Raj Savarapu, "High Performance SQL", 11th IEEE India International Conference on Emerging Trends in Innovation and Technology, Pune, India, December 2014.

BIOGRAPHY

Vamsi Krishna completed his Bachelor's in Computer Science from JNTUK University College of Engineering, Vizianagaram. He is Oracle Certified Professional JAVA SE7 Programmer,DBA 11g, SQL Expert and PL/SQL Developer, Microsoft Certified Professional (Specialist in HTML5 with JAVA Script & CSS3),. He has fervor towards SQL & Database Tuning.

Sunitha Geloth is currently working as Software Engineer at Open Text Corporation, Hyderabad. She completed her Bachelor's in Computer Science from PRRMCollege of Engineering, Hyderabad. She is Oracle Certified Professional JAVA SE7 Programmer. She has fervor towards JAVA Programming.