Whitepaper

Secure Programming in Java

Document History

Version-Number	Date	Editor	Reviewer	Status	Remarks
0.9	11/2005	EUROSEC		DRAFT	

created by

EUROSEC GmbH Chiffriertechnik & Sicherheit

Sodener Straße 82 A, D-61476 Kronberg Germany

www.eurosec.com mail: kontakt @ eurosec. com

Phone: +49 (0) 6173 60850

Table of Content

1 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. (In this paper, every kind of Java code running in some Java runtime environment, will be denoted as a Java application. In particular, applets are not distinguished from regular Java applications.) Indeed, under the often used slogan "Java Security", quite different aspects concerning the security of Java applications are addressed.

First of all, to set−up the foundation for secure programs, Java has been designed to be an inherent safe programming language. In particular Java does not allow to directly access or manage memory. Furthermore, Java is strongly typed and elementary data types are definitely defined in the Java Language Specification

[4.4 (1)]. Simply by its very basic language features, 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. Bytecode (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 [4.4 (2)], 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 bytecode. 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. Finally the JRE allows the 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".

While not being part of the innermost security features, "Java Security" also encompasses some standard libraries and extensions that are shipped with SUN's Java Development Kits (JDKs) and that are particularly aimed for the usage in security critical tasks. While the Java standard API does only define appropriate interfaces for most of these purposes , the JDKs also include implementations of cryptographic and security providers that might be used out of the box.

As outlined above, the Java programming language provides a lot of security features, build directly into the language and also supplied by security relevant APIs and implementations. Nevertheless, simply by choosing Java as the programming language for some program, will not guarantee that the program will be safe against secrecy attacks, integrity or availability attacks. Security concerns should be considered throughout the whole software development process, independent of the particular programming language chosen.

This paper discusses all the above mentioned aspects of "Java Security", but focuses on "secure programming" techniques that should be considered when programming in Java.

As "secure programming" is not possible without adhering to some general "good programming" practices, some of the guidelines are therefore of a more general nature.

2 Security Mechanisms of the JRE

Java security relies heavily on the execution model as specified in the Java language specification [4.4 (1)] and in the Java virtual machine specification

[4.4 (2)]. In particular it relies on bytecode verification, class loading mechanisms, security managers and access controllers. Before we are going to discuss secure programming guidelines in the following section, a short review of the execution model might be appropriate. Note: The Java security architecture has been improved considerably in JDK1.2 compared to older JDK versions. This paper focuses on this improved architecture.

2.1 The execution of a Java program − an overview

Chapter 12 of the Java language specification [4.4 (1)] specifies the activities that occur during the execution of a Java program. For execution of a Java program a Java virtual machine (JVM) has to be started with a class file that implements the main method of the program. At start−up the JVM holds no binary representation of the class containing the main method. Therefore the JVM invokes its class loader to try to load this class. After (successfully) loading the class it must be linked by the JVM. Linking comprises verification, preparation and (optionally) resolution. After the class has been linked, it will be initialized. As initialization of a class starts with the initialization of its super class, this step may involve (recursively) loading, linking and initializing further classes. Finally, after the class has been initialized, its main method is invoked. In case that unresolved references make their appearance in the execution thread(s), the corresponding classes have to be loaded, linked, and initialized, as well.

2.1.1 Loading

The method **defineClass** of the class **ClassLoader** is used to construct class objects from binary representations given in the class file format (bytecode). Well− behaved class loaders maintain the following properties [4.4 (1), 12.2]:

- Given the same name, a class loader should always return the same class object.
- If a class loader L1 delegates loading of a class C to another loader L2, then for any type T that occurs as the direct superclass or a direct superinterface of C, or as the type of a field in C, or as the type of a formal parameter of a method or constructor in C, or as a return type of a method in C, L1 and L2 should return the same class object.

2.1.2 Linking

The specification allows an implementation of the JVM flexibility as to when linking activities (and, because of recursion, loading) take place, e.g.

- lazy (late) resolution: resolves symbolic references in a class/interface, only when it is used,
- static resolution: all references are resolved at once while the class is being verified.

2.1.2.1 Verification

If the bytecode verifier detects a violation against the JVM specification, it throws an instance of VerifyError (subclass of LinkageError).

2.1.2.2 Preparation

During preparation the static fields (class variables and constants) for a class or interface are created and initialized to the default values. No source code has to be executed in this phase. (Additional data structures (such as a "method table") may be precomputed by the JVM (implementation dependent).)

2.1.2.3 Resolution

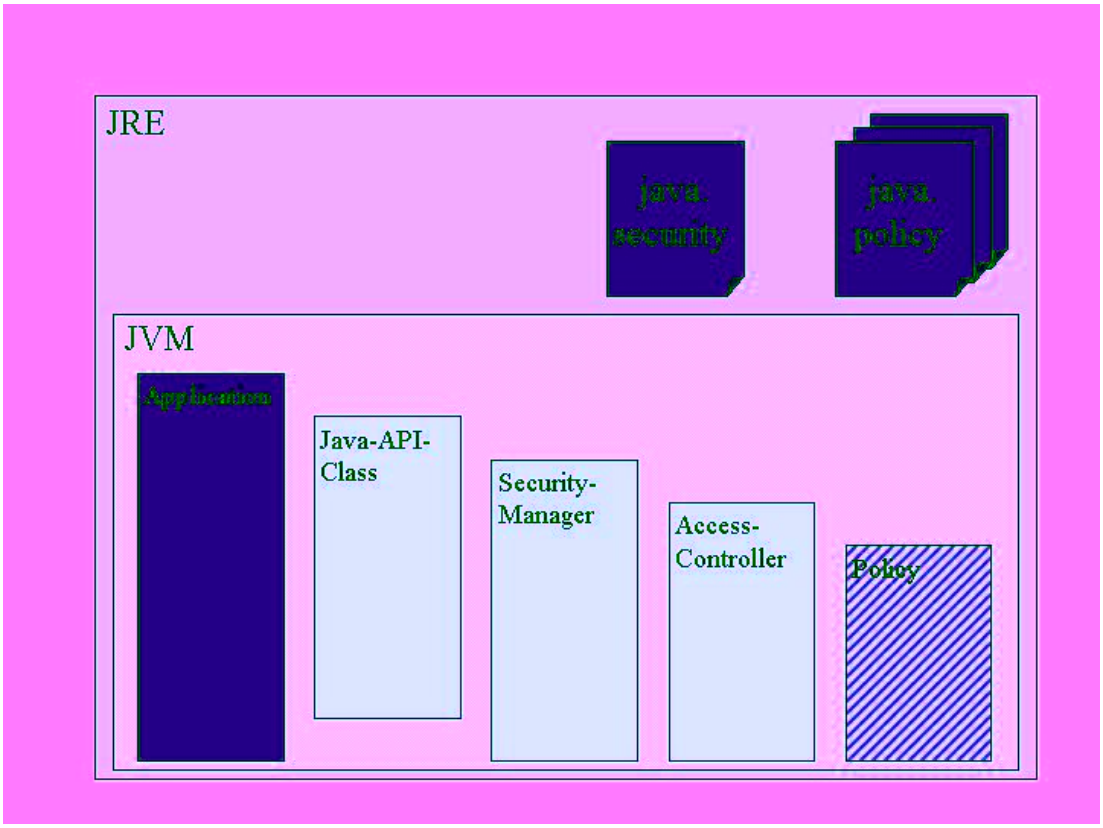
Symbolic references are checked to be correct and replaced with a direct reference that can be more efficiently processed.

2.1.3 Initialization

- Initialization of a class: First its superclass is initialized (if not already done). Then the static initializers and the initializers for the static fields are executed.
- Initialization of an interface: The initializers for the fields are executed. Warning: It is possible to construct examples where the value of a class variable can be observed when it still has its initial default value, i.e. before its initializing expression is evaluated (see [4.4 (1), 12.4.1]).

2.1.4 Security checks

At runtime all security critical Java API methods call the check methods of the actually loaded SecurityManager class. Remember that by default SUN's Java virtual machine does not initialize a SecurityManager object, i.e. an application is granted all permissions by default. To instantiate an object of the base SecurityManager class the option **−Djava.security.manager** has to be used. The base SecurityManager class delegates most checks to the AccessController class of the JVM. The behavior of the AccessController class is configured at startup from policy files.


JRE
java.
security
java.
policy
JVM
Application
Java-API-
Class
Security-
Manager
Access-
Controller
Policy

3 Secure Programming Guidelines

As mentioned in the introduction, secure programming is not possible without obeying some general good programming practices. Therefore this chapter is divided into two parts. The first part contains general rules that should be followed to write secure programs, while the second part concentrates on Java specific topics. Although the guidelines given in the first part are of a somewhat general nature and similar rules can be formulated for other programming languages as well, this paper concentrates on the specific implications for programming applications in Java.

3.1 General Guidelines

3.1.1 Coding style guidelines

- Establish and enforce company wide coding style guidelines.

Having a well established software development process provides the necessary framework to produce robust, stable and secure software. Part of such a development process should be the implementation (and enforcement) of a set of coding style guidelines. Only if all software developers adhere to the same best programming practices, the final code will be in a consistent and simple state. This is the basis for maintainable and robust code, that can be evaluated (reviewed) and trusted to function as specified. Of course, experienced software engineers should be involved, when best practices are established.

3.1.2 Input data validation

- Always validate input to public methods.

Java is equipped with a lot of security features, such as automatic memory management, no pointer arithmetic and type safety. For that reason, programs written in Java are for example resistant against buffer overflow or format string attacks. Nevertheless, input validation for public methods is most important to guarantee a well defined behavior of a program. Parameters whose values are out of range should lead to cleanly flagged error states, which allows an application to fail securely. Moreover, input parameters may influence resources (e.g. memory consumption) or may be forwarded to native methods, making input validation even more important.

- Provide utility methods for input data validation and transformation purposes.

To support a stable and uniform mechanism for input parameter checking (simplifying implementation and code auditing as well), appropriate methods for such purposes should be provided in a dedicated package. For example, a task that occurs frequently in input validation is the transformation of some string representing a number into some elementary data type (int, float, ...) or to just check if a given string represents a number in a given range (e.g. an integer with 16 digits). Providing some globally reusable methods for such purposes, avoids having multiple different implementations scattered throughout the code, some of them possibly having flaws.

- Consider to automatically generate input data validation methods.

If the functionality for a formally specified API (for example given in some XML format) has to be implemented, it should be considered to generate the code for input validating methods directly from the specification (parsing the parameter types and specified ranges for parameter values from the API specification and generating the code). Using such an automated code generation mechanism will contribute to a very high level of assurance, that the input validation methods are correct, complete and uniformly implemented. Furthermore, this approach may help to keep the code synchronized with the specification. (The code can be updated to reflect some small changes in the specification simply by recompilation.)

3.1.3 Performance optimization

- Optimize performance only after profiling.

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

3.1.4 Assertions and debug traces

- Add debug traces to your code.

During software development and testing phases 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.

- 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.

```
Public final class DebugConstants {
  public final boolean assertionEnabled = true;
  public final int traceLevel = 3;
}
public class Demo {
  void exampleMethod() {
        // Assertion code, that will be compiled only if
        // DebugConstants.assertionEnabled is true.
        if (DebugConstants.assertionEnabled) {
             // ....
        }
        // Debug trace code, that will be compiled only if
        // DebugConstants.traceLevel is at least 2.
        if (DebugConstants.traceLevel >= 2 ) {
             System.err.out("Debug trace output for
traceLevel 2 orhigher");}}}
```

Using global static variables, as shown in the example allows the 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.

3.1.5 Mutable objects

- Make objects immutable whenever possible.

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

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

A necessary condition for programming secure software is the implementation of clear interfaces and strict data encapsulation. The possibility to change member variables of an object from the outside does represent a severe violation of this basic rule. Hence, if references to mutable objects are given as parameters to a public constructor or method, the constructor/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.

- Never return references to mutable member objects from public methods.

Again, the possibility to change member variables of an object from the outside does represent 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 to be constructed first, to finally return a reference to this clone.

3.1.6 Static variables

- Do not use non−final public static variables.

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

- 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 class uses a static final variable that has been changed, this class must be recompiled in order to reflect the changes in the bytecode.

3.1.7 Inheritance

- 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).

- Program by contract using the template method design pattern.

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

- 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 your class with subclasses that override the called method.

3.1.8 Access of variables and methods

- 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.

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

Making variables private and limit access only via accessor methods allows to differentiate 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.

3.2 Java specific Guidelines

3.2.1 Garbage Collector

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

- Flag (large) objects to be ready for garbage collection as soon as possible, by setting all references to such objects 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 you are concerned about memory consumption, you 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 usually not such a good idea 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 (since JDK 1.2) can be employed.

3.2.2 Exceptions

- Never write try/catch blocks with an empty catch block.

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

- 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 resources are released if an exception occurs. This can be done in catch or finally blocks. If you use the catch blocks to release claimed resources, you have to take care to release the resources in the try block as well. Using finally blocks for such purposes unifies and simplifies the release of resources.

3.2.3 Serialization and Deserialization

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

- Do not directly serialize variables that contain sensitive data.

While being serialized, an object is outside of the control of the JVM and might be vulnerable to modification or inspection. The serialization of variables can be prohibited by declaring 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.

- 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 deserialized object is in a consistent and valid state.

- Use the transient keyword for fields that contain direct handles to system resources (such as 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 is good programming practice to avoid serialization of such handles.

- When writing your own serialization methods, take care not to use DataOutput or DataInput methods that may expose byte arrays to malicious code.

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

3.2.4 Java Native Interface (JNI)

- 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 your 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 executed very carefully.

3.3 Security critical modules and tasks

3.3.1 Separation of security critical modules

- 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 of the software development process is mandatory in order to define clean interfaces to such functionalities. Furthermore, having identified such tasks/modules, implementation and review of such modules can be delegated to software engineers with appropriate skills.

- Regularly review all code to check whether security critical methods are only implemented in distinguished packages/classes.

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

3.3.2 Cryptographic and security relevant packages

- Use well established or evaluated packages for security critical algorithms and services.

In case that cryptographic or other security relevant algorithms or services have to be integrated in an application, this should be done by integrating appropriate packages serving the needed functionality. The JDK itself ships with quite some security specific packages such as JCA, JCE, JAAS, JSSE and SUN's provider classes (or they are obtainable as extensions, depending on the JDK's version). While the packages shipped with the JDK do provide some sound interfaces and provide sufficient functionalities for many applications through the included provider packages, it should be noted, that the provider classes are not open source and are not certified in any way. Hence you have to finally assess, if the usage of SUN's provider classes complies with your security requirements. Furthermore, the cryptographic mechanisms implemented may not satisfy all your needs. Currently there are no implementations for the AES and for ECC (elliptic curve cryptography) included. To get an overview of the providers that are installed in a Java runtime environment the ProviderCheck application [4.2 (4)] may be used. The output of this tool lists all installed providers and their implemented functionalities. The result for the Java runtime version 1.4.0 is reproduced in an Appendix.

While writing your own provider is possible, it should be evaluate whether using third party software packages might be an alternative. Using third party software that has been certified or is open source and has been reviewed thoroughly, might increase the level of assurance that the packages satisfy the security requirements considerably.

- Do not implement some proprietary cryptographic or security relevant routines.

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

3.3.3 Sensitive data

- Do not store some secrets in your 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 your code, but it does not help to hide secret data.

- 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 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.)

- 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 you 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.)

3.3.4 Pseudo random data

- Never use java.util.Random for the generation of random numbers in security relevant contexts.

Random numbers are needed in many security relevant contexts, e.g. for the generation of initial passwords, PINs, TANs or SessionIDs. It is not at all safe to use java.util.Random for the generation of random numbers in such security critical applications, as the sequence of numbers returned by java.util.Random is predictable, once any of its values has been observed. In a security relevant application a cryptographic secure pseudo random number generator has to be used, and it has to be guaranteed that this pseudo random number generator is initially seeded with an appropriate amount of real entropy. The class java.security.SecureRandom is included in the Java API to provide access to pseudo random number generator implementations. But note that the implementation of a SecureRandom object that is constructed with the standard constructor depends on the configuration of the environment (security providers configured in the configuration file java.security). To enforce the usage of a particular SecureRandom implementation you have to take the Java runtime environment into consideration or to supply your own SecureRandom class. The default implementation that comes with SUN's JDK implements a proprietary seeding mechanism. Therefore, if you need a certified mechanism to seed your pseudo random number generator, you may have to look for another source of initial real random data. If you have access to some source of real randomness, it is always a good idea to add such data to the seed (using the setSeed() method).

3.4 Java specific security mechanisms

3.4.1 Bytecode verification

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

As bytecode verification is an integral part of the enforcement of the security of a Java application, it is important do know what classes are checked, when running a Java application. To speed up performance, virtual machine implementation do not check "trusted" classes. Unfortunately it is not specified what classes are "trusted" classes. For example in JVM implementations shipped with JDK 1.1 all classes on the CLASSPATH are trusted, whereas JDK 1.2 (and higher) implementations only takes core Java platform classes as trusted.

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

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 not always performed by a JVM. To enforce such checks for virtual machines shipped with JDKs the −verify option has to be used.

3.4.2 Policies and security managers

- Supply policy files or implement a SecurityManager class, such that your 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.

3.4.3 Privileged code

- Keep privileged code blocks as short as possible.
- 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 to circumvent 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 and obey the rules mentioned above.

3.4.4 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. As the option "package.definition" is not supported by SUN's default class loaders, and as the usage of these options makes the protection dependent on the configuration of the JRE it is recommended to ship sealed JAR files instead.

4 Resources

4.1 Research

4.1.1 SUN

- Java Security Homepage (http://java.sun.com/security/)
- **Security Code Guidelines**, Sun Microsystems, Inc.(http://java.sun.com/security/seccodeguide.html)
- Frank Yellin. **Low Level Security in Java**. *Proceedings of the 4th International World Wide Web Conference*, Boston, Mass., December 1995. (http://www.w3.org/Conferences/WWW4/Papers/197/40.html)

4.1.2 IBM

- IBM Research Java Security (http://www.research.ibm.com/javasec/)

- L. Koved, A. Nadalin, D. Neal and T. Lawson, **The Evolution of Java Security**, IBM Systems Journal (Vol 37,No 3). (http://www.research.ibm.com/journals/373/koved.html)

- S. Porat, M. Biberstein, L. Koved, B. Mendelson, **Automatic Detection of Mutable Fields in Java**, CASCON 2000. Copyright © 2000 IBM. (http://www.research.ibm.com/people/k/koved/papers/MutabilityCASCON2000.pdf)

- L. Koved, M. Pistoia and A. Kershenbaum, **Access Rights Analysis for Java**, Copyright © 2000 IBM. (http://www.research.ibm.com/javasec/OOPSLA2002preprint.pdf)

4.1.3 Princeton University − Secure Internet Programming project

- Homepage (http://www.cs.princeton.edu/)

- Lujo Bauer, Andrew W. Appel, and Edward W. Felten, **Mechanisms for Secure Modular Programming in Java**, Technical report CS−TR−603−99, Department of Computer Science, Princeton University, July 1999. (ftp://ftp.cs.princeton.edu/techreports/1999/603.ps.gz) [Interface of a module should be separate from its implementation, but Java packages do not provide explicit module interfaces. The paper introduces explicit **export interfaces** and **membership lists** that are hierarchical scalable to allow the definition of multiple interfaces with a convenient name−space management scheme. A prototype has been implemented as a source−to−source and bytecode−to−bytecode transformation wrapped around a standard Java compiler (no source, no bin).]

- Andrew W. Appel, **Protection against untrusted code**, IBM Developer Works, September 1999. (http://www−106.ibm.com/developerworks/library/untrusted−code/) [JIT are very complex (more than 100.000 lines of code). Therefore there you cannot be sure whether untrusted code might misuse some bug to circumvent the type−checking mechanism, that is whether the native code output from the JIT is still save. Extending the JIT to provide deliver proof− carrying code, the native code could be checked by a proof−checker (small trusted program (1000 lines)) to being safe.]

4.2 Tools

(1) Junit (http://www.junit.org/index.htm) [Junit is Open Source Software, released under the IBM's Common Public License Version 0.5 and hosted on SourceForge.]

(2) Jtest (http://www.parasoft.com/j) [Jtest automates unit testing and coding standard enforcement.]

(3) ESC/Java (http://www.research.compaq.com/SRC/esc/Esc.html) [The Compaq Extended Static Checker for Java (ESC/Java) detects, at compile time, common programming errors that ordinarily are not detected until run time, and sometimes not even then; for example, null dereference errors, array bounds errors, type cast errors, and race conditions.]

(4) ProviderCheck (EUROSEC)

4.3 Books

4.3.1 Java Security

- Jess Garms, Daniel Somerfield, *Professional Java Security*, Wrox Press Inc, 2001.
- Li Gong, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation,* Addison−Wesley,Reading, Mass., 1999.
- Gary McGraw, Edward W. Felten, *Securing Java: Getting Down to Business with Mobile Code,* John Wiley & Sons, 1999.
- Gary McGraw, Edward W. Felten, *Java Security: Hostile Applets, Holes, and Antidots,* John Wiley & Sons, New York, 1996.
- Jamie Jaworski, Paul J. Perrone, *Java Security Handbook*, Sams, 2000
- Scott Oaks, *Java Security*, O'Reilly & Associates, Sebastopol, CA, 1998.

4.3.2 Java virtual machine

- Jon Meyer, Troy Downing, *Java Virtual Machine*, O'Reilly 1997.

4.3.3 General Java programming guidelines

- Allan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trvor Misfeldt, Jim Shur, Patrick Thompson, *The Elements of Java(TM) Style*, Cambridge University Press, 2000.
- Michael C. Daconta (Editor), Eric Monk, J. Paul Keller, Keith Bohnenberger, *Java Pitfalls: Time−Saving Solutions and Workarounds to Improve Programs*, John Wiley & Sons, 2000.
- Peter Haggar, *Practical Java(TM) Programming Language Guide*, Addison− Wesley, 2000.

4.3.4 General secure programming guidelines

- John Viega, Gary McGraw, *Building secure software: How to avoid security problems the right way,* Addison−Wesley, 2002. (http://www.buildingsecuresoftware.com)
- David A. Wheeler, *Secure Programming for Linux and Unix HOWTO,* (http://www.dwheeler.com/secure-programs/)

- Michael Howard, David LeBlanc, *Writing secure code,* Microsoft Press, 2002.

- Lawrence C. Paulson, *Software Engineering II*, Lawrence C Paulson Computer Laboratory University of Cambridge, Lecture Notes, 1999. (http://www.cl.cam.ac.uk/Teaching/2001/SWEng2/notes.pdf)

- Steve Bellovin (smb@research.att.com), *Shifting the Odds, Writing (More) Secure Software*, 1996, 36 Seiten

4.4 Java specifications

(1) James Gosling, Bill Joy, Guy Steel, Gilad Bracha, Java Language Specification, Second Edition, Sun Microsystems, 2000 (http://java.sun.com/langspec−2.0/j.title.doc.html)

(2) Tim Lindholm, Frank Yellin, The Java Virtual Machine Specification, Second Edition, Sun Microsystems, 1999 (http://java.sun.com/vm/vmspec/VMSpecTOC.doc.html)

(3) JavaBeans(TM) API Specification, ed. Graham Hamilton, Sun Microsystems, Inc. (1997) (http://www.javasoft.com/beans/docs/beans.101.pdf)

(4) Javadoc home page, Sun Microsystems(http://java.sun.com/j2se/javadoc/index.html)

(5) Java(TM) Code Conventions, Sun Microsystems, Inc. (20 April 1999) (ftp://ftp.javasoft.com/docs/codeconv/CodeConventions.pdf)

(6) Sheng Liang, The Java Native Interface: Programmer's Guide and Specification, Addison−Wesley, 1999

5 Appendix − JRE 1.4.0 Security Providers

Security providers and their implemented classes of the JRE 1.4.0. (Output from the ProviderCheck application [4.2 (4)].)

```
  Provider 1:


  Provider: SUNVersion: 1.2Info: SUN (DSA key/parameter generation; DSA signing;
  SHA-1, MD5 digests; SecureRandom; X.509 certificates; JKS keystore; PKIX
  CertPathValidator; PKIX CertPathBuilder; LDAP, Collection CertStores)

SecureRandom:SHA1PRNG
  sun.security.provider.SecureRandomSHA1PRNG ImplementedIn
  Software

MessageDigest:MD5 sun.security.provider.MD5MD5
  ImplementedIn SoftwareSHA
  sun.security.provider.SHA
  SHA-
  1SHA1SHA
  Implemen
  tedIn
  Software

AlgorithmParameters:DSA
  sun.security.provider.DSAParameters1.2.8
  40.10040.4.1
    1.3.14.3.2.12
  DSA ImplementedIn Software


AlgorithmParameterGenerator:DSA
  sun.security.provider.DSAParameterGeneratorDSA ImplementedIn
  SoftwareDSA KeySize 1024

KeyFactory:DSA
  sun.security.provider.DSAKeyFactory
  1.2.840.10040.4.1
    1.3.14.3.2.12
  DSA ImplementedIn Software


KeyPairGenerator:DSA
  sun.security.provider.DSAKeyPairGenerator
  1.2.840.10040.4.1
  1.3.14.3.2.12

    OID.1.2.840.10040.4.1DSA ImplementedIn SoftwareDSA KeySize 1024

Signature:SHA1withDSA
  sun.security.provider.DSA1.2.
  840.10040.4.3
  1.3.14.3.2.13

  1.3.14.3.2.27
  DSADSAWithSHA1DSSOID.1.2.840.10040.4.3SHA-1/DSASHA/DSASHA1/DSASHAwithDSA
  SHA1withDSA ImplementedIn SoftwareSHA1withDSA KeySize 1024

CertificateFactory:X509 sun.security.provider.X509Factory
    X.509
  X509 ImplementedIn Software


KeyStore:JKS sun.security.provider.JavaKeyStoreJKS
  ImplementedIn Software

CertPathBuilder:PKIX sun.security.provider.certpath.SunCertPathBuilderPKIX
  ImplementedIn SoftwarePKIX ValidationAlgorithm draft-ietf-pkix-new-part1-
  08.txt
```

```
CertPathValidator:PKIX sun.security.provider.certpath.PKIXCertPathValidatorPKIX
  ImplementedIn SoftwarePKIX ValidationAlgorithm draft-ietf-pkix-new-part1-
  08.txt

CertStore:Collection
  sun.security.provider.certpath.CollectionCertStoreCollection ImplementedIn
  SoftwareLDAP sun.security.provider.certpath.LDAPCertStoreLDAP ImplementedIn
  SoftwareLDAP LDAPSchema RFC2587


  Provider 2:


  Provider: SunJSSEVersion: 1.4Info: Sun JSSE provider(implements RSA Signatures,
  PKCS12, SunX509 key/trust
   factories, SSLv3, TLSv1)


KeyFactory:RSA com.sun.net.ssl.internal.ssl.JSA_RSAKeyFactory

KeyPairGenerator:RSA com.sun.net.ssl.internal.ssl.JSA_RSAKeyPairGenerator

Signature:MD2withRSA
  com.sun.net.ssl.internal.ssl.JSA_MD2RSASignatureMD5withRSA
  com.sun.net.ssl.internal.ssl.JSA_MD5RSASignatureSHA1withRSA
  com.sun.net.ssl.internal.ssl.JSA_SHA1RSASignature

KeyStore:PKCS12 com.sun.net.ssl.internal.ssl.PKCS12KeyStore

KeyManagerFactory:SunX509 com.sun.net.ssl.internal.ssl.KeyManagerFactoryImpl

SSLContext:SSL com.sun.net.ssl.internal.ssl.SSLContextImplSSLv3
  com.sun.net.ssl.internal.ssl.SSLContextImplTLS
  com.sun.net.ssl.internal.ssl.SSLContextImplTLSv1
  com.sun.net.ssl.internal.ssl.SSLContextImpl

TrustManagerFactory:SunX509 com.sun.net.ssl.internal.ssl.TrustManagerFactoryImpl


  Provider 3:


  Provider: SunRsaSignVersion: 1.0Info: SUN's
  provider for RSA signatures

KeyFactory:RSA com.sun.rsajca.JSA_RSAKeyFactory

KeyPairGenerator:RSA com.sun.rsajca.JSA_RSAKeyPairGenerator

Signature:MD2withRSA
  com.sun.rsajca.JSA_MD2RSASignatureMD5withRSA
  com.sun.rsajca.JSA_MD5RSASignatureSHA1withRSA
  com.sun.rsajca.JSA_SHA1RSASignature
  Provider 4:


  Provider: SunJCEVersion: 1.4Info: SunJCE Provider (implements DES, Triple DES,
  Blowfish, PBE, Diffie-Hellman, HMAC-MD5, HMAC-SHA1)

AlgorithmParameters:Blowfish
  com.sun.crypto.provider.BlowfishParametersDES
  com.sun.crypto.provider.DESParametersDESede
  com.sun.crypto.provider.DESedeParameters
    TripleDESDiffieHellman
  com.sun.crypto.provider.DHParametersD
  HPBE
  com.sun.crypto.provider.PBEParameters
  PBEWithMD5AndDES

AlgorithmParameterGenerator:DiffieHellman
  com.sun.crypto.provider.DHParameterGeneratorDH

KeyFactory:DiffieHellman
  com.sun.crypto.provider.DHKeyFactoryDH

KeyPairGenerator:DiffieHellman
  com.sun.crypto.provider.DHKeyPairGeneratorDH
```

```
KeyStore:JCEKS com.sun.crypto.provider.JceKeyStore

Cipher:Blowfish com.sun.crypto.provider.BlowfishCipherDES
  com.sun.crypto.provider.DESCipherDESede
  com.sun.crypto.provider.DESedeCipher
    TripleDESPBEWithMD5AndDES
  com.sun.crypto.provider.PBEWithMD5AndDESCipherPBEWithMD5AndTripleDES
  com.sun.crypto.provider.PBEWithMD5AndTripleDESCipher

KeyGenerator:Blowfish com.sun.crypto.provider.BlowfishKeyGeneratorDES
  com.sun.crypto.provider.DESKeyGeneratorDESede
  com.sun.crypto.provider.DESedeKeyGenerator
    TripleDESHmacMD5 com.sun.crypto.provider.HmacMD5KeyGeneratorHmacSHA1
  com.sun.crypto.provider.HmacSHA1KeyGenerator

KeyAgreement:DiffieHellman
  com.sun.crypto.provider.DHKeyAgreementDH

SecretKeyFactory:DES com.sun.crypto.provider.DESKeyFactoryDESede
  com.sun.crypto.provider.DESedeKeyFactory
    TripleDESPBEWithMD5AndDES com.sun.crypto.provider.PBEKeyFactory

Mac:HmacMD5 com.sun.crypto.provider.HmacMD5HmacSHA1
  com.sun.crypto.provider.HmacSHA1


  Provider 5:


  Provider: SunJGSSVersion:
  1.0Info: Sun (Kerberos v5)

GssApiMechanism:1.2.840.113554.1.2.2
  sun.security.jgss.krb5.Krb5MechFactory
```