



Name

Java Security Anti-Patterns and the Top-Ten Guidelines to Avoid Them

Authors

Marco Pistoia, Ph.D., and David Safford, Ph.D.
IBM Thomas J. Watson Research Center
Hawthorne, NY 10532, USA

{pistoia,safford}@us.ibm.com

Abstract

The Java programming platform has a very sophisticated security architecture embedded into the language itself. This includes fine-grained access control, flexible and customizable user authentication and authorization, and cryptographic libraries that can be used to guarantee integrity, confidentiality, and non-repudiation. Unfortunately, in today's software lifecycle, security is often considered only as an afterthought, and most of the times it is either not used at all, or used in the wrong way. In addition, programmers who have transitioned from other languages such as C, often are not trained in Java's security features, and frequently fail to use these features properly. Analysts agree that security should be considered an integral part of any IT solution from the very beginning. A security hole can compromise the reputation of a product or even an entire company much more than any other bug can do, resulting in very severe market losses. In spite of the risks of expensive security flaws and embarrassing headlines, the lesson does not seem to have been learned yet.

The purpose of this paper is to describe frequently observed Java security anti-patterns, and provide the top-ten security guidelines that software architects, engineers, programmers, and system administrators should always follow and respect to guarantee that a program is secure throughout the entire software lifecycle. Besides describing how to design an application securely, the top-ten guidelines presented in this paper are:

1. Access control do's and don'ts
2. Privileged code guidelines
3. How to avoid tainted variables
4. How to make access scope decisions
5. How to use cryptographic services
6. Protecting against package access and package insertion
7. Inner classes – something to avoid
8. Class loaders do's and don'ts
9. Security guidelines for native methods
10. What not to do when using multithreading

This paper explains what to do and what not to do, based on practical experience gained through security audits of strategic IBM and customer applications.

Type

Define Architectural Strategy, which includes



- Solution Architecture

Solution Definition

- Business Processes and Business Objects
- Existing Services
- Solution Usage

Build, Test and Deploy, which includes

- Software Implementation
- Build Configuration
- Application Enablement
- Solution Tests
- Deployment Configuration

Context

The anti-patterns described in this paper arise during both the development and deployment phases of software lifecycle and are strictly related with secure coding. Writing secure code is hard and even the most experienced programmer can make mistakes.

Problems

Modern access control systems, such as Java 2 and Microsoft CLR, are based on *stack inspection*. What this simply means is that when a restricted resource is being accessed, all the callers on the stack must exhibit the permission to access that resource. Permissions are granted to code and users in a policy database. By default, the policy database is implemented as a flat file called *policy file*. Access control in Java is fine-grained. This means that code and users can be assigned permissions to access restricted resources. Permissions are assigned to code based on the URL from which the code is coming and the digital certificates of the code signers. Permissions are assigned to users based on their authenticated principals.

Library code is often programmed to access a restricted resource (for example, a configuration file) on its own, without an explicit client request. Since the client will be on the stack when that resource is accessed, the client will need to exhibit the permission as well. This could result in a security violation. Fortunately, access-control systems based on stack inspection allow declaring a portion of library code as privileged. When a library's privileged code is encountered on the stack, the access controller does not check the permissions of the library's clients. For large programs, however, it is often almost impossible to correctly identify what portions of library code should be made privileged by simply using code inspection or testing. Another risk is introducing "tainted variables" in privileged code. A variable is *tainted* if its value can be influenced by client code. A tainted variable is not necessarily a security exposure. It may become a security problem if it is used inside a library's privileged code because, in that case, the client's permissions are not checked.

Leaving authorization aside, another problem that should be considered is that often code is written without security in mind. Code is written and componentized in a way that can make it very difficult to enforce restrictions on the security-sensitive information managed by that code. Developers should be trained on:

- How to make access scope decisions—and use the private, protected, public, static, and final modifiers in the proper way.
- Protecting against package access and package insertions—so that untrusted code cannot access a security-sensitive package or declare classes part of that package.



- Avoiding inner classes—as useful as they appear to be, they constitute a security threat because the compiler gives default scope to the private fields of the outer classes, without even warning the developer!

Multithreading can dramatically improve efficiency. However, it can create race conditions especially if methods have access to shared data. Races can be exploited to create security attacks, such as:

- Data corruption
- Unplanned side effects
- Denial of service via locking shared resources

Similarly, deadlocks can cause an entire application to stop working. Compilers, deployment tools, and run-time environments do not check for improper use of multithreading from within Java applications. This means that the trust is transferred to the developer. Even when allowed, multithreading can cause unwanted synchronization and race conditions.

Native methods have several advantages but they can also be the source of severe security problems. Specifically, a native method bypasses all the security controls performed by the Java Runtime Environment (JRE), including authorization checks, class verification, isolation, and enforcement of access-control modifiers. Frequently observed problems include using native methods when it is not necessary, and writing native methods without addressing security issues specific to the native methods. Uses native methods that are based on native scripting languages are particularly difficult to secure, as the scripting languages interpret the requests in ways that are difficult to predict.

Forces

The problem with Java authorization is that it is often difficult, if not impossible, to manually identify all the permissions needed to execute a program without `SecurityExceptions`. In addition, in many cases, Java programmers who do not understand the signing features available in Java do not sign their code at all. Not enforcing access control at all is the easiest solution. It is a seductive solution because it allows shipping a program without worrying in advance about detecting its security requirements. System administrators are attracted to this easy solution too because with no access control a program performs faster and better, with no risk of run-time authorization failures.

From an accessibility point of view, several problems can arise in which client code running on top of a library can gain access to the library's security-sensitive fields and methods. For example, client code could (maliciously or inadvertently) change the states of classes in the underlying library, break component isolation, or gain access to private information (such as private keys or passwords).

Many enterprise programs make heavy use of multithreading. However, understanding whether a multithreaded program can generate race conditions, deadlock, or unwanted synchronization can be very complicated, especially for large and complex programs. Oftentimes, programs that perform multithreading are not well implemented and serious problems may be discovered only when programs are deployed.

Since native methods are executed directly on top of the operating system, developers tend to use them every time there are performance problems, or a particular function cannot be implemented in pure Java.

Common Solution

When it comes to Java authorization, people tend to grant too many permissions or to disable security because access-control configuration is very complex. This can result in violation of



the so-called “Principle of the Least Privilege.” As a result, code that should be considered untrusted gains permissions to perform operations that it should not be allowed to perform.

Another solution that is typically adopted is to identify the permission requirements by running the program with a SecurityManager enabled, but without granting the program any permission. This will generate a SecurityException every time the program attempts to access a restricted resource. By inspecting the stack trace, a developer or system administrator can detect what code needs to be granted a permission, and what permission is actually needed to eliminate that particular SecurityException. This process must be repeated many times until no SecurityException is thrown. However, if the test cases do not cover all the possible paths of execution, some SecurityExceptions may remain undiscovered until run time, making the application unstable. A variant of this solution can be adopted also to identify which portions of code should be made privileged. Basically, the solution consists of logging all the SecurityExceptions obtained by running a library with a test client that has not been granted any permission. For each SecurityException, it is then necessary to distinguish whether that Exception was generated by the client’s not having the permission or by the library code’s not being privileged.

From an accessibility perspective, the easiest solution that many developers adopt is to declare public any class and any method and field inside it. Another quick, but unsafe, solution is to reference fields directly instead of using getters and setters that check for the validity of the operation. These quick solutions certainly allow a development team to ship an apparently well-working product in a shorter time, but the risk remains that the product is very unsafe.

To avoid the problems caused by multithreading, developers and system administrators perform testing. However, lack of a sufficiently large test-case suite can cause multithreading-related problem to go undiscovered until run time.

Using native methods has several advantages:

- They are executed directly on the operating system
- They can do things Java code may not be able to do
- They may be more efficient than Java

However, native methods should be used with maximum care and only when strictly necessary. It is always recommended for native methods to be checked by a team of people to really assess whether they are required and they were written without compromising the integrity of the program the contains them or the system that runs them.

Resulting Context

If security is not enforced at all, or is enforced in the wrong way, any untrusted program can potentially gain control of the platform on which it is running. This includes also not using privileged code at all, or using it in the wrong way. In such cases, client code can misuse the permissions granted to it and compromise the security of an entire system.

The task of manually identifying the permission and privileged-code requirements by looking at the SecurityExceptions is tedious, time consuming, and error prone. Additionally, if a run-time execution path remains undiscovered during testing, it will cause a run-time authorization failure that can potentially compromise the stability of a program.

An incorrect use of access modifiers in object-oriented code exposes the state of a program’s objects to the external world. Untrusted programs could modify the state of internal objects, break component isolation, and get access to security-sensitive data.



The security vulnerabilities caused by an incorrect use of multithreading can involve data corruption, unplanned side effects, and denial of service attacks via locking of shared resources.

An inconsiderate use of native methods can bring several disadvantages. Specifically, native methods:

- Are non-portable
- Bypass all the Java run-time checks
- Effectively run with AllPermission
- Can violate type safety
- Can open a program or platform up to security attacks not otherwise possible in Java
- Are almost impossible to check for security

Preferred Solution

The solution we propose is to adopt static analysis to identify the permission and privileged-code requirements of a program or library. This step should be executed by both developers and system administrators. A static analysis solution for Java authorization is available as part of IBM's Security Workbench Development Environment for Java (SWORD4J), a new static analysis tool currently available on IBM alphaWorks.

When it comes to writing secure code that makes proper use of access modifiers, static analysis can offer invaluable help. Some suggestions to keep in mind are the following:

- Refrain from using non-final, public, static variables
 - There is no way to check whether the code that changes such variables has the appropriate permissions
 - Any mutable static state can cause unintended interactions between supposedly independent subsystems
- Always reduce scope for classes and class members as much as possible
- Refrain from using public variables
 - Let the interface to variables be through accessor methods—this way it is possible to add centralized security checks
 - Any public method that reads and/or writes any sensitive internal state should include a security check
- Protect against package access and insertion by configuring the security of the Java system accordingly and by sealing packages
- Take advantage of class-loading-enforced class isolation to prevent untrusted classes from accessing members in more trusted classes
 - This requires enforcing access-control restrictions on which code and users can create a ClassLoader and operate with it



- The permission to create a ClassLoader is implicitly equivalent to AllPermission!
- Wrap security-sensitive classes and return clones of security-sensitive fields
- The java.util collection classes can be subclassed
 - After a collection is constructed, it should be *frozen*, meaning that the following conditions should be made true:
 - No more inserts or deletes
 - Access only to fields inside contained instances, not to the instances themselves
 - No mutating methods for contained instances
 - Keep the rest of the API unchanged
- Never return a reference to an internal array that contains sensitive data
 - Even if the array contains immutable objects, it is better to return a copy so the user cannot change which objects are in the array
 - Instead of returning the internal array, it is better to make a copy of it, and return the copy
- Never store a user-given array of objects directly
 - Constructors and methods taking arrays of objects should clone the arrays before saving them internally
 - If, instead, the array is assigned directly to an internal variable, any changes made by the user to the external array could change the state of the internal object holding a reference to the array, even if the internal object was supposed to be immutable
- Never use an inner class if the fields of the outer class it uses are private—the compiler will give those private fields default scope and will not even warn you!
- Use static and dynamic analysis to identify security vulnerabilities due to incorrect access-scope decisions
- Be especially careful when access-scope decisions can affect encryption:
 - Encryption should be based on a random key
 - Anyone who knows the key and the algorithm may attack the system by decrypting and/or forging messages
 - Keep the key inaccessible
 - If the key is generated by a random number generator, keep the random number generator (algorithm, seed) inaccessible

The solution we propose is to adopt static analysis to identify whether multithreading has been used correctly. A combination of static and dynamic analysis can lead to identifying all the code locations that can generate unwanted synchronization, deadlocks, and race conditions.



If native methods are really needed (for example, to guarantee access to resources not otherwise obtainable in pure Java) the following patterns should be applied:

- Make sure that the native methods are really needed.
- If possible, avoid the use of native scripting languages
- Restrict their use to very simple functions
- Limit their accessibility
 - Make them private
 - Put them in sealed or restricted packages
- Test them thoroughly
 - What they take as parameters
 - What they return
 - Whether they bypass security checks
 - Whether they are public, protected, default-scope, or private

Design Rational

A sound static analyzer explores all the paths of execution and does not require building test cases. Static analysis can be integrated with dynamic analysis (testing) for more precise results. A sound static analyzer can also be very helpful in identifying tainted variable in an automated way, without having to rely on test cases.

Even excellent static analyzers for Java, however, can be *unsound*. This means that they may not find all the problems that can occur at run time. For example, a static analyzer for Java typically does not analyze native methods or reflective calls. Therefore, the information it produces may not reveal all the possible run-time problems. The main recommendation to break the secure coding anti-pattern is always to keep security in mind from the very beginning of software lifecycle and not just when a serious flaw is reported. It might be too late then.

Examples

One of the reasons why Java 2 security is often not adopted is that it is very complicated to configure. In fact, it is necessary to understand what permissions are needed by each program. In spite of the power of the Java 2 access control model, several organizations run Java software without activating the underlying security mechanism. A common recommendation is to repeatedly test a program without giving it any permission, logging all the SecurityExceptions it throws, and finally grant the program the permissions it requires, as long as the program is trusted enough to be granted those permissions. However, this work must be repeated possibly many times and is not guaranteed to actually identify all the permission requirements.

Some enterprise code recently analyzed revealed that developers wanted to do the right thing and used privileged code to prevent untrusted clients from requiring permissions. However, writing privileged code can be tedious. The code performing the security-sensitive operation must be wrapped in the run() method of a class implementing the PrivilegedExceptionAction or PrivilegedAction interface, and an object of that class must be passed as a parameter to AccessController.doPrivileged(). To simplify their task, those developers wrote a class that implemented a number of helper methods with signatures, for example:



- `public static String getProperty(String propertyKey)`
- `public static FileOutputStream getOutputStream(String fileNameToWrite)`
- `public static FileInputStream getInputStream(String fileNameToRead)`

The implementation of each one of those methods constructed an object of type `PrivilegedAction` or `PrivilegedExceptionAction` with a `run()` method performing the security sensitive operation, such as getting a system property or creating a `FileOutputStream` or a `FileInputStream`. That object was passed to `doPrivileged()`. The problem was that those methods, being public, could be invoked by any client code. Therefore, untrusted client code could get *any* system property and gain write and read access to *any* file of the file system without a security check, since the call to `doPrivileged()` would prevent the stack inspection from reaching the client. Using `SWORD4J` revealed that the variables `propertyKey`, `fileNameToWrite`, and `fileNameToRead` were all tainted.

Other examples of bad code practice were two enterprise programs, both delivered to client systems of an enterprise. The first program did not use code signing. Therefore, client users were supposed to dynamically install code without having a chance to verify that the system from which the code was coming was indeed the system of their enterprise. A further investigation revealed that Java was simply used as a means to deliver native executables to client machines. Another program that was audited for security purposes contained both Java code and native code. Unfortunately, native code and Java code each assumed that the other was performing sanity checks on the user input, and no sanity check was actually performed.

Related Patterns and Anti-Patterns

1. Management Anti-Pattern
2. Development Anti-Pattern

References

1. Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access Rights Analysis for Java. In Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOSPLA), pages 359-372, Seattle, WA, USA, November 2002. ACM Press.
2. Marco Pistoia, Robert J. Flynn, Larry Koved, and Vugranam C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP), pages 362-386, Glasgow, Scotland, UK, July 2005. Springer-Verlag.
3. Marco Pistoia, Duane Reller, Deepak Gupta, Milind Nagnur, and Ashok K. Ramani. Java 2 Network Security. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.
4. Marco Pistoia, Nataraj Nagaratnam, Larry Koved, and Anthony Nadalin. Enterprise Java Security. Addison-Wesley, Reading, MA, USA, February 2004.