Anti-Patterns in JDK Security and Refactorings

Marc Schönefeld

schonef@acm.org

Abstract: This paper underlines the importance of security awareness whilst programming Java applications. Several problems in current JDK implementations are demonstrated that allow to undermine the security of Java applications. Coding errors and quality problems in current Java distributions create possibilities to create covert channels, cause resource blocking and denial-of-service attacks. To make things worse Java components are often deployed according to the AllPermissions antipattern with non-restrictive security settings, which allows bugs on the system layer to be exploited by attackers. Coping with this antipattern from the user side is connected with the definition of adequate permission sets. A tool that automates this time consuming task is presented as a refactoring for the AllPermission antipattern.

1 Introduction

Java is one of the few programming languages that was designed from the beginning with security goals in mind[Go99b]. In the first versions (1.0 and 1.1) a simple sandbox model was available that allowed containment for remote code. Current editions of Java extend this model to a fine-grained architecture that extends the locality based approach of the first sandbox model with more trust-establishing parameters such as the codebase of the current instruction, the signature of the code signer and with the Java Authentication and Authorization System (JAAS) framework also the identity of the current user. An application can be secured with a policy file to define what actions are allowed according to the actual trust level of the current user and executed code parts.
As with all security architecture specifications, hackers typically do not try to attack the system through the front door, they seek errors in the implementations and try to exploit covert channels and triggers to the layer below[Go99a] to bypass security mechanisms.
The following sections will show that although equipped with language based security features such as type safety, antipatterns such as covert channels can be found in the Java architecture that allow to exploit logical errors in the code of trusted libraries. In addition false assumptions in the packaging the of the classes in the trusted system libraries of the JDK offer a harmful range of ready-to-call functionality to the attacker.

2 Java Security

Traditional requirements towards programming languages are reliability, performance, flexibility, abstraction and broad applicability. Induced by the growing importance of distributed middleware models like CORBA[Ob01] and the Java 2 Enterprise Edition (J2EE[Sh]) the requirements towards security gained importance. This is due to the fact, that code could not be considered as trusted as it is downloaded on demand from unknown remote sites which trustworthiness is typically unknown.

The experiences with traditional programming languages like C and C++ were the basis to design Java [GJSB00]from scratch with security goals in mind and avoid the potential vulnerabilities of direct memory access, pointer arithmetic and arbitrary type casts. Such concepts are found with an emphasis on system security in the Java 2 Standard Edition (J2SE[Sua]) as platform independent framework for desktop applications as well as with an additional emphasis on access control in J2EE. Java is based on the principles of language based security which is enforced by the trusted kernel to provide code safety. These principles are control flow safety, memory safety, stack safety which support safety of the Java type system [Ko99]. Java code is stored in platform-independent bytecode format that is verified in accordance to the subdisciplines of type safety by the trusted kernel.

2.1 Secure Coding and Antipatterns

As the maintainer of the Java programming language Sun Microsystems has published a set of coding guidelines for secure Java programming [Suc]. The coding guidelines give hints when dealing with the following issues:

R1 Refrain from using non-final public static variables

R2 Reduce scope

R3 Refrain from using public variables

R4 Protect packages

R5 Make objects immutable if possible

R6 Never return a reference to an internal array that contains sensitive data

R7 Never store user-supplied arrays directly

R8 Serialization

R9 Native methods

R10 Clear sensitive information

Violation of these guidelines typically results in security antipatterns. The negative effects of ignoring the guidelines R1, R2, R3, R4 , R9 and combinations of them is discussed the following sections.

According to Brown et al. [BMIM98]

> the essence of an AntiPattern is two solutions, instead of a problem and a solution for ordinary design patterns. The first solution is problematic. It is a commonly occurring solution that generates overwhelmingly negative consequences. The second solution is called the refactored solution. The refactored solution is a commonly occurring method in which the AntiPattern can be resolved and reengineered into a more beneficial form.

In the following the negative consequences of the integration of the `org.apache.*` classes into JDK 1.4.x are demonstrated.

2.2 Non-final public static methods and fields in the JDK-Packages

The packages in the JDK are subdivided in the classes that form the core Java language (`java.lang.*`), supporting classes (`java.*`), implementation dependent `sun.*` prefixed classes and other packages. Beginning with version 1.4.x of the JDK the ”other” packages contain several classes from the Apache Xalan and crimson libraries to provide functionality to process XML and XSLT data [Sub]. A typical Java class consists of instance related methods and fields and of class related methods and fields, which are identified via the `static` keyword. It is common that data structures that are designed to be available globally are implemented with `public static final` modifiers, which makes them available throughout the application (`public`), binds them to the class (`static`) and makes them writable only once (`final`).

According to secure coding guideline `R1` declaration of non-final public static methods and fields is harmful, this is especially true when executing code from untrusted sources.

The Java Plugin [Su03a] from Sun Microsystems is designed to execute Java code from untrusted sources in an Internet browser like the Microsoft Internet Explorer or Mozilla. It starts a single JVM and creates an instance of the applet classloader class for each loaded applet. The Java virtual machine uses private `ClassLoader`objects to create separate address spaces between processes to provide an environment of code confinement. Any user class may be loaded multiple times if it is loaded by private class loaders. In contrast classes residing in the `rt.jar` (JDK boot system classes) are loaded once and their static variables are strict singletons[GHJ95]. The classes of the Apache XML and XSLT utility packages (`org.apache.*`) expose several public static fields and property values that can be set from untrusted user code. These static variables become risk and threat to integrity when they can be modified by untrusted code in such a way that they modify system behavior. As Sun did not rename these packages with a `sun.*` prefix the XML utility

classes technically became part of the public Java interface available to all code types including applets, although the `org.apache` namespace is not mentioned in the official JDK documentation. The applet sandbox[Go99b] permission set allows access to packages and classes of the public interface, which enables class loaders to define and access these classes, in contrast to the unaccessible classes in private `sun.*` packages as such as `sun.security.util.PropertyExpander` class. These classes are restricted from definition and direct access by the default security manager policy settings located in `jre/lib/security/java.policy`.

2.3 Covert Channel and Triggers Antipatterns

According to Bishop[Bi00] a Covert channel is

> a path of communication that was not designed to be used for communication

An attacker can find out potential covert channels in the JDK classes by analyzing the communication and calling paths between the classes. Those classes are packed in a Java archive (jar file), which is an extension of the Zip-Format by a manifest which holds Java specific meta information.

An attacker may use bytecode engineering techniques [Sc02] to perform the following analysis while scanning the jar file:

1. Acquisition of a list of the public classes in the jar file as these are accessible via the reflection API to outer Java scripting such as Beanshell, Javascript or stored procedures in several JDBC drivers.
2. Scan these classes for public, static non-final fields and methods. The acquired fields can be used to establish covert channels for processes running inside the same VM. The acquired methods can be used to trigger actions in the mentioned script environments.

3 Covert channels and triggers

Two misuse cases will be shown that exhibit the danger of exploiting these shared resources and functionality. The scenarios are:

1. **Covert channels** that allow unsigned applets to communicate to other signed and unsigned applets
2. **Covert triggers** that allow to execute arbitrary programs on the machine running the VM via remote JDBC calls

According to the security pattern framework by Yoder and Barcalow[YB97] a single access point limits the entrance to critical functions and resources of applications. This pattern is typically undermined by covert channels that bypass the ”single entrance” premise. The Java security manager concept that enforces the applet sandbox is an implementation of such a single access point. It technically intercepts the critical calls by referring to the policy in place prior to resource access. The strict default applet policy to protect the integrity of the users workplace when working with mobile contents loaded from untrusted sources.

3.1 Covert channels in the JDK

3.1.1 Applet covert channel

One of the restrictions enforced by the default applet security manager is the limitation of package access, limiting access to methods and fields of classes from a package that not prefixed by a `sun` top-level package name.

3.1.2 Detection techniques

In order to find the potential covert channel and triggers inside the JVM, the class files residing in the Java runtime libraries (`rt.jar`) were inspected. This was done with the help of the Bytecode Engineering Library (BCEL) which is part of the Apache Jakarta project[Daa]. Scanning the public classes in `rt.jar` of the Java version 1.4.2_03 resulted in 91 public static non-final fields and 4286 public static methods. These were exported to XML files via an XMLEncoder object to allow optional further automatic processing.

3.1.3 Proof-of-concept

In order to demonstrate the danger of covert JDK channels with a proof-of-concept implementation the public and static field `LANGUAGE` from the `XSLTProcessorVersion` class inside the `org.apache.xalan.processor` package was chosen, which resides in `rt.jar`. Then an applet was constructed that tested read and write access this field. This applet was distributed to three different remote web sites. The three identical applets were then loaded into a single web browser in different frames, which means sharing the same VM by all applets. During runtime applet access to the variable from the three applets was tested. As expected all three applets were allowed to modify this static variable and exchange data and serializable (Guideline `R8`) objects which were serialized via an `ObjectOutputStream` in `String` object and therefore violated the sandbox restrictions. The risk potential for this behavior ranges from denial-of-service of the XSLT functionality to sandbox escape by bypassing containment through covert channel communication. Additionally signed applets may leak information to unsigned applets which may circumvent the Bell LaPadula[BL] privacy considerations intended by the applet developer.

To improve quality of the Java runtime environment the issue was submitted to the Java bug database and was labeled with the internal number 236774. The bug was considered to be new and will be made visible to the public in the database after a refactored version is available, which will be case with the 1.4.2_05 version of the Sun JDK.

3.1.4 Memory reading applet

A covert channel to physical system memory was found by the author [Su03b] in the Java Media Framework (JMF), which is a toolset that allows to play multimedia elements such as music, movies and other stream data within pure Java applications. As the JMF is concerned with access to system hardware functionality such as the sound card and graphics equipment and uses several native codecs, performance-oriented and therefore platform dependent code has to be accessed by the JMF libraries. The JMF libraries are installed as endorsed library to the `lib/ext` directory of the JRE. Libraries in this directory are loaded by the boot class loader and are trusted fully by the applet security manager (which equals an `AllPermission` setting). A result of bytecode analysis of the classes of the `jmf.jar` in version 2.11.c a covert channel was identified that allowed indirect access to the system memory which is a violation of the strict containment premise enforced by the permission sets of the Java sandbox. The `NBA` class (NativeBlockAccessor) is responsible to provide a specified communication area between the components in pure Java and the native memory storage. The cause for the problem is the inappropriate guarding of an internal variable of the NBA class inside the JMF. This is a violation of the `R2` (reduce scope) secure coding guideline and causes the memory access antipattern. The field `data` holds the pointer to a native memory block. By subclassing the `NBA` class, the information stored in `data` was made available to arbitrary Java applets, which after using conversion routines were able to map Java byte values to the exposed system memory.

3.2 Covert trigger

A **covert trigger** in analogy to covert channels is defined here as

> A trigger of actions that was not designed to trigger actions

3.2.1 Applet floppy hardware attack antipattern

A containment problem can be raised with a covert trigger. Due to a implementation logic error in the Java virtual machine for the Windows platform, the security manager check is called after the physical check whether a floppy drive is available in the disk drive. When running the `createXmlDocument` of the `XmlDocument` class of the `org.apache.crimson.tree package` in an endless loop the machine (tested with the Java Plugin in IE6) stops working because the floppy drive is busy with antagonistic accesses to the disk. The hardware stress applet can lead to overheating in the floppy drive which might cause physical damage to the drive and the other components of the affected

PC,but even on systems without floppy drives the applet allows a simple denial-of-service attack by accessing other blocking device types via their file names. This antipattern was created by executing privileged code (triggering physical floppy access) without proper access control checks (`FilePermission`).

3.2.2 JDBC macros and covert triggers via remote command injection

JDBC is a Java centric standard to establish client-server database applications. Java clients use services of a database server via remote JDBC calls. Three 100% pure java databases were tested for vulnerabilities in regards of the remote command injection antipattern. The databases were:

HSQLDB aka Hypersonic SQL[HS], an open-source SQL database bundled with JBOSS 3.x

Pointbase DB 4.6 [Dab], a commercial SQL database system, is bundled with the J2EE 1.4 reference implementation

Cloudscape SQL [IB] from IBM is a standalone 100% pure java database and is also available bundled with the Websphere application server

By the time period of penetration testing local installations of all three databases, these products were not designed to run with a Java security manager. As a consequence they have been found to be vulnerable to remote command injection, information disclosure. A simple JDBC SQL statement was sufficient to start an arbitrary executable on the host running a SQL database, which open a covert trigger. With a SecurityManager in place this would only be feasible with an explicit ”execute”FilePermission.

As a demonstration the following SQL statement injects Java code in the address space of the server VM.

```
CREATE FUNCTION COMPDEBUG (IN P1 boolean) returns VARCHAR(100)
LANGUAGE JAVA NO SQL EXTERNAL NAME
"org.apache.xml.utils.synthetic.JavaUtils::setDebug"
PARAMETER STYLE SQL;
SELECT COMPDEBUG(true) FROM SYSUSERS;
CREATE FUNCTION SETPROP (IN P1 VARCHAR(100),
IN P2 VARCHAR(100)) returns VARCHAR(100)
LANGUAGE JAVA NO SQL EXTERNAL NAME
"java.lang.System::setProperty" PARAMETER STYLE SQL;
SELECT SETPROP('org.apache.xml.utils.synthetic.javac',
'cmd.exe') FROM SYSUSERS;
CREATE FUNCTION COMPILE (IN P1 VARCHAR(100),
IN P2 VARCHAR(100)) returns VARCHAR(100)
LANGUAGE JAVA NO SQL EXTERNAL NAME
"org.apache.xml.utils.synthetic.JavaUtils::JDKcompile"
PARAMETER STYLE SQL;
SELECT COMPILE('', '/c notepad.exe') FROM SYSUSERS;
```

The example (in Pointbase SQL syntax) starts a `notepad.exe` on the host executing the JDBC database as a proof of concept. As every other more harmful executable such as a remote shell could be started as well. The SQL statement is equal in functionality to the following Java code:

```
{
 org.apache.xml.utils.synthetic.JavaUtils.setDebug(true);
 System.setProperty("org.apache.xml.utils.synthetic.javac","cmd.exe");
 org.apache.xml.utils.synthetic.JavaUtils.JDKcompile("","/c notepad.exe");
}
```

The syntax between the databases differ in small details but the possibility of injection was shown for HSQLDB, Pointbase and Cloudscape SQL, when running on a Sun JDK 1.4.x virtual machine.

The code does the following:

- It sets a debug mode
- Then an internal variable of the Xerces classes is set that defines the default Java compiler to an arbitrary executable program
- Finally it calls the compile function, which invokes an executable file (`cmd.exe`).

Due to the individual mapping mechanism of the SQL data types to the Java data types which is different for each database product. The smallest set of available functionality was restricted by the mapping of HSQLDB. Via bytecode engineering candidate methods in rt.jar were retrieved that have a `public static void` signature with primitive (such as boolean) or `java.lang.String` input values. This set was scanned whether the member calls privileged code parts such as file operations and shell execution. Beside the presented examples other functionality can be called that can be misused for log manipulation or abnormal program termination such as demonstrated in the next SQL statement which calls a vulnerable JVM routine in the sun packages which causes an immediate JVM crash in the remote JDBC server. This vulnerability was communicated to Sun by the author in 2002 but is not fixed until today.

```
CREATE FUNCTION CRASH5(IN P1 VARCHAR(20)) RETURNS VARCHAR(20)
 LANGUAGE JAVA
 NO SQL
 EXTERNAL NAME "sun.misc.MessageUtils::toStderr"
 PARAMETER  STYLE SQL;
SELECT CRASH5(null) from SYSUSERS;
```

The SQL statement is equal in functionality to the following Java code.

```
{
  sun.misc.MessageUtils.toStderr(null);
}
```

The enhanced problem with Hypersonic SQL which was deployed with JBOSS application server 3.2.1 was the lack of a security manager. It opened a listing TCP port 1701 that

accepted anonymous JDBC calls, which allowed to inject arbitrary remote commands into the J2EE process as the JDBC DB was running in the same VM as the J2EE server process. By closing the open port and switching default configuration to the internal JVM communication mode the vulnerability has been fixed by the JBOSS developers after a vendor notification. The Pointbase DB has also been refactored with version 4.8 after a vendor notification by deploying an optional security manager. IBM admits that is a good idea to apply a security manager. As the secure coding guidelines `R1, R2, R4` are violated when allowing JDBC macros,

4 Structural Refactoring of the AllPermission antipattern

Refactorings are a means to turn anti-patterns in a good solution by applying well-known and scalable patterns. The problems identified above were problems of security unaware coding which in combination with unlimited access rights is very harmful. This is typically caused by an `AllPermissions` or equivalent settings, that is the default case when there is no instance of a SecurityManager is installed with a virtual machine. Running a system with a minimal set of privileges is therefore an admirable goal as it enhances the assurance level. Even when a security manager is installed determining the range of a minimal permission set is a time-consuming task. Therefore jChains has been developed. The functionality it provides was first used when a minimal set of rules was needed to refactor the JDBC macros antipattern.

4.1 jChains

jChains is a custom security manager framework records the permissions needed for the codebases (jars) of J2SE applications running under the access control enforcement of the Java security manager. This allows security unaware applications to be run under a security manager. The resulting policy file is recorded while running the program and is useful as a starting point when developing a security policy for a Java application. When run against libraries when source is not available it is useful for reverse engineering, revealing the permission needed to use the libraries. This is helpful when a developer does not trust the jar , and do not want to grant it the AllPermission free ride ticket. jChains is designed to acquire a policy set from a Java application by doing a training mode and later enforce this policy set in a production mode.

4.1.1 Refactoring JDBC macros with jChains

A typical use case for jChains was evaluating a useful set of rules for the Pointbase database when providing the vendor with a possible security-related refactoring suggestion.

```
grant codeBase "file:${pointbase.lib}${file.separator}-" {
permission java.net.SocketPermission "*:1024-", "connect,resolve,accept";
permission java.io.FilePermission "<<ALL␣FILES>>", "read,write,delete";
permission java.util.PropertyPermission "*", "read";
};
```

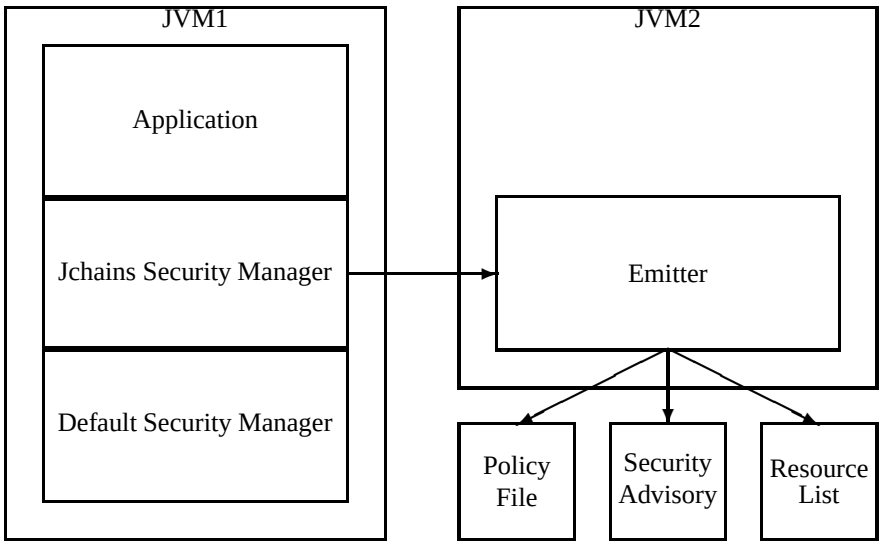

Figure 1: jChains-Architecture

jChains [Sc] is either configurable in a local mode or can communication to a remote CORBA server. CORBA was chosen in favor of RMI because of it's independence from a concrete implementation language. The distributed CORBA mode allows to decouple permission recording from permission evaluation. This is useful for remote permission training scenarios, e.g. when there is no direct user access to the system hosting the virtual machine. As benefit inherited from CORBA location transparency jChains may also operate locally.

5 Conclusion

It has been shown that the Java platform is not free of security related issues although providing a large scale of default security precautions. But these mechanisms cannot help in a case when the attacker attacks the system on a semantical level below the mechanism implementation, which typically is the case when antipatterns such as logical errors and packaging structures are exploited. These coding related antipatterns are often caused by circumventing the assumptions of the secure coding guidelines. To provide a refactoring for the AllPermissions antipattern jChains has found acceptance under developers (is a published freshmeat project and is hosted at the `java.net` site. In addition jChains

was useful for refactoring the permission set of the Pointbase SQL server product and is currently in evaluation by a major german banking group to adjust the appropriate policy set for the J2EE thin-clients of a banking client-server system.

References

[Bi00] Bishop, M.: *Computer Security*. Addison-Wesley. 2000.

[BL] Bell, D. und LaPadula, L.: Secure computer systems. Technical report. Air Force Elec. Syst. Div.

[BMIM98] Brown, W. J., Malveau, R. C., III, H. W. S. M., und Mowbray, T. J.: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons. 1998.

[Daa] Dahm, M. BCEL manual. `http://jakarta.apache.org/bcel/manual.html`.

[Dab] Data Mirror Software. Pointbase product homepage. `http://www.pointbase.com`.

[GHJ95] Gamma, E., Helm, R., und Johnson, R.: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley. 1995.

[GJSB00] Gosling, J., Joy, B., Steele, G., und Bracha, G.: *The Java Language Specification Second Edition*. Addison-Wesley. Boston, Mass. 2000. `http://citeseer.ist.psu.edu/gosling00java.html`.

[Go99a] Gollmann, D.: *Computer Security*. Wiley & Sons. 1999.

[Go99b] Gong, L.: *Inside Java 2 Platform Security*. Addison-Wesley. 1999.

[HS] HSQLDB Development Team. Hsqldb product homepage. `http://hsqldb.sourceforge.net`.

[IB] IBM Corporation. Cloudscape product homepage. `http://www-306.ibm.com/software/data/cloudscape`.

[Ko99] Kozen, D.: Language-based security. In: *Mathematical Foundations of Computer Science*. S. 284–298. 1999. `http://citeseer.nj.nec.com/kozen99languagebased.html`.

[Ob01] Object Management Group: *The Common Object Request Broker: Architecture and Specification*. Object Management Group. 2.5. September 2001.

[Sc] Schoenefeld, M. jChains homepage. `http://jchains.org`.

[Sc02] Schoenefeld, M.: Security Aspects in Java Bytecode Engineering. In: *Blackhat USA 2002 Proceedings*. 2002.

[Sh] Shannon, B. Java™2 Platform Enterprise Edition Specification, v1.4. `http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf`.

[Sua] Sun Microsystems. Java 2 Platform, Standard Edition v 1.4 Datasheet. `http://java.sun.com/j2se/1.4/datasheet.1_4.html`.

[Sub] Sun Microsystems. Java™API for XML processing release notes. http://java.sun.com/webservices/docs/1.2/jaxp/ReleaseNotes.html.

[Suc] Sun Microsystems: *Security Code Guidelines*. http://java.sun.com/security/seccodeguide.html.

[Su03a] Sun Microsystems. Java Plug-in 1.4.2 Developer Guide. 2003. http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/contents.html.

[Su03b] Sun Microsystems. Security Advisory on Java Media Framework. 2003. http://sunsolve.sun.com/pub-cgi/retrieve.pl?doc=fsalert%2F54760.

[YB97] Yoder, J. und Barcalow, J. Architectural patterns for enabling application security. 1997. http://citeseer.nj.nec.com/yoder98architectural.html.