

Secure Mobile Agent Systems Using Java: Where are We Heading?

Walter Binder
CoCo Software Engineering
Margaretenstr. 22/9, 1040 Vienna, Austria
w.binder@coco.co.at

Volker Roth
Fraunhofer IGD
Rundeturmstr. 6, 64283 Darmstadt, Germany
vroth@igd.fhg.de

ABSTRACT

Java is the predominant language for mobile agent systems, both for implementing mobile agent execution environments and for writing mobile agent applications. This is due to inherent support for code mobility by means of dynamic class loading and separable class name spaces, as well as a number of security properties, such as language safety and access control by means of stack introspection. However, serious questions must be raised whether Java is actually up to the task of providing a secure execution environment for mobile agents. At the time of writing, it has neither resource control nor proper application separation. In this article we take an in-depth look at Java as a foundation for secure mobile agent systems.

Keywords

Java, mobile agents, security, survey

1. HOME-GROWN MOBILE CODE

The proliferation of the Java programming language [8] led to the development of numerous mobile agent platforms. Actually, Java seems perfect for developing an execution environment for mobile agents, because Java offers many features that ease its implementation and deployment. Java runtime systems are available for most hardware platforms and operating systems. Therefore, mobile agent platforms that are built on Java are highly portable and run seamlessly on heterogeneous systems. Furthermore, mobile agents profit from continuous performance and scalability enhancements, such as increasingly sophisticated compilation techniques and other optimizations, which are provided by the underlying Java Virtual Machine (JVM) [10].

In addition to portable code, Java offers a *serialization* mechanism allowing to capture a mobile agent's object instance graph before it migrates to a different host, and to resurrect the agent in the new environment. Java also supports dynamic loading and linking of code by means of a hierarchy of *class loaders*. A class loader constitutes a separate *name space* that can be used to isolate classes of the agent system and of different agents from each other.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2002, Madrid, Spain
© 2002 ACM 1-58113-445-2/02/03...\$5.00.

In general, mobile agent platforms execute multiple agents and service components concurrently in a time sharing fashion. Java caters for this need by means of multi-threading. Java is also a *safe* language, which means that the execution of programs proceeds strictly according to the language semantics (this is not entirely true, as we discuss in section 4). For instance, types are not misinterpreted and data is not mistaken for executable code. The safety properties of Java depend on techniques such as *byte-code verification*, *strong typing*, *automatic memory management*, *dynamic bound checks*, and *exception handlers*. On top of that, the Java 2 platform includes a sophisticated security model with flexible access control based on dynamic stack introspection.

In summary, Java is highly portable and provides easy code mobility. This caused numerous mobile agent systems based on Java being developed and experimented with. From the point of security, two points can still be criticized: first, all systems focus on a particular aspect of agent mobility and none address all problems simultaneously, whose solution is required to come up with a system ready for field use. This is particularly true for security. Second, practical experience with Java shows that considerable security concerns remain, which are subject of the remainder of this article.

2. OBJECT MANAGEMENT

A Java class is represented in the JVM by a *class object*. The class object that represents a class is initialized upon the first *active* use of that class. Mere declaration of a typed variable does not constitute active use and does not trigger initialization of the class that represents that type. However, as soon as a static method actually declared in that class is invoked, or a constructor is invoked, or a non-constant field is accessed, the class is initialized. When a Java class is initialized, first its class variable initializers and static initializers are executed. This opens a loophole for executing potentially malicious code before even the first instance is generated.

Further loopholes are hidden in the *Serialization Framework* of Java. During deserialization of an object instance, no constructors are invoked. The fields of unmarshalled objects are initialized directly. However, if the object that is unmarshalled implements a method of the exact signature

```
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

then this method is invoked in the deserialization process. A similar callback method exists for object serialization. This means that:

- agent state appraisal and authentication must complete before the agent instance is marshalled and the first agent class is loaded into the VM. Once this happens, it is already too late to defend against malicious agents.
- whenever an agent class is initialized, or an agent instance is marshalled or unmarshalled, the agent may grab the current thread and do all kinds of things the server least expects.

The loopholes described above potentially allow agents to run code before the system expects it. Another loophole in Java's garbage collector allows to plant an egg that is hatched after the agent is (officially) already gone. When the VM detects that no strong references exist anymore to some object, then it garbage collects this object and reclaims the memory occupied by it. However, before the garbage collector thread does this, it gives the object a chance to clean up any leftover state by invoking the finalizer of that object (if it is implemented). Consequently, if the method call does not terminate, no garbage gets collected anymore, and the VM soon chokes from a lack of memory. A less obvious attack would be to set an alarm (by means of a new thread) that triggers a destructive method only after a delay. In that case, the log files of the agent system (if there are any) will give evidence that the malicious agent already left the server and thus is not likely to be responsible for the crash. In any case, it becomes complicated to find out what actually happened, and to prove it to somebody else.

The developer of an agent system might be tempted to eliminate these loopholes by refusing to load any classes that implement the finalizer method. However, this is insufficient, since there are several classes in the Java core packages that already implement a finalizer and invoke callbacks in it. A malicious agent might, for instance, bring a class that inherits e.g., from *FileInputStream*, *FileOutputStream*, or *ZipFile*. Any of these classes invokes the method *close()* in its finalizer. Hence, rather than overriding the finalizer itself, malicious code may override the *close()* method. Consequently, all classes that inherit from one of these must be blocked as well (at least, if they implement *close()*).

Regardless how an agent becomes executed, once it runs it may hamper other threads and agents in a variety of ways. One way is to launch a *denial of service* (DoS) attack by means of synchronizing on class locks. In Java all class objects of classes loaded by the system class loader are visible, and some classes synchronize on their class locks. One prominent example is illustrated below:

```
synchronized(Thread.class) { while (true); }
```

Clearly, touching an object is a dangerous thing. Yet, mobile agent systems often allow uncontrolled aliasing (sharing of object references), which is both convenient and typical of object-oriented programming. Again, DoS attacks can take on various forms. Catching the current thread is one possibility, keeping references to other agents' objects is another. As long as a strong reference to some object exists, it will not be garbage collected. In Java it is not possible to revoke an object reference. However, dynamic proxy generation mechanisms that are available since JDK version 1.3 can be applied to this problem.

In summary, the concurrent execution of multiple agents requires isolation boundaries, where the passing of references has to be controlled. Marshalling and unmarshalling of objects must be done by

a thread that can be sacrificed, or belongs already to a sandbox that is set up in advance for the object in question. Migration must take place only after all threads of a mobile agent have terminated and none of its classes is on the stack of any running thread anymore (or referenced by any object with a strong reference). Abuse of class locks is a matter that is addressed best by means of a modification of Java.

3. THREAD MANAGEMENT

The Java language includes a set of APIs and primitives to manage multiple concurrent threads within a Java program. Synchronization between threads is based on *monitors*, which are associated with objects. Java *synchronized{}* statements are mapped to matching *monitorenter* and *monitorexit* instructions at the bytecode level. Monitors are implemented based on *locks*; each object has an associated lock that is used whenever a *synchronized{}* statement refers to that object. Methods of an object can be declared *synchronized*, which implies that the method is executed in a monitor whose lock is the one associated with that object. Instance methods are associated with the lock of the object instance, whereas static methods are associated with the lock of the object instance that represents the object class (and which is of type *java.lang.Class*).

3.1 Inconsistency due to Asynchronous Termination

One important function of a mobile agent platform is the termination of agents. When an agent migrates or terminates, all of its allocated resources should be reclaimed as soon as possible. That is, all threads of the agent shall be stopped and memory allocated by the agent shall become eligible for garbage collection. Especially, when a misbehaving agent is detected, it has to be removed from the system with immediate effect.

Java allows to asynchronously terminate a running thread by means of the *stop* method of class *java.lang.Thread*. This method causes a *ThreadDeath* exception to be thrown asynchronously in the thread to be stopped. Unfortunately, thread termination in Java is an inherently unsafe operation, because the terminated thread immediately releases all monitors. Consequently, objects may be left in an inconsistent state. As long as these objects are exclusively managed by the agent to be removed from the system, a resulting inconsistency may be acceptable.¹

However, if a thread is allowed to cross agent boundaries for communication purpose (e.g., inter-agent method invocation), the termination of a thread has to be deferred until it has completed executing in the context of other agents. Otherwise, the termination of one agent may damage a different agent that is still running in the system. Unfortunately, delayed thread termination prevents immediate memory reclamation, because references to objects of the terminated agent may be kept alive on the execution stack of the thread. Even worse, if shared objects, such as certain internals of the JVM, are left inconsistent, asynchronous thread termination may result in a crash of the JVM. For this reason, the *stop* operation has been deprecated in the Java 2 platform.

To solve these problems, the mobile agent platform has to enforce a

¹If the agent state is captured (e.g., serialized) after termination, inconsistencies may corrupt the further execution of the agent on other platforms. The agent is responsible to freeze its non-transient state before requesting migration.

```

while (true) {
    try { while (true); } catch (Throwable t) {}
}
↓
while (true) {
    try {
        while (true);
    }
    catch (Throwable t) {
        if (t instanceof ThreadDeath) { throw t; }
    }
}

```

Figure 1: Catching *ThreadDeath* can be prevented by rewriting bytecode in a way that is functionally equivalent to the given Java code transformation.

thread model where each thread is bound to a single agent. Threads must not be allowed to cross agent boundaries arbitrarily. Upon the invocation of a method in a different agent, a thread switch is necessary. The called agent has to maintain worker threads to accept external method calls. However, this approach negatively affects performance, because thread switches are rather expensive operations.

To ensure the integrity of shared data structures and of JVM internals, the mobile agent system has to enforce a user/kernel boundary, where shared structures are manipulated only within the kernel. With the aid of a locking mechanism, it is possible to ensure atomic kernel operations. That is, requests for asynchronous termination are deferred until the thread to be stopped has left the kernel. Because kernel operations can be implemented with a short and bounded execution time, domain termination cannot be delayed arbitrarily. All critical JVM operations have to be guarded by a kernel entry. Again, this solution causes some overhead.

3.2 Interception of Asynchronous Termination

There are further problems with asynchronous thread termination: The *stop* method does not guarantee that the thread to be killed really terminates, because the thread may intercept the *ThreadDeath* exception. For instance, consider the upper code fragment in figure 1, which cannot be terminated easily.

Note that not only exception handlers may intercept *ThreadDeath* exceptions, but *finally*{ } clauses may prevent termination as well. However, the Java compiler maps *finally*{ } statements to special exception handlers. Thus, it is sufficient to solve the problem with exception handlers that catch *ThreadDeath* or a superclass thereof.

The JavaSeal mobile agent kernel [5] enforces a set of restrictions on exception handlers that may catch *ThreadDeath*, in order to ensure the termination of such handlers. However, this approach imposes severe restrictions on the programming model. For instance, untrusted agents may not use *finally*{ } clauses. Furthermore, the JavaSeal implementation is incomplete, as a *monitorexit* instruc-

tion² in an exception handler may throw *NullPointerException* or *IllegalMonitorStateException*, which can be caught by user code.

Another solution to this problem involves rewriting of agent bytecode so that *ThreadDeath* exceptions are immediately thrown again by all exception handlers. This approach is used in the J-SEAL2 mobile agent kernel [2]. Figure 1 shows a portion of Java code and its rewritten counterpart. For the ease of reading, we give the transformation at the Java level, whereas rewriting would be done actually at the JVM bytecode level.

3.3 Scheduling

Neither the Java language [8] nor the JVM specification [10] specify the scheduling of Java threads. Therefore, it is not guaranteed that, on every Java platform, high-priority surveillance threads preempt other threads. A related problem is priority inversion. This means that a high-priority thread may have to wait until a low-priority thread releases a monitor. However, the low-priority thread may not be scheduled if there are other threads ready to run that have a higher priority.

Fortunately, most standard Java runtime systems offer native threads that are scheduled by the operating system. Depending on the scheduler of the operating system, often a high-priority surveillance task executes as expected. Priority inversion may be addressed by temporarily raising the priority of a thread executing a critical section. However, in many JVMs adapting thread priorities is an expensive operation, since it triggers the scheduler.

The realtime specification for Java [4] specifies priority-based preemptive scheduling with at least 28 different levels of priority for all compliant implementations. The realtime specification covers many other topics important for realtime systems. Therefore, standard Java runtime systems will not likely conform to the realtime specification, because they target environments without realtime requirements, and probably the underlying operating system does not offer any realtime guarantees either. Consequently, we think that a subset of the realtime specification should be integrated into a new version of the standard Java specification, so that applications that depend on scheduling mechanisms – such as mobile agent systems – may run consistently across different JVM implementations.

4. BYTECODE VERIFICATION

Java relies on static and dynamic checks to ensure that the execution of programs proceeds according to the language semantics. Before a program is linked into the JVM, the Java bytecode verifier performs static analysis of the program to make sure that the bytecode actually represents a valid Java program. Dynamic checks (e.g., array bounds checks) are incorporated in many JVM instructions.

Unfortunately, bytecode verifiers of several current standard Java implementations also accept bytecode that does not represent a valid Java program. The results of the execution of such bytecode is undefined, and it may even compromise the integrity of the Java runtime system.

At the Java bytecode level, the allocation of an object is separated

²The compilation of a *synchronized*{ } statement creates an exception handler whose task is to release the monitor in case of any exception. Because synchronization is an important concept of the Java language, JavaSeal allows agents to use the *monitorexit* instruction within exception handlers.

```

static void captureMonitor(java.lang.Class)
  0 aload_0
  1 monitorenter
  2 return

```

Figure 2: Example bytecode which acquires a lock that is not released after completion.

from its initialization. One important task of the Java verifier is to prevent uninitialized objects from being used. In [6] the authors take an in-depth look at object initialization in Java and define rules to be enforced by the Java verifier. However, one particular issue is not addressed: Finalizers will be invoked on uninitialized objects, which completely undermines the properties on object initialization guaranteed by the Java verifier.

Another source of problems are the synchronization primitives of the Java bytecode. The example given in figure 2 depicts the bytecode of a method that acquires a class lock without releasing the lock after completion. This code sequence does not constitute a valid Java program, because the *monitorenter* instruction (that acquires a lock) is not paired with a matching *monitorexit* (that releases the lock). Neither is an exception handler present that releases the lock in case of an exception. Nonetheless, several Java verifier implementations do not reject this code. The effects of executing this code are undefined and depend on the particular JVM implementation. We tested the method invocation *captureMonitor(Thread.class)* with 3 different JVMs on a Windows platform and observed varying outcomes with each of them:

Sun Hotspot Server VM 2.0: An *IllegalMonitorStateException* is thrown at the end of the method and the monitor is released.

Sun JDK 1.2.2 Classic VM: No exception is thrown and the monitor remains locked until the thread that has acquired the monitor terminates.

IBM JDK 1.3.0 Classic VM: The monitor is not released even after the locking thread has terminated. Subsequent attempts by other threads to create new threads are blocked, because thread creation involves a static synchronized method, which waits for the release of the class lock. In fact, this kind of attack is similar to the DoS attack shown in section 2. However, this attack is even worse, because the lock is not released after all attacker threads have terminated, whereas the attack in section 2 can be resolved by stopping the attacking thread.

Figure 3 depicts another example of disarranged bytecode that is not rejected by several standard Java verifiers. In this bytecode sample, the target of the exception handler is the first instruction protected by the same handler. Such a construction is not possible at the Java language level, because a *try{}* block cannot serve as its own *catch{}* clause. At bytecode position 1 there is an infinite loop (*goto 1*), which is protected by the exception handler.³ In case

³The *aconst_null* instruction at position 0 ensures that there is always a single reference on the stack at code position 1 (this constraint is enforced by the Java verifier). When an exception is caught, the stack is cleared and a reference to the exception object is pushed onto the stack. The *return* instruction is never reached.

```

static void preventTermination()
  0 aconst_null
  1 goto 1
  4 return
Exception table:
  from   to target type
    1     4     1   <Class java.lang.Throwable>

```

Figure 3: Example bytecode with a disarranged exception handler.

of an exception, the handler continues the same loop. Therefore, it is not possible to stop a thread executing such code. Even the transformation shown in figure 1 does not help, since its application would cause an infinite loop of catching and re-throwing the same *ThreadDeath* exception.

In order to prevent such attacks, improved bytecode verification is necessary. A better solution would be the definition of an alternative Java class format, which enables simpler verification. For instance, *Slim Binaries* [7] encode the abstract syntax tree of a program and can be verified easily, because the code can be restricted to valid syntax trees of the programming language. Thus, expensive bytecode verification can be avoided.

5. SECURITY MODEL

One of the most prominent security features in the Java security model is the fact that permissions of a thread are limited to the permissions granted to the least privileged class on its execution stack. Permissions are assigned by the *class loader* when the class is defined. Any class can check whether the current thread has a particular permission by invoking the *access controller* with a template of the permission that shall be checked. The access controller responds by throwing an exception if the permission is not granted, and silently returns otherwise.

Classes can execute *privileged actions*, which means that the class assumes responsibility for subsequent actions, and the permissions granted subsequently to the executing thread shall be the ones granted to the class that invoked the privileged action (in that case, stack introspection stops at the privileged context). New permission types can be introduced by means of a permission class that represents this type. While this gives great flexibility in terms of implementing security checks, this also lacks central control.

Security checks as well as privileged actions may be scattered throughout the class packages, and it is next to impossible to determine with certainty whether a given application actually enforces a particular security policy. Even a small error can have disastrous effects on the system security as a whole; in particular, multiple small errors culminate into bigger ones. For instance, write access to the VM binary or permission to execute native code is virtually equivalent with granting the *all permission*.

Rather than binding permissions to a class, permissions need to be bound to a particular *instance*. This can be achieved as follows: in its constructors, the trusted class stores a snapshot of the current access control context (ACC) in a private variable. Whenever the privileged action is executed, the stored ACC is set. The effective set of permissions granted to the thread is therefore the intersection of the privileges granted to the trusted class and the permissions current at the time when the class was created. Consequently, less privileged code will gain no additional permissions by instantiating

the trusted class, whereas the trusted *instance* can be used without restrictions. However, this is a strict design requirement and must be enforced consistently throughout the design and implementation phase.

A feature that is desirable for any mobile agent system is instant revocation of permissions. In other words, permissions granted to an agent can be expanded or withdrawn dynamically (e.g., when the agent misbehaves). One way to achieve this is to implement a custom *ProtectionDomain* that supports that feature, and which is assigned to all classes of the agent by means of a custom class loader. However, this approach is not guaranteed to work because protection domains may be compressed as a consequence of optimizations (although it does work as intended in the reference implementation of the JDK as of version 1.3).

In summary, the security model of the Java 2 platform is very flexible and powerful on the one hand, but on the other hand it is also very complicated and depends on the perfect meshing of all components of the application and the mobile agent middleware. This constitutes considerable risk, because breach of security or integrity of the VM may expose the account under whose authority the VM is executed.

6. MISSING JAVA FEATURES AND RELATED WORK

Several awkward problems complicate development of secure mobile agent systems in Java, several of which are caused by sharing class objects (classes loaded by the *system class loader*). It has been shown that abstractions comparable to the *process* concept in operating systems are required to create secure execution environments for mobile agents [1]. Consequently, Java runtime systems have to be re-designed for multi-tasking, and each agent should execute within a task of its own, isolated from all other tasks in the JVM. In particular, static variables of shared classes have to be replicated for each task. Recently, task isolation is being addressed by the Java Community Process [9].

A crucial feature that is also clearly missing is accounting and control of resource consumption (including but not limited to memory, CPU, threads, and network bandwidth). Even in the absence of malicious code, a poorly programmed agent may occupy too many resources, and hence prevent other agents from executing properly. Extreme resource overuse may crash the JVM or even the operating system. Again, a sound task model is a prerequisite for countering these threats.

Several researchers have developed special (non-standard) Java runtime systems capable of controlling resource consumption of tasks. However, mobile agent systems have to be deployed in large-scale heterogeneous environments and they must support many different hardware platforms and operating systems. Consequently, due to the limited portability of modified JVMs, most mobile agent platforms do not rely on such Java runtime systems. The Java Resource Accounting Facility⁴ (J-RAF) [3] is implemented in pure Java. Therefore, it can be used with standard JVMs. J-RAF is based on bytecode rewriting techniques to reify the resource consumption of applications. Memory allocation instructions are redirected to a controller object that denies object allocation if a limit is exceeded. CPU accounting is based on the number of executed bytecode instructions.

⁴<http://abone.unige.ch/>

However, bytecode rewriting techniques have drawbacks: If they are applied immediately before an agent is loaded, then rewriting causes considerable preprocessing overhead. Furthermore, rewritten code usually executes more instructions than the original code, which causes additional overhead at runtime. In [3] the authors show that the runtime overhead can be kept reasonably small with the aid of a carefully tuned accounting scheme. Nevertheless, bytecode rewriting techniques are limited to Java code, and resource consumption of native code is not measured.

7. CONCLUSION

The widespread distribution of Java as well as the mass of code and support that is available to Java developers makes it a sine qua non for mobile agent systems. This said, Java is probably the best and the worst that happened to mobile agents. The best because developing and deploying mobile agent systems became easy and highly portable; the worst because it is next to impossible to preserve Java's usefulness and to build a sufficiently secure system at the same time. In order to deploy industrial strength mobile agent systems that are robust against various forms of DoS as well as breaches of confidentiality, Java has to evolve from an application-level runtime system into a true operating system with proper accounting and application separation capabilities.

8. REFERENCES

- [1] G. Back and W. Hsieh. Drawing the red line in Java. In *Seventh IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, Mar. 1999.
- [2] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, Jan. 2001.
- [3] W. Binder, J. Hulaas, A. Villazón, and R. Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, Tampa Bay, Florida, USA, Oct. 2001.
- [4] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, MA, USA, 2000.
- [5] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, Oct. 1999.
- [6] S. Doyon and M. Debbabi. On object initialization in the Java bytecode. *Computer Communications*, 23(17):1594–1605, Nov. 2000.
- [7] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.
- [8] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1st edition, 1996.
- [9] Java Community Process. JSR 121 – Application Isolation API Specification. Web pages at <http://www.jcp.org/jsr/detail/121.jsp>.
- [10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.