Verification of class Files

Even though a compiler for the Java programming language must only produce class files that satisfy all the static and structural constraints in the previous sections, the Java Virtual Machine has no guarantee that any file it is asked to load was generated by that compiler or is properly formed. Applications such as web browsers do not download source code, which they then compile; these applications download already-compiled class files. The browser needs to determine whether the class file was produced by a trustworthy compiler or by an adversary attempting to exploit the Java Virtual Machine.

An additional problem with compile-time checking is version skew. A user may have successfully compiled a class, say PurchaseStockOptions, to be a subclass of TradingClass. But the definition of TradingClass might have changed since the time the class was compiled in a way that is not compatible with pre-existing binaries. Methods might have been deleted or had their return types or modifiers changed. Fields might have changed types or changed from instance variables to class variables. The access modifiers of a method or variable may have changed from public to private.

Because of these potential problems, the Java Virtual Machine needs to verify for itself that the desired constraints are satisfied by the class files it attempts to incorporate. A Java Virtual Machine implementation verifies that each class file satisfies the necessary constraints at linking time .

Linking-time verification enhances the performance of the interpreter. Expensive checks that would otherwise have to be performed to verify constraints at run time for each interpreted instruction can be eliminated. The Java Virtual Machine can assume that these checks have already been performed. For example, the Java Virtual Machine will already know the following:

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.
- The arguments to all the Java Virtual Machine instructions are of valid types.

The verifier also performs verification that can be done without looking at the code array of the Code attribute . The checks performed include the following:

- Ensuring that final classes are not subclassed and that final methods are not overridden . Checking that every class (except Object) has a direct superclass.
- Ensuring that the constant pool satisfies the documented static constraints; for example, that each CONSTANT_Class_info structure in the constant pool contains in its name_index item a valid constant pool index for a CONSTANT_Utf8_info structure.
- Checking that all field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

Note that these checks do not ensure that the given field or method actually exists in the given class, nor do they check that the type descriptors given refer to real classes. They ensure only that these items are well formed. More detailed checking is performed when the bytecodes themselves are verified, and during resolution.

There are two strategies that Java Virtual Machine implementations may use for verification:

- Verification by type checking must be used to verify class files whose version number is greater than or equal to 50.0.
- Verification by type inference must be supported by all Java Virtual Machine implementations, except those conforming to the Java ME CLDC and Java Card profiles, in order to verify class files whose version number is less than 50.0.

  Verification on Java Virtual Machine implementations supporting the Java ME CLDC and Java Card profiles is governed by their respective specifications.

Security and the Virtual Machine

The first popular use of Java was to permit web surfers to download applets into their web browsers. This greatly extended the capabilities of web browsers, but some people were concerned that the capabilities had been extended too far, allowing malicious applet writers to wreak havoc on the computer of anybody using the applets. They pointed out that Java is a full-featured language, which would allow applet writers to do just about anything. It is extremely difficult to prove things about computer programs written in conventional languages, and it is usually easy to hide malicious code in an innocent-looking program.

The Java platform provides a solution to these security fears. The platform contains checks designed to prevent unfriendly behavior by limiting the capabilities of applets, placing potentially harmful operations off-limits. To ensure that these checks cannot be circumvented, the Java virtual machine contains a verification algorithm that eliminates the possibility of behavior that could be used to bypass the security checks.

This chapter examines the relationship between the Java platform's security architecture and the verification algorithm, and it shows how the verification algorithm helps ensure the promises of the Java platform's security policy.

Java Platform and Need for Security

The JVM instruction set has very little ability to interface with your computer system. There are no instructions to read files, connect to sockets, or use system resources other than memory and the CPU. While that's a very safe way to live, it's not a very interesting one, since people who use Java programs want to do things like save their work, print it out, get information over the network, and so on. These capabilities are exposed to programs running on the JVM through a set of application programming interfaces, collectively called the Java platform.

To enable the platform, the JVM is extended through use of native methods, methods written in the native language of the computer instead of in bytecodes. Because they are implemented in native code, native methods have the ability to do just about anything on your computer. Therefore, access to native methods must be controlled, and the controls must not be able to be bypassed.

The verification process allows the JVM to make certain promises about code that is executed.

Security Promises of the JVM

Here are some of the promises the Java virtual machine makes about programs that have passed the verification algorithm:

- Every object is constructed exactly once before it is used.
- Every object is an instance of exactly one class, which does not change through the life of the object.
- If a field or method is marked private, then the only code that ever accesses it is found within the class itself.
- Fields and methods marked protected are used only by code that participates in the implementation of the class.
- Every local variable is initialized before it is used.
- Every field is initialized before it is used.
- It is impossible to underflow or overflow the stack.
- It is impossible to read or write past the end of an array or before the beginning of the array.
- It is impossible to change the length of an array once it has been created.
- Final methods cannot be overridden, and final classes cannot be subclassed.
- Attempts to use a null reference as the receiver of a method invocation or source of a field cause a NullPointerException to be thrown.

The Java platform security architecture depends on all these promises and many more.

Security Architecture and Security Policy

The Java platform builds security architecture on top of the protections promised by the JVM. Security architecture is a way of organizing the software that makes up the Java platform so that potentially harmful operations are isolated from unprivileged code but available to privileged code. Most code is unprivileged; only carefully selected pieces of code are privileged to perform potentially dangerous operations. The security architecture is responsible for making sure that unprivileged code does not masquerade as privileged code.

The core of the Java platform security architecture under Java platforms 1.0 and 1.1 is the SecurityManager class.[1] This class decides which pieces of code can perform certain operations and which cannot. Collectively, these decisions are called the security policy. The security policy is enforced by the Java platform classes, which check the SecurityManager before proceeding with any operations under the control of the SecurityManager.

On the Java 2 platform, the core of the security architecture is shifted to a class called AccessController. See http//java.sun.com for more information.

Only one instance of the SecurityManager can be installed, and once it is installed it cannot be removed. It is called the security manager. By default, there is no security manager, and all operations are permitted. The class java.lang.System is responsible for ensuring that there is only one security manager. It provides the static methods getSecurityManager and setSecurityManager to get and set the security manager.

The SecurityManager class has a set of methods that are called by the Java platform code before proceeding with certain potentially harmful operations. These methods throw a SecurityException if the operation is forbidden. If no exception is thrown, then the caller may assume that the operation is permitted, and it can proceed with the operation. Table 15.1 describes the operations that are checked by the SecurityManager in the Java 1.02 platform.

The security manager uses a variety of factors to determine whether an operation is permitted or not. These factors include the source of the code attempting the operation and the preferences of the user (discussed further in sections 15.3.2 and 15.3.3). First we present an example of how the security architecture and the security policy interact.

Table 15.1. Security checks

Method	**Operation checked**	**Called by**
checkAccept (String host, int port)	Accepting a socket connection from host on port	ServerSocket.accept
checkAccess (Thread g)	Modifying the thread g	Thread.stop Thread.suspend Thread.resume Thread.setPriority Thread.setName Thread.setDaemon
checkAccess (ThreadGroup g)	Modifying the thread group g	ThreadGroup.<init> ThreadGroup.setDaemon ThreadGroup.setMaxPriority ThreadGroup.stop ThreadGroup.resume ThreadGroup.destroy
checkConnect (String host, int port)	Opening a socket to host on port	Socket.connect
checkCreateClassLoader()	Creating a class loader	ClassLoader.<init>

checkDelete(String file)	Deleting a file	File.delete
checkExec(String cmd)	Creating a subprocess	Runtime.exec
checkExit(int status)	Exiting the JVM	Runtime.exit
checkLink(String lib)	Loading a library	Runtime.load Runtime.loadLibrary
checkListen(int port)	Listening at a port	Socket.listen
checkPackageAccess (String package)	Attempting to access package	ClassLoader.loadClass
checkPackageDefinition (String package)	Defining a class in package	ClassLoader.loadClass
checkPropertiesAccess()	Reading or writing properties	System.getProperties System.setProperties
checkPropertyAccess (String property)	Reading the property named property	System.getProperty
checkRead (FileDescriptor fd)	Reading from the file descriptor fd	FileInputStream.<init>
checkRead(String file)	Reading from the file named file	FileInputStream.<init>
checkSetFactory()	Setting a socket factory	ServerSocket SetSocketFactory
checkWrite (FileDescriptor fd)	Writing to a file descriptor	FileOutputStream.<init>
checkWrite(String f)	Writing to a file named file	FileOutputStream.<init>

15.3.1 Example

The security architecture demands that before a file is opened for reading, the security manager will be checked to see whether the code requesting the file is allowed to open the file. This is a sensible policy, since the ability to read files gives a program access to potentially important and confidential information.

The most important ways to read files are through the FileInputStream or FileReader classes in the java.io package. These classes include native methods that are implemented in a machine-dependent way to interact with the computer's file system. Before these methods can be used, an instance of FileInputStream or FileReader must be constructed. Following the security architecture, all the publicly accessible constructors of FileInputStream and FileReader begin with a call to the checkRead method of SecurityManager:

```
/** Open the file named filename */
public FileInputStream(String filename) throws
    FileNotFoundException
{
  SecurityManager security = System.getSecurityManager();
  // If a security manager is present, check it.
  if (security != null)
     security.checkRead(name);
  // Open the file named filename; code omitted
}
```

The system's security manager is found by calling System.getSecurityManager(). There may not be a security manager installed, in which case the FileInputStream is free to open the file. If the security manager has not been established, then any code may set it. Secure systems, like web browsers, install a security manager before any applets are loaded.

If a security manager is present, then its checkRead method determines whether or not the code that called the constructor is allowed to read the file. How it decides this is a matter of the security policy of the particular system. If it decides that the read is not allowed, then a SecurityException is thrown.

This is where the JVM's promises become important. If an exception is thrown from checkRead, it will not be caught in the body of the constructor, causing the constructor to terminate abnormally. Since the constructor did not complete normally, the FileInputStream object is not constructed. If the code attempts to use the object before it has been constructed, then the verification algorithm rejects the program.

If the checkRead method returns normally, then the FileInputStream constructor also returns normally. Now that the FileInputStream object is constructed, all the public methods of FileInputStream, including read, may be invoked on it. The methods don't need to check for permission again, since the existence of the object implies that permission was granted.

Basic Applet Security Policy

For the writers of web browsers, security is of tremendous concern, since a malicious or mistaken applet could cause problems for many, many web surfers. For this reason, the Java applet security policy is very restrictive. The security manager discriminates between two kinds of classes: system classes (loaded by the JVM automatically) and applets (loaded by a custom class loader from over the network). The web browser implements its own security manager, which subclasses SecurityManager. Let's call it the AppletSecurityManager.

The AppletSecurityManager can ascertain the source of the applet by looking at the applet's class loader. Each class maintains a permanent association with its class loader. Anybody can find out which class loader loaded the class by invoking getClassLoader, a method of Class. Since Class is final, the verification algorithm ensures that the method getClassLoader cannot be overridden, so it is not possible for a class to lie about its class loader. The class loader can track information about the source of the applet.

The security manager needs to know which class is invoking the operation. For example, suppose an applet called NastyApplet tries to create a FileInputStream so that it can read the system's password file. In order to do that, it must call the constructor for FileInputStream, or the verification algorithm will reject the applet before it gets a chance to run. As shown in section 15.3.1, the constructor for FileInputStream calls checkRead in the security manager. The Java stack now looks like this:

Method	**Class**	**Class loader**
checkread	AppletSecurityManager	none
<init>	FileInputStream	none
attack	NastyApplet	AppletClassLoader
run	NastyApplet	AppletClassLoader

The SecurityManager provides the method currentClassLoader, which looks down the execution stack for the first method that comes from a class that was loaded by a class loader. Since AppletSecurityManager and FileInputStream are part of the browser itself, they are loaded without any class loader. The SecurityManager looks down the stack until it finds the run method from NastyApplet, which was loaded by AppletClassLoader.

The policy of the web browser is that no applet may try to read a file. The web browser's security policy is implemented by the class AppletSecurityManager. Part of the definition of AppletSecurityManager is

```
class AppletSecurityManager extends SecurityManager
{
  public void checkRead(String name) throws SecurityException
  {
    ClassLoader loader = currentClassLoader();
    if(loader instanceof AppletClassLoader)
      throw new SecurityException("file.read");
  }
}
```

Since the class loader is an AppletClassLoader, a security exception is thrown. It is not caught in the constructor for FileInputStream, so that method does not terminate normally. The FileInputStream is therefore invalid and may not be used to read any files. Security has been preserved.

More Sophisticated Policies

The security policy of the basic class loader, while effective at preventing security problems, is too tight for much work to get done, since it's not possible for an applet to save work or read previously written files.

A more sophisticated policy might allow applets to read and write files in a certain sandbox area: all files beginning with a certain absolute path would be permitted, all others denied. In some browsers, the system properties acl.read and acl.read.default control the sandbox area. If the user had configured acl.read as /sandbox, then a word processing applet would be permitted to read /sandbox/Letter.doc but the NastyApplet would not be allowed to read /etc/passwd.

Even this seemingly safe idea has some potential problems. The NastyApplet might try to fool the system by opening /sandbox/../etc/passwd, which names the password file on some systems. This is easily prevented by checking for relative references like "..", but it gives some idea of the sort of thing security developers must consider. This sort of potential security problem is beyond the domain of the JVM itself.

Another kind of security policy allows some code more trust than others. An applet programmer may cryptographically sign the bytecodes that make up the applet. The cryptographic signature makes it possible to determine whether the applet has been altered after it has been signed, since the signature is made from the original bytecodes that have been encrypted with the programmer's private key. Use of the programmer's public key allows the applet user to ascertain the source of the signature. Since the programmer's private key should be truly private, the signature uniquely identifies that programmer as the source of the code.

To make these more sophisticated security policies possible, the standard Java platform version 1.1 and later contain code in the java.security package to provide the code necessary for doing encryption, decryption, and signature verification. The classes in this package were designed to prevent applets from interfering with them, and they depend on the promises of the JVM to ensure that the cryptographic routines are not tampered with.

4. Some Potential Attacks

JVM security is often discussed in terms of the Java language. The Java compiler rejects programs that attempt some kinds of attacks, such as reading private variables or casting an object reference to an invalid type. However, some attackers may try to bypass the compiler errors by writing in Oolong instead of Java or by writing bytecodes directly. In this section, we show how the Java virtual machine thwarts some potential attacks on system security.

4.1 Implementation Details

Examples in this section frequently involve private methods and fields of standard classes. Private methods and fields are not necessarily consistent from JVM implementation to JVM implementation; the private keyword denotes parts of the code that are implementation dependent.

Security must not depend on the fact that the attacker does not know the implementation details of the system. An attacker may guess what private methods are being used, since the attacker can easily obtain a copy of the same virtual machine implementation the victim is using. If the attack is through an applet, then the attacker can reasonably assume that the victim is using one of the popular Java-supporting web browsers and may use attacks using private implementation details of these browsers.

Fortunately, the JVM provides a strong base for making the implementation secure, even if the attacker knows how the system was implemented. This section shows how some potential attacks are thwarted, either by the verification algorithm or by the runtime system, which depends on the verifier.

4.2 Protecting the Security Manager

The system has exactly one security manager, which is kept by the java.lang.System class. It is retrieved with getSecurityManager, a static method in java. lang.System.

Initially, there is no security manager, and getSecurityManager returns null. The security manager can be set with setSecurityManager:

```
AppletSecurityManager asm = new AppletSecurityManager();
System.setSecurityManager(asm);
```

The first time setSecurityManager is called, it sets the security manager. After that, all calls to setSecurityManager will throw a SecurityException without setting the manager. This prevents anybody else, such as a malicious applet, from altering the security manager. It is the job of the browser to set the security manager before loading any applets.

Each JVM implementation can implement the System class differently. In subsequent sections, we assume that it stores the security manager in a private field called security:

```
package java.lang;
public final class System
{
  private static SecurityManager security;

  public static SecurityManager getSecurityManager()
  {
    return security;
  }

  public static setSecurityManager(SecurityManager sm)
  {
    if(security == null)
      throw new SecurityException
        ("SecurityManager already set");
    security = sm;
  }
}
```

The integrity of the security field is critical, since an applet that could set that field could control the security policy of the system. This makes security a likely target for attack.

4.3 Bypassing Java Security

The most straightforward attack would be to try to set the security field. The attacker's applet might be written in Java:

```
public class NastyApplet extends Applet
{
  void attack()
  {
    System.security = null;
    // The security manager is null, so everything is
    // permitted
    // Put code to wreak havoc here
  }
}
```

When this class is compiled, the Java compiler will notice that the security field is private to the System class and refuse to compile the file. This would not stop a determined attacker, who would proceed to rewrite the applet in Oolong:

```
.class public NastyApplet
.super java/applet/Applet

.method attack()V
aconst_null              ; Install null as the
                         ; security manager
putstatic java/lang/System/security Ljava/lang/SecurityManager;
;; Wreak havoc
.end method
```

This is a valid Oolong program. Once you have downloaded the applet into your web browser, the attacker's applet will try to use the attack method to cause trouble.

The JVM prevents this attack from succeeding by running the verification algorithm on the class when it is loaded. All the information needed about what fields are accessed is present in the code. That is, it isn't necessary to run the applet to find out which fields it may try to access. The verification algorithm finds that the security field is private, so this code cannot access it. The verification algorithm causes the system to refuse to load this class, thwarting the attack before the applet starts to run.

The following method is perfectly safe because the potentially dangerous code can never actually be executed:

```
.method attack()V
return
; Just kidding. There's no way to execute the next line, since
; the method always returns before it gets here
aconst_null              ; Install null as the
                         ; security manager
```

```
putstatic java/lang/System/security Ljava/lang/SecurityManager;
return
.end method
```

The verification algorithm rejects this applet just because of the presence of this security-violating instruction, even though the dangerous code is unreachable.

There is no way to bypass the verification step, since it occurs automatically when the resolveClass method is invoked in the ClassLoader that loads the class. The class is unusable until it has been resolved. When it is resolved, the verification algorithm catches the attack and refuses to load the class. If it is never loaded, then it can never be executed, so the attack is avoided.

4.4 Using Unconstructed Objects

Some classes depend on their constructors to control who may create instances. For example, the setSecurityManager method in java.lang.System is public, which permits anybody to set the security manager.

An applet might try to install its own security manager:

```
class WimpySecurityManager extends SecurityManager
{
  // Override the methods so that they don't do anything
}
class MaliciousApplet extends Applet
{
  SecurityManager mySecurityManager = new
    WimpySecurityManager();
  System.setSecurityManager(mySecurityManager);
  // Now I can do anything I want!
}
```

This attack won't work, even though setSecurityManager is public. The reason is that you're not allowed to create your own SecurityManager once a security manager has been installed. The SecurityManager's constructor has code to check if there already is a security manager installed:

```
public class SecurityManager
{
  protected SecurityManager()
  {
    if (System.getSecurityManager() != null)
       throw new SecurityException(
          "security manager already installed.");
    // Continue constructing the security manager
  }
}
```

The verification algorithm requires that each constructor call a superclass constructor. Since this is the only constructor for SecurityManager, the WimpySecurityManager must call it. In the example, the Java compiler creates a default constructor for WimpySecurityManager, which calls the constructor for SecurityManager. That constructor finds that a SecurityManager has already been created and throws a SecurityException. Since the system won't permit use of an unconstructed object, the attacker can never install the WimpySecurityManager as the security manager of the system.

Suppose, however, that the attacker wrote a constructor for WimpySecurityManager that didn't call the constructor:

```
.class WimpySecurityManager
.super java/lang/SecurityManager

.method public <init>()V
return              ; Don't call the superclass constructor!
.end method
```

The JVM verification algorithm rejects WimpySecurityManager, since one of the things the verification algorithm checks is that each constructor calls a superclass constructor somewhere in the body of the constructor.

The attacker might try to fool the verification algorithm by hiding the call:

```
.method public <init>()V
goto end                ; Skip over the superclass constructor
aload_0
invokespecial java/lang/SecurityManager/<init> ()V
end:
return
.end method
```

The verification algorithm discovers that the superclass <init> method is not always called, and it rejects this class. Another attempt might involve hiding the call behind an if instruction:

```
.method public <init>(I)V
.limit locals 2
iload_1
ifeq end            ; Skip the superclass constructor if
aload_0             ; the argument is nonzero
invokespecial java/lang/SecurityManager/<init> ()V
end:
return
.end method
```

This constructor calls the superclass constructor if the argument is nonzero; otherwise, it skips it. However, any path through the method must call the superclass constructor, or the verification algorithm rejects it. The verification algorithm does not have to run the code to determine whether the constructor will be called. All the verification algorithm has to do is to prove that there is a way through the method that does not involve the superclass constructor call, to provide sufficient grounds to reject the entire class.

This must be true of all constructors. Even if you provide a valid constructor as well as an invalid one, the verification algorithm still rejects the class. The goal is that the verification algorithm must be able to prove that the superclass constructor is called no matter which constructor is called and no matter which arguments are provided.

4.5 Invalid Casts

An attacker would gain a lot of information if the JVM permitted invalid casts, especially with reference types. An attacker may assume, often correctly, that a reference is represented internally by a pointer into memory. The attacker may be able to guess how the object is stored. If the system could be fooled into thinking that the object had a different class, it might permit a program to read or write fields on an object where access would otherwise be denied.

For example, a web browser might keep a copy of the user's private key for signing messages. The Message class is public to permit programs to sign messages, but the private key should not be revealed, even to those programs allowed to do the signing.

```
public class Message
{
  private PrivateKey private_key;

  /** Sign the data, but don't reveal the key
  public byte[] sign(byte[] data);
}
```

An attacker might try to get the private key by creating a class like Message but with different permissions:

```
public class SneakyMessage
{
  public PrivateKey private_key;
}
```

This class is likely to have the same layout in memory as Message. If the attacker could cast a Message into a SneakyMessage, then the value of private_key would appear to be public.

Fortunately, this can't happen. This attack, written in Java, might look like this:

```
void attack(Message m)
{
  SneakyMessage hack = (SneakyMessage) m;
  // Read the private_key field from hack
}
```

The Java compiler rejects this class, pointing out that the cast from Message to SneakyMessage will never succeed. The attacker might try to bypass the compiler like this:

```
.class SneakyApplet
.super java/applet/Applet

.method attack(LMessage;)V
aload_1             ; Variable 1 contains a Message
                ; Try to get the private key out
getfield SneakyMessage/private_key Ljava/security/PrivateKey;
;; Use the private key
.end method
```

The verification algorithm rejects this code, even though the private_key in SneakyMessage is public, since it knows that a SneakyMessage is not a Message and vice versa. Since variable 1 contains a Message, not a SneakyMessage, the getfield instruction is invalid.

Some people are surprised that the verification algorithm makes this determination, since it is not always possible to prove that variable 1 contains a Message. This is possible because it is not the verification algorithm's job to prove that variable 1 contains a nonsneaky Message before rejecting this class. Rather, the verification algorithm tries to prove that variable 1 does contain a SneakyMessage in order to accept the getfield instruction. This is not true, since variable 1 is initialized to a Message and the variable is never altered, so the verification algorithm rejects the code.

It is easy to create code leading up to the getfield where it is impossible to be sure what the top of the stack is without actually running the code:

```
.method attack(LMessage;LSneakyMessage;I)
iload_3             ; Push the number
ifeq attack          ; If it is 0, then try the attack
   aload_2          ; Push the sneaky message
   goto continue      ; Continue with the attack
attack:
   aload_1          ; Push the message, which we try to
continue:           ; treat as a sneaky message
getfield SneakyMessage/private_key Ljava/security/PrivateKey;
```

At the last instruction, the value on top of the stack may or may not be SneakyMessage, depending on the value of the third argument to the method. The verification algorithm will unify two stack pictures at continue: One with a SneakMessage and one with a Message. The unification is Object. This makes the getfield instruction invalid, so the class is rejected.

4.6 Changing the Class of a Reference

The checkcast instruction changes the verification algorithm's perception of the class of an object. The attack method could be rewritten using checkcast:

```
.method attack(LMessage;)V
aload_1                 ; Variable 1 contains a Message
checkcast SneakyMessage    ; Ask the verifier to believe it is
                        ; a SneakyMessage
                        ; Try to get the private key out
getfield SneakyMessage/private_key Ljava/security/PrivateKey;
;; Use the private key
.end method
```

The verification algorithm approves this code, and the applet gets a chance to run. However, the attack is still ineffective, because the checkcast instruction checks the actual type of the value on top of the stack when the program is running. Since the value on top of the stack is really a Message, and a Message is not a SneakyMessage, the checkcast instruction throws an exception. The exception transfers control away from the getfield instruction, so the information is not compromised.

The checkcast instruction does not affect the underlying object. It only changes the verification algorithm's perception of the object as the class is being loaded. The verification algorithm attempts to prove that the program is safe. The checkcast instruction tells the verification algorithm, "You cannot prove that this is safe, but if you check it out as the program runs, you will find that the cast will succeed."

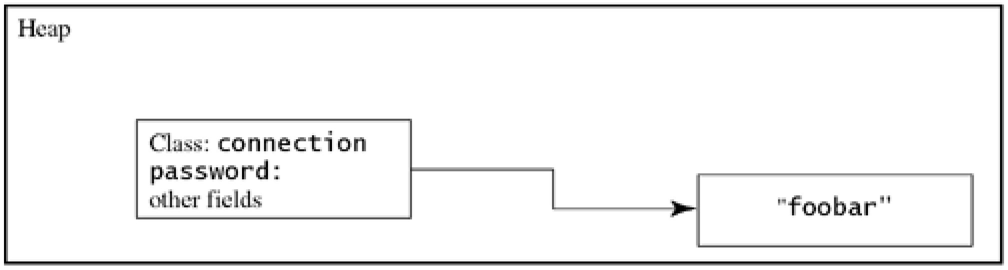
As the program runs, each time checkcast is encountered the JVM checks to ensure that the class of the argument is really a SneakyMessage. If it fails, the JVM throws a ClassCastException. Because no Message can be a SneakyMessage, this code always causes the exception to be thrown.

There is one way for the code to get past the checkcast at runtime. If the Message is null, then the checkcast allows the program to proceed. However, this doesn't help the attacker, since any attempt to read fields from a null reference will be met with a NullPointerException.

4.7 Reading Uninitialized Fields

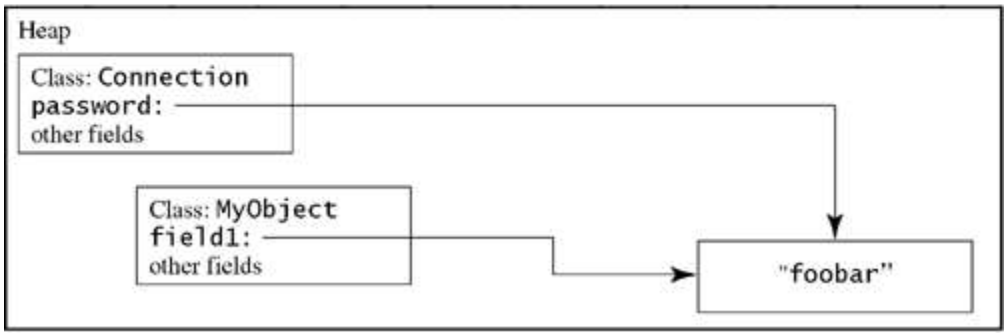
When an object is first created, it may be assigned a place in memory where something important used to be. For example, the web browser may have the user's password to some web site stored in a Connection object. The Connection object and the password string live somewhere in the system's memory, as in Figure 1.

Figure 1. A Connection object points to your password



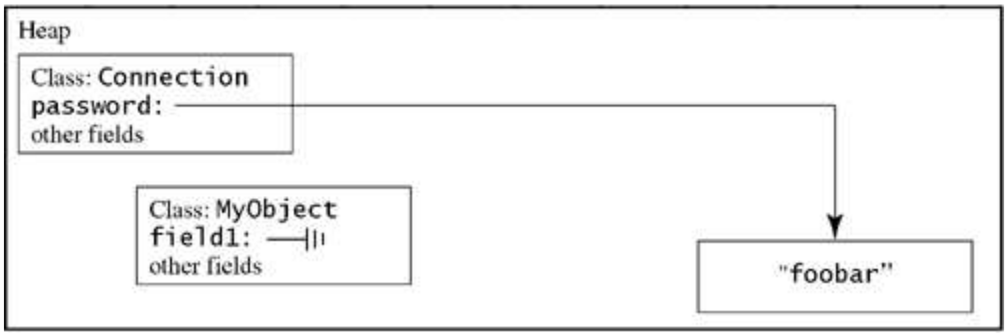
Later, the garbage collector may move the Connection object. Things don't really move in a computer's memory; instead, they are copied to a new location and the old location is forgotten. Suppose a user creates a new object that is assigned to the memory space that the Connection object used to occupy. An attacker might hope that the memory looks like Figure 2. The first field in MyObject, field1, happens to fall in the same place as the password field of the Connection used to. Before field1 is initialized, it appears to contain a reference to the value that used to be the password field of the Connection object. Although it is unlikely that this would happen, it is not impossible, and the applet code could use this information in a malicious fashion.

Figure 2. Connection moved, and a new object in the same memory



Fortunately, the virtual machine prevents this attack by requiring that all fields be initialized before they are read. If the field does not have an explicit initializer, it is implicitly initialized to 0 for numeric fields or null for reference fields. This happens immediately after the object is created, before it can be used. Therefore, in reality the picture looks like Figure 3. Even though the new object occupies the same memory space as the Connection used to, it is unable to access the values that used to be there because field1 is initialized to null before the code has a chance to read the old value.

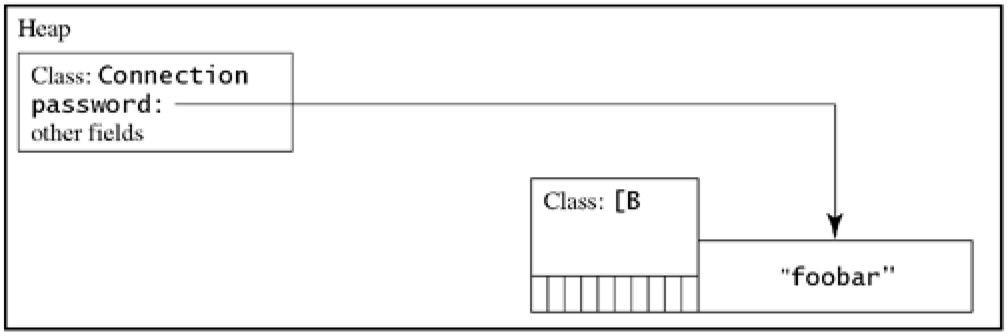
Figure 3. field1 is initialized immediately



15.4.8 Array Bounds Checks

Another kind of memory-based attack is to try to read past the ends of an array. To extend an earlier example, suppose that the user created an array of ten bytes that just happened to be placed next to an important place in memory (Figure 4). The first ten elements of the array are safe to be read and written. The location that would hold the eleventh byte, however, is the beginning of the password string. If an applet were to read beyond the end of the array, the password would be revealed.

Figure 4. Password object in memory immediately after an array



The only way to read those bytes is to use the baload instruction:

```
bipush 10
newarray byte            ; Create an array of ten bytes
bipush 10               ; Try to read the eleventh byte
baload
```

Both the size of the array and the array element to be accessed are determined by operands on the operand stack, not by instruction arguments. This means that there is no way for the virtual machine to be certain what these values will be without running the program.

Therefore, this sort of error cannot be caught by the verification algorithm. Instead, this attack is stopped by the baload instruction. Each time the instruction is executed, the length of the array is checked to ensure that the element does not fall out of bounds. If it does, an ArrayIndexOutOfBoundsException is thrown.

The bounds check operation is part of the baload instruction and all the other array load instructions. The virtual machine does not depend on the program to check array bounds before accessing elements. This removes some responsibility from the programmer, while still ensuring that array bounds are respected.

4.9 Catching Exceptions

An attacker might try to catch exceptions that are intended to prevent harmful things from happening to the system. The attack would try to ignore the exceptions.

For example, it was shown earlier that the SecurityManager initializer does not permit a second SecurityManager to be created once one has been installed. If the initializer detects the existence of another security manager, it throws a SecurityException, which prevents the new SecurityManager from being used.

Suppose that the attacker catches the exception from the SecurityManager constructor within the constructor of WimpySecurityManager:

```
.class WimpySecurityManager

.method <init>()V
.catch java/lang/SecurityException from begin to end using handler
begin:
   aload_0          ; Call the superclass constructor
   invokevirtual java/lang/SecurityManager/<init>()V
end:
   return           ; If I'm allowed to invoke the
                  ; constructor, then something is wrong
   handler:         ; Since a SecurityException is thrown,
                  ; control continues here
      return        ; See if I can return with the object
                  ; only partially constructed
.end method
```

The JVM refuses to load this code because it fails to pass the verification algorithm. One of the verification algorithm's goals in tracing through a constructor is that every return is on a path through a call to a superclass constructor. Because all the lines between begin and end are covered by an exception handler, any of them might throw an exception.

This means that there is a path through the method that does not invoke the constructor. The verification algorithm judges the method to be invalid, which means that the entire class is invalid. This class is rejected, and the system is safe.

15.4.10 Hidden Code

An attacker might try to hide bytecodes inside other bytecodes. Consider an innocuous-looking piece of Oolong code like this:

```
sipush -19712
ineg
```

This code could actually contain a potential attack hidden in the bytecodes. When assembled, the above code produces the following bytecodes:

```
Location     Value     Meaning
0050         11        sipush
0051         b3            0xb300=-19712
0052         00
0053         74        ineg
```

An attacker could generate a class file in which the next bytes were

```
0054         a7        goto
0055         ff            0xfffd=-3
0056         fd
```

The goto points to location 51, which is in the middle of the sipush instruction. It is not possible to write this code in Oolong, since the Oolong assembler will permit a label only between instructions, not in the middle of an instruction.

The attacker hopes that when the code reaches the instruction at location 54, it will attempt to go back three bytes to the byte at 51. If we interpret the bytes at location 51 as bytecodes, we get

```
0051         b3        putstatic
0052         00                0x0074=116
0054         74
```

This is an interpretation of the bytes' attempts to read the field contained in the constant 116.

The attacker might make the constant 116 a Fieldref to a private field like security in java.lang.System, hoping to wipe out the security manager for the system. The attacker hopes that the JVM will not attempt to verify the bytecodes interpreted in this fashion. The attacker hopes that the system will assume that it has checked the code statically, and won't try to check whether or not the field is private at runtime.

There is nothing wrong with the bytes at locations 51 through 53, as long as they are interpreted properly. The problem is the code at 54, which tries to jump into the middle of an instruction.

The JVM discovers that the code is attempting to branch into the middle of an instruction, and it will reject the class.

Of course, not all code like this is necessarily harmful. This code may just come from an overly clever programmer who attempted to shrink the code by reinterpreting it this way, and the code may be perfectly safe to execute. However, the verification algorithm never promised that it would accept all safe programs. It only promises to reject all that do not meet certain criteria, which includes all unsafe programs. Because this program does not follow the rules, it is rejected.

Security and the class loader architecture

A look at the role played by class loaders in the JVM's overall security model

A sandbox refresher

Java's security model is focused on protecting end-users from hostile programs downloaded from untrusted sources across a network. To accomplish this goal, Java provides a customizable "sandbox" in which Java programs run. A Java program can do anything within the boundaries of its sandbox, but it can't take any action outside those boundaries. The sandbox for untrusted Java applets, for example, prohibits many activities, including:

- reading or writing to the local disk
- making a network connection to any host, except the host from which the applet came
- creating a new process
- loading a new dynamic library and directly calling a native method

By making it impossible for downloaded code to perform certain actions, Java's security model protects users from the threat of hostile code. For more information on the sandbox concept, see last month's "Under the Hood."

The class loader architecture

One aspect of the JVM that plays an important role in the security sandbox is the class loader architecture. In the JVM, class loaders are responsible for importing binary data that defines the running program's classes and interfaces. In the block diagram shown in Figure 1, a single mysterious cube identifies itself as "the class loader," but in reality, there may be more than one class loader inside a JVM. Thus, the class loader cube of the block diagram actually represents a subsystem that may involve many class loaders. The JVM has a flexible class loader architecture that allows a Java application to load classes in custom ways.

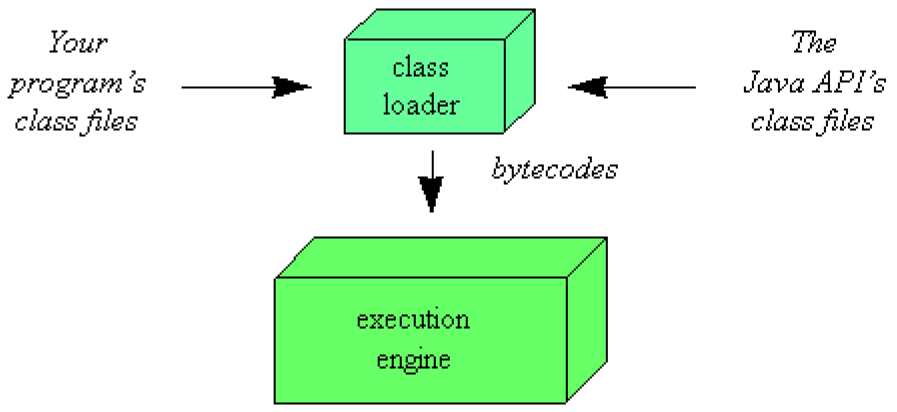


Figure 1. Java's class loader architecture

A Java application can use two types of class loaders: a "primordial" class loader and class loader objects. The primordial class loader (there is only one of them) is a part of the JVM implementation. For example, if a JVM is implemented as a C program on top of an existing operating system, then the primordial class loader will be part of that C program. The primordial class loader loads trusted classes, including the classes of the Java API, usually from the local disk.

At run time, a Java application can install class loader objects that load classes in custom ways, such as by downloading class files across a network. The JVM considers any class it loads through the primordial class loader to be trusted, regardless of whether or not the class is part of the Java API. It views with suspicion, however, those classes it loads through class loader objects. By default, it considers them to be untrusted. While the primordial class loader is an intrinsic part of the virtual machine implementation, class loader objects are not. Instead, class loader objects are written in Java, compiled into class files, loaded into the virtual machine, and instantiated just like any other object. They really are just another part of the executable code of a running Java application. You can see a graphical depiction of this architecture in Figure 2.

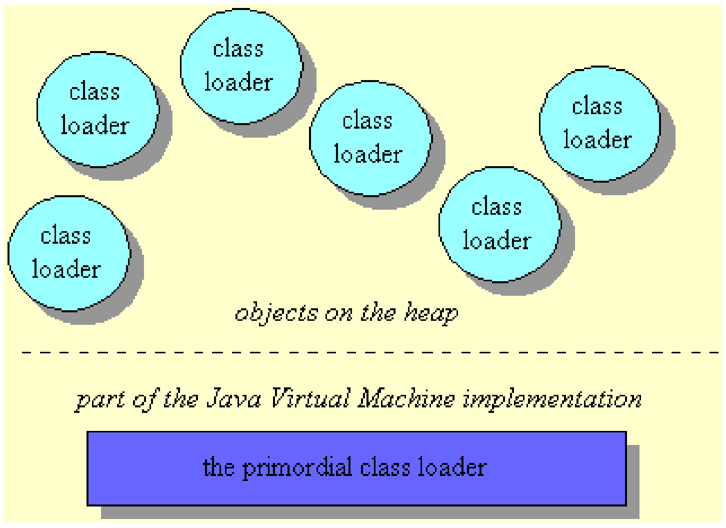


Figure 2. Java's class loader architecture

Because of class loader objects, you don't have to know at compile-time all the classes that may ultimately take part in a running Java application. They enable you to *dynamically extend* a Java application at run time. As it runs, your application can determine what extra classes it needs and load them through one or more class loader objects. Because you write the class loader in Java, you can load classes in any manner: You can download them across a network, get them out of some kind of database, or even calculate them on the fly.

Class loaders and name-spaces

For each class it loads, the JVM keeps track of which class loader -- whether primordial or object -- loaded the class. When a loaded class first refers to another class, the virtual machine requests the referenced class from the same class loader that originally loaded the referencing class. For example, if the virtual machine loads class Volcano through a particular class loader, it will attempt to load any classes Volcano refers to through the same class loader. If Volcano refers to a class named Lava, perhaps by invoking a method in class Lava, the virtual machine will request Lava from the class loader that loaded Volcano. The Lava class returned by the class loader is dynamically linked with class Volcano.

Because the JVM takes this approach to loading classes, classes can by default only see other classes that were loaded by the same class loader. In this way, Java's architecture enables you to create multiple *name-spaces* inside a single Java application. A name-space is a set of unique names of the classes loaded by a particular class loader. For each class loader, the JVM maintains a name-space, which is populated by the names of all the classes that have been loaded through that class loader.

Once a JVM has loaded a class named Volcano into a particular name-space, for example, it is impossible to load a different class named Volcano into that same name-space. You can load multiple Volcano classes into a JVM, however, because you can create multiple name-spaces inside a Java application. You can do so simply by creating multiple class loaders. If you create three separate name-spaces (one for each of the three class loaders) in a running Java application, then, by loading one Volcano class into each name-space, your program could load three different Volcano classes into your application.

A Java application can instantiate multiple class loader objects either from the same class or from multiple classes. It can, therefore, create as many (and as many different kinds of) class loader objects as it needs. Classes loaded by different class loaders are in different name-spaces and cannot gain access to each other unless the application explicitly allows it. When you write a Java application, you can segregate classes loaded from different sources into different name spaces. In this way, you can use Java's class loader architecture to control any interaction between code loaded from different sources. You can prevent hostile code from gaining access to and subverting friendly code.

Class loaders for applets

One example of dynamic extension with class loaders is the Web browser, which uses class loader objects to download the class files for an applet across a network. A Web browser fires off a Java application that installs a class loader object -- usually called an *applet class loader* -- that knows how to request class files from an HTTP server. Applets are an example of dynamic extension, because when the Java application starts, it doesn't know which class files the browser will ask it to download across the network. The class files to download are determined at run time, as the browser encounters pages that contain Java applets.

The Java application started by the Web browser usually creates a different applet class loader object for each location on the network from which it retrieves class files. As a result, class files from different sources are loaded by different class loader objects. This places them into different name-spaces inside the host Java application. Because the class files for applets from different sources are placed in separate name-spaces, the code of a malicious applet is restricted from interfering directly with class files downloaded from any other source.

Cooperation between class loaders

Often, a class loader object relies on other class loaders -- at the very least, upon the primordial class loader -- to help it fulfill some of the class load requests that come its way. For example, imagine you write a Java application that installs a class loader whose particular manner of loading class files is achieved by downloading them across a network. Assume that during the course of running the Java application, a request is made of your class loader to load a class named Volcano.

One way you could write the class loader is to have it first ask the primordial class loader to find and load the class from its trusted repository. In this case, since Volcano is not a part of the Java API, assume the primordial class loader can't find a class named Volcano. When the primordial class loader responds that it can't load the class, your class loader could then attempt to load the Volcano class in its custom manner, by downloading it across the network. Assuming your class loader was able to download class Volcano, that Volcano class could then play a role in the application's future course of execution.

To continue with the same example, assume that some time later a method of class Volcano is invoked for the first time, and that the method references class String from the Java API. Because it is the first time the reference is used by the running program, the virtual machine asks your class loader (the one that loaded Volcano) to load String. As before, your class loader first passes the request to the primordial class loader, but in this case, the primordial class loader is able to return a String class back to your class loader.

The primordial class loader most likely didn't have to actually load String at this point because, given that String is such a fundamental class in Java programs, it was almost certainly used before and therefore already loaded. Most likely, the primordial class loader just returned the String class that it had previously loaded from the trusted repository.

Since the primordial class loader was able to find the class, your class loader doesn't attempt to download it across the network; it merely passes to the virtual machine the String class returned by the primordial class loader. From that point forward, the virtual machine uses that String class whenever class Volcano references a class named String.

Class loaders in the sandbox

In Java's sandbox, the class loader architecture is the first line of defense against malicious code. It is the class loader, after all, that brings code into the JVM -- code that could be hostile.

The class loader architecture contributes to Java's sandbox in two ways:

1. It prevents malicious code from interfering with benevolent code.
2. It guards the borders of the trusted class libraries.

The class loader architecture guards the borders of the trusted class libraries by making sure untrusted classes can't pretend to be trusted. If a malicious class could successfully trick the JVM into believing it was a trusted class from the Java API, that malicious class potentially could break through the sandbox barrier. By preventing untrusted classes from impersonating trusted classes, the class loader architecture blocks one potential approach to compromising the security of the Java runtime.

Name-spaces and shields

The class loader architecture prevents malicious code from interfering with benevolent code by providing protected name-spaces for classes loaded by different class loaders. As mentioned above, *name-space* is a set of unique names for loaded classes that is maintained by the JVM.

Name-spaces contribute to security because you can, in effect, place a shield between classes loaded into different name-spaces. Inside the JVM, classes in the same name-space can interact with one another directly. Classes in different name-spaces, however, can't even detect each other's presence unless you explicitly provide a mechanism that allows the classes to interact. If a malicious class, once loaded, had guaranteed access to every other class currently loaded by the virtual machine, that class potentially could learn things it shouldn't know, or it could interfere with the proper execution of your program.

Creating a secure environment

When you write an application that uses class loaders, you create an environment in which the dynamically loaded code runs. If you want the environment to be free of security holes, you must follow certain rules when you write your application and class loaders. In general, you will want to write your application so that malicious code will be shielded from benevolent code. Also, you will want to write class loaders such that they protect the borders of trusted class libraries, such as those of the Java API.

Name-spaces and code sources

To get the security benefits offered by name-spaces, you need to make sure you load classes from different sources through different class loaders. This is the scheme, described above, used by Java-enabled Web browsers. The Java application fired off by a Web browser usually creates a different applet class loader object for each source of classes it downloads across the network. For example, a browser would use one class loader object to download classes from http://www.niceapplets.com, and another class loader object to download classes from http://www.meanapplets.com.

Guarding restricted packages

Java allows classes in the same package to grant each other special access privileges that aren't granted to classes outside the package. So, if your class loader receives a request to load a class that by its name brazenly declares itself to be part of the Java API (for example, a class named java.lang.Virus), your class loader should proceed cautiously. If loaded, such a class could gain special access to the trusted classes of java.lang and could possibly use that special access for devious purposes.

Consequently, you would normally write a class loader so that it simply refuses to load any class that claims to be part of the Java API (or any other trusted runtime library) but that doesn't exist in the local trusted repository. In other words, after your class loader passes a request to the primordial class loader, and the primordial class loader indicates it can't load the class, your class loader should check to make sure the class doesn't declare itself to be a member of a trusted package. If it does, your class loader, instead of trying to download the class across the network, should throw a security exception.

Guarding forbidden packages

In addition, you may have installed some packages in the trusted repository that contain classes you want your application to be able to load through the primordial class loader, but that you don't want to be accessible to classes loaded through your class loader. For example, assume you have created a package named absolutepower and installed it on the local repository accessible by the primordial class loader. Assume also that you don't want classes loaded by your class loader to be able to load any class from the absolutepower package. In this case, you would write your class loader such that the very first thing it does is to make sure the requested class doesn't declare itself as a member of the absolutepower package. If such a class is requested, your class loader, rather than passing the class name to the primordial class loader, should throw a security exception.

The only way a class loader can know whether or not a class is from a restricted package, such as java.lang, or a forbidden package, such as absolutepower, is by the name of the class. Thus, a class loader must be given a list of the names of restricted and forbidden packages. Because the name of class java.lang.Virus indicates it is from the java.lang package, and java.lang is on the list of restricted packages, your class loader should throw a security exception if the primordial class loader can't load it. Likewise, because the name of class absolutepower.FancyClassLoader indicates it is part of the absolutepower package, and the absolutepower package is on the list of forbidden packages, your class loader should throw a security exception.

A security-minded class loader

A common way to write a security-minded class loader is to use the following four steps:

1. If packages exist that this class loader is not allowed to load from, the class loader checks whether the requested class is in one of those forbidden packages mentioned above. If so, it throws a security exception. If not, it continues on to step two.
2. The class loader passes the request to the primordial class loader. If the primordial class loader successfully returns the class, the class loader returns that same class. Otherwise it continues on to step three.
3. If trusted packages exist that this class loader is not allowed to add classes to, the class loader checks whether the requested class is in one of those restricted packages. If so, it throws a security exception. If not, it continues on to step four.
4. Finally, the class loader attempts to load the class in the custom way, such as by downloading it across a network. If successful, it returns the class. If unsuccessful, it throws a "no class definition found" error.

By performing steps one and three as outlined above, the class loader guards the borders of the trusted packages. With step one, it prevents a class from a forbidden package to be loaded at all. With step three, it doesn't allow an untrusted class to insert itself into a trusted package.

Conclusion

The class loader architecture contributes to the JVM's security model in two ways:

1. by separating code into multiple name-spaces and placing a "shield" between code in different name-spaces
2. by guarding the borders of trusted class libraries, such as the Java API

Both of these capabilities of Java's class loader architecture must be used properly by programmers so as to reap the security benefit they offer. To take advantage of the name-space shield, code from different sources should be loaded through different class loader objects. To take advantage of trusted package border guarding, class loaders must be written so they check the names of requested classes against a list of restricted and forbidden packages.

Conclusion

We have seen how the Java platform security architecture is based on the promises made by the Java virtual machine. Both simple and complex security policies can be implemented, depending on the degree of sophistication required and the time and effort the policy implementor wants to exert to ensure that the policy is consistent with good practice. It is critical that the implementor understand the guarantees made by the JVM to understand how the security measures might be circumvented.