JAVA ACCESS MODIFIERS IN PARALLEL UNIVERSES

Amr Sabry   Stephen Fickas
Department of Computer Science
University of Oregon
Eugene, OR 97403

Technical Report CIS-TR-98-03

Abstract

This short note describes a gap in Java's access control mechanism that sometimes allows a Java class to illegally find the values of private variables in other classes.

This research was partially sponsored by the Advanced Resarch Projects Agency under the title: Quality of Service Dynamic Validation Qualifiers – The ASSERT System, ARPA order number F269, under contract number N66001-97-C-8521.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. government.

First Edition: July 6, 1998

1 Introduction

We assume the reader is familiar with the Java programming language and its security model. For more details about the language, we recommend the Java Language Specification [1] and for the latest update on security issues, we recommend http://www.javasoft.-com/security.

The problem we report is that, for every implementation of the Java Virtual Machine (JVM), we can write the following method:

```
public void evilMethod (Object x) {
  // we use pure Java code to SOMETIMES determine the values
  // of the private variables of the argument x.
}
```

The trick is to use two JVM implementations that will run in parallel. The first JVM is the one running the main computation: it is invoked (and trusted) by the user. The second JVM is invoked from within `evilMethod`: it is written completely in Java and the user is unaware of its existence. To distinguish between the two JVM implementations, we refer to the former as the *main JVM* and the latter as the *meta JVM*.

By invoking a second, parallel JVM, the implementation of `evilMethod` attempts to reconstruct the state of the main JVM. In some cases, this may require sophisticated reasoning, but in others, it is as straightforward as the following sequence of steps:

1. Use a `Throwable` object within `evilMethod` to get the backtrace of the main JVM's computation up to the call to `evilMethod`.

2. Print the backtrace information on a `PrintWriter` object.

3. Parse the contents of the `PrintWriter` object to find the class file whose `main` method was invoked to start the computation.

4. Use the meta JVM to load that class and execute its bytecodes to recreate a parallel state. (This will replay the entire computation of the main JVM, loading more classes as necessary.)

5. Read the values of the private data members of `x` in the meta JVM. This is possible since the meta JVM has its own internal data structures that parallels the data structures used in the main JVM.

Figure 1 illustrates the idea.

We have already implemented most of a meta JVM in Java, and have tried the simple examples in this paper to confirm the feasibility of the idea. More sophisticated examples will be possible once we have the complete implementation of the meta JVM.

An analogy might help here: consider an entity (the IRS, a suspicious spouse, etc) who needs to know the amount of money in a bank account. The owner of the bank account considers this information private and will not divulge it. Suppose an expert banker with knowledge of banking processes and transactions was available for hire. If this banker could gain access to a backtrace of all transactions from the legitimate bank, the transactions

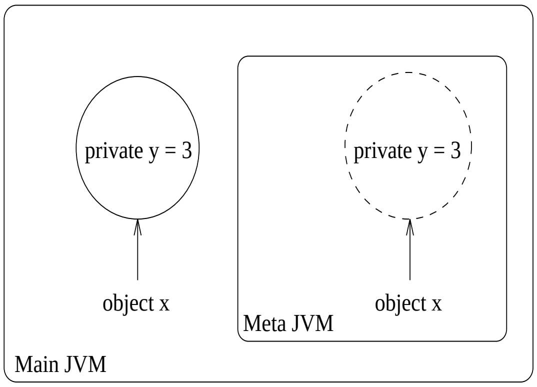

Figure 1: Recreating the State of the Main JVM

together with the banker's knowledge of banking practice could be sufficient to reconstruct the state of the account, including the current balance. The IRS is our evil method; the expert banker is our meta JVM.

Unlike many of the security-related problems reported at http://java.sun.com/sfaq/-chronology.html, this gap in access control is not about an implementation bug. We are assuming that the main JVM is operating according to its specification. It just appears that the backtrace is revealing too much information about the state of the main JVM. In other words, to protect the current value of a private variable, it is also necessary to protect the computation that was used to produce it.

In the next section, we outline the implementation of `evilMethod` in more detail. Section 3 concludes.

2 How Private is Private?

Consider a simple object with a private instance variable:

```
class A {
  private int x;
  A (int x) { this.x = x; }
}
```

The Java specification states that the instance variable `x` is not accessible from any other class. This property is not only important for encapsulation but also for security [3]. The only exception to this rule is that, after proper security checks, certain sophisticated clients of the Java core reflection API are given capabilities to access private fields by disabling the default access control checks. (See: http://www.javasoft.com/products/jdk/1.2/docs/-guide/reflection/reflection.html for more details.) Our scheme enables *any* client, irrespective of its privilege, to sometimes find the values of the private variables.

Both the compiler and the JVM check for possible violations in access control. At compile time, the following method is rejected:

```
class B {
  public static void evilMethod (A a) {
    System.out.println("The value of a.x is " + a.x);
  }
}
```

But due to the dynamic nature of Java, compiler checks are not sufficient. A malicious user can fool the compiler by writing a new version of class `A` with a `public` modifier, compile the code in class `B` against it, and then delete the spurious class `A`. Or even better, it is possible to completely bypass the compiler and generate the desired bytecodes directly. In both cases, the JVM must dynamically check the validity of the access `a.x`. In our test cases, only the JVM in version 1.2beta3 of the Java Development Kit (JDK) caught the violation by throwing an `IllegalAccessError` exception. Earlier versions of the JVM erroneously allowed the access. (And strangely enough the JVM specification [2] does not appear to require that the instructions `getField` and `putField` enforce access control other than for `protected` variables.)

But even when JVM implementations take care to protect the value of a private variable from illegal access, they readily expose the sequence of steps that was used to compute the value of that variable in the form of a backtrace. A malicious program can, in some situations, exploit this information to find the value of the private variable.

To explain this point, let's consider a small but complete application that uses the classes `A` and `B` above:

```
class Test {
  public static void main (String[] args) {
    A a = new A(3);
    B.evilMethod(a);
  }
}
```

From within class `B`, instead of asking directly for the value of `a.x`, we'll instead ask for the backtrace of the entire computation up to the point of the call. The backtrace is easily accessible using `Throwable` objects and then parsed to produce the name of the class whose `main` method started the computation. The outline of the solution is then:

```
class B {
  public static void evilMethod (A a) {
    Throwable te = new java.lang.InternalError(); // any exception would do
    java.io.StringWriter sw = new java.io.StringWriter();
    te.printStackTrace(new java.io.PrintWriter(sw));
    String backtrace = sw.toString();

    String className = parseBackTrace(backtrace); // definition omitted

    // Invoke meta JVM ...
    String[] metaArgs = { className };
    MetaJVM.start(metaArgs);
```

```
    // Inspect the data structures of the meta JVM to get the value of a.x
  }
}
```

When executing this code, the string `backtrace` in `evilMethod` gets the value:

```
java.lang.InternalError
  at B.evilMethod(Evil.java:15)
  at Test.main(Evil.java:9)
```

It is easy to parse this string to bind the variable `className` to the string `Test`. The remainder of the solution is omitted since it depends on the internals of the our `MetaJVM` which are not relevant here.

3 Conclusion

We can write a Java library that can, in some cases, illegally find the values of private variables in an application. The technique used for the illegal access has some limitations:

1. It needs access to the application's bytecodes. The most straightforward approach would be to access the bytecodes contained in the main JVM. However, typical JVM implementations do not make loaded bytecodes available, nor do any of the standard Java libraries, *e.g.*, the reflection package. This forces the meta JVM to directly read the application's class files, which means that it must do a set of file reads. This is a clear weakness in the evil plan, and a place where it can fortunately be foiled. In particular, if a security policy is in force that precludes file reads, as it is for most applets, then our scheme fails - we cannot access the application's code to recreate state information.

2. It relies on the format of the stack trace which is non-portable across JVM implementations.

3. It works best if the sequence of steps used to produce the value of the private variable is deterministic, *i.e.*, can be easily replayed. For simplicity, we have replayed the entire computation to find the value of the private variable, but in practice that value might only depend on the last few method calls in the backtrace. If the value of the private variable depends on interactive input, it may be impossible to replay the computation. (But again the stack trace might help determining the control flow implied by the input, or even the form of the input itself.)

In summary, the access modifier `private` by itself is not sufficient to guarantee that some private value won't be read. It must be used in conjunction with either a trace package that does not make trace information available to an evil method, or a security manager that prevents the evil method from accessing user bytecodes to recreate state.

It is not yet clear if it is possible to exploit this technique for serious security violations. We defer an analysis of this question and a report on our meta JVM to future papers.

References

[1] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, Mass., 1996.

[2] LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, Mass., 1997.

[3] OAKS, S. *Java Security.* O'Reilly & Associates, Inc., Sebastopol, California, 1998.