Java Security in Parallel Universes[1]

Amr Sabry[2]   Stephen Fickas[3]
Department of Computer Science
University of Oregon
Eugene, OR 97403

Summary

We describe a gap in Java's security model that allows a pure Java library to infer information to which it should not have access. The security violation exploits two pieces of public information:

1. The stack trace: Every Java exception object contains a string representation of the stack trace that is usually used for debugging purposes. Access to this stack trace is public.

2. The class files: Every Java program (application or applet) is composed of class files. Access to the class files is usually public, *e.g.*, they are sent over the network, and they have a standard format.

By putting together these two pieces of information, a malicious Java library can sometimes construct a *parallel universe* in which Java security rules are not enforced. In particular, the parallel universe contains copies of private variables which can be accessed by any Java class.

We explain the problem in the current security model, analyze the parallel universe attack, and discuss ways to provide better security assurance for Java.

---

[1]This paper expands on a earlier technical report [11].

[2]Partially supported by the National Science Foundation under the title: "Career: Controlling Space Properties of Higher-Order Typed Programs", grant number CCR-9733088.

[3]Partially supported by an Intel research grant.

The research was partially sponsored by the Advanced Research Projects Agency under the title: "Quality of Service Dynamic Validation Qualifiers – The ASSERT System", ARPA order number F269, 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.

1 Programming Languages and Security

Modern programming languages provide several guarantees in the form of type safety, memory safety, and security. The Java programming language in particular makes strong *but mostly informal* security guarantees. One of Java's fundamental, but informally stated, security guarantees is the protection of the confidentiality of information stored in `private` variables. This guarantee is the focus of our paper.

What is the security requirement? To prevent attackers from obtaining confidential information, it is first necessary to define, precisely, what is meant by *obtaining confidential information*. The current interpretation in the Java security model is that an attacker cannot obtain confidential information without gaining physical access to the memory locations holding a private variable. Enforcing this interpretation prevents a large class of attacks known in the security literature as *direct attacks* [10]. But the security literature is also abundant with another class of attacks called *indirect attacks*. An indirect attack receives information from or about an object without attacking the object itself [10]. The Java security model simply does not consider such attacks and needs to be modified to take them into account. We formulate an appropriate new security requirement that states that confidential information should not even be *observable*, let alone accessible, in any Java context.

Is the security requirement violated? Having precisely defined the security requirement, we ask whether the Java language rules enforce it. As we demonstrate using a new indirect attack, the answer is negative. Our attack, the *parallel universe attack* (PUA), exploits public information to reconstruct the state of execution of a Java program, including copies of all its private variables.

How to prevent this and other violations? The natural question is how to reconcile the security requirement with the existence of attacks such as the PUA. Two approaches are possible:

1. Modify the Java language specification to prevent the PUA.
2. Relax the Java security requirement that we formulated, and explicitly state that, contrary to statements in recent books on Java security [8:p.18], `private` variables should *not* be used to encapsulate sensitive or confidential information.

In both cases, it is desirable to provide some assurance (*e.g.*, a proof) that no further violations are possible.

Outline. The remainder of the paper is organized as follows. The next section reviews the current Java security model. In Section 3, we formulate a security requirement that states that confidential information should not be observable. Section 4 illustrates an attack that violates this security requirement (while remaining "under the radar" when viewed in the narrower scope of direct attacks). Section 5 discusses ways to reconcile the conflict. Section 6 concludes.

2 A Review of Java Security

The Java security model was designed to protect the confidentiality and integrity of information on the host computing infrastructure [8]. In this context, objects to be protected include memory resources on the user's machine as well as other system resources (CPU, disk, etc). In Java, many aspects of the security model (*e.g.*, access to the disk) can be tailored by applications using class loaders, security managers, and access controllers. The security components that are built-in the Java language itself primarily protect the memory map. They are the most essential part of the Java security model [6].

Within a Java program, every entity that resides in memory (object reference or primitive data element) has an access level associated with it. The access level is one of the following four possibilities: `private`, default (package), `protected`, or `public`. Like many other documents, a recent book on Java security [8] emphasizes that, in addition to the usual software engineering benefits of encapsulation, Java access modifiers provide a convenient way of storing sensitive and confidential information such as credit card numbers. A fundamental goal of Java security is then to

protect the confidentiality and integrity of information stored in private variables.

In an attempt to guarantee this aspect of security, the Java language specifies that the following rules must be enforced [8]:

- Access levels are strictly adhered to (unless security checks indicate a privileged client).
- Programs cannot access arbitrary memory locations.
- Entities that are declared `final` must not be changed.
- Variables may not be used before they are initialized.
- Array bounds must be checked on all array accesses.
- Objects cannot be arbitrarily cast into other objects.

A violation of any of these rules allows malicious Java applications to illegally access information stored in `private` variables. For example, by willfully writing past the end of one (public) array, one can change the contents of another (private) object in memory.

2.1 Enforcing Access Control

We explain the mechanisms used to enforce the access control rule in detail. Enforcing the other rules is explained in various Java sources [5, 7, 8].

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. Applications may rely on this promise by storing sensitive information in the field `x`.

To enforce the access control rule, program-manipulation tools, *i.e.*, Java compilers, bytecode verifiers, and runtime systems, perform a variety of checks. There is one exception: after proper security clearance, certain clients of the Java core reflection API are given capabilities to disable the default checks. (See: http://www.javasoft.com/products/jdk/-1.2/docs/guide/reflection/reflection.html for more details.) This capability is used by clients such as object serializers and debuggers.

An unprivileged, malicious client attempting to compromise the confidentiality or integrity of `x` faces several hurdles. A naive attempt to directly access `x` is caught by the compiler:

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

class B {
  public static void naive (A a) { // method rejected by compiler
    System.out.println("The value of a.x is " + a.x);
  }
}
```

However, it is well known that compiler checks are an insufficient mechanism to enforce the access control rule. First, it is possible to directly generate the desired bytecodes. Even simpler, it is possible to fool the compiler by writing a new version of class `A` in which the field `x` is declared with a `public` modifier, compile the code in class `B` against this new version, and then use the generated bytecodes to access the original `x`.

Compiler checks must therefore be supplemented by dynamic checks performed by the Java Virtual Machine (JVM). A subtlety here is that version 1.1 of the Java Developer's Kit (JDK) allows the access `a.x` when the classes `A` and `B` are loaded from the (trusted) local file system. Later versions of the JDK (version 1.2) catch the attempted access to `a.x` by throwing an `IllegalAccessError` exception, irrespective of the origin of the classes `A` and `B`.

A more fundamental problem is that the collection of rules specified by the Java language is necessary but *not* sufficient to protect the confidentiality and integrity of information stored in private variables.

3 Confidentiality Requirement

The notion of *access control* is not unique to Java: it occurs in operating systems and databases where it is perhaps more crucial. A related security attack common in databases is *database inference* [9]. In such attacks, information is received from or about an object without attacking the object itself. Instead, the attacker asks indirect questions to the database about the object, and from the answers derives confidential information.

In our case, the Java runtime system plays the role of the database. Not only should it catch direct questions asking for the value of a private variable as illustrated in Section 2.1, but also it should not provide answers to indirect questions that could be used to derive such values. Also, to protect the confidentiality of private information at the language level, attackers should not be able to obtain confidential information about even *one* object. In other words, the Java runtime system is responsible for protecting all objects: leaving some objects susceptible to attacks compromises the security of the entire language.

Thus, when taking indirect attacks into account, it is not enough to restrict access to confidential information. We must also restrict other channels that might be used to infer the confidential information. From a semantic point of view, confidential information should not be *observable*. The definition of observational equivalence is standard and repeated here for convenience.

Definition 3.1 *Two terms M and N are* observationally equivalent*, written $M \cong N$, if for all contexts C such that $C[M]$ and $C[N]$ are programs, the evaluation of $C[M]$ terminates if and only if the evaluation of $C[N]$ terminates.*

Intuitively, two observationally equivalent program fragments always behave the same, and are identical as far as users are concerned. Thus an attacker, who is denied direct access to a private piece of information, is content with constructing an observationally equivalent version by indirect means.

This connection to observational equivalence leads to the following definition of what it means for one object to leak the value of a private variable. The definition is adapted from a definition given by Abadi which defines what it means for a security protocol to leak a secret value [1].

Definition 3.2 *Let $I(x)$ be a Java object with a private instance variable x. We say $I(x)$ does not leak the value of x if no pure Java context can distinguish $I(V)$ from $I(V')$ for all values V and V'. In other words,*

$$\forall C.\ \forall V, V'.\ C[I(V)] \cong C[I(V')]$$

Since allowing native methods would easily defeat any security property, the definition only considers pure Java contexts. The exact definition of a pure Java context is subtle as such contexts should allow uses of the Java core API libraries which ultimately use native (but trusted) methods. We do not give the precise definition of pure contexts but note that the security manager can be used to restrict contexts as desired by disallowing applications from loading or linking (untrusted) native libraries.

3.1 Confidential Information

So far we have tacitly assumed that a confidential value coincides with the value of a private variable. This is clearly not correct in general since applications may freely expose the value of a private variable. Indeed a common object-oriented pattern is:

```
class A {
  private int x;
```

```
  public int getx () { return x; }
  ...
}
```

Thus even though the variable `x` is `private`, its contents are not treated as confidential by the application.

Since we are looking at a language property, we abstract from such behavior and only require that objects with no public methods whatsoever do not leak information.

3.2 Summary

The following requirement is a first step towards a useful statement of privacy that can at least set a context for discussion of access attacks in Java.

Requirement 3.1 *Let $I(x)$ be an object with a private variable x and no public methods. The Java language protects confidential information if no object $I(x)$ leaks the value of x (See Definition 3.2).*

4 The Parallel Universe Attack

The idea of the attack rests on the following observation. The Java language specification allows arbitrary Java code to get access to a textual representation of the stack trace (using exception objects). In some cases, such a representation can be parsed and replayed using a custom JVM implementation. To fulfill the requirement that the attack only uses pure Java code, the custom JVM is written in Java. In fact, we discovered the attack after having implemented a JVM in Java as a instructional tool, and we have used our JVM to confirm the feasibility of the simple examples in this paper. There is at least one other JVM implementation written in Java [12], but which is not accessible to us.

To avoid confusion we distinguish between the JVM which is invoked (and trusted) by the user and running the main computation, the main JVM, and the JVM written in Java and invoked by the attacker, the meta JVM.

To explain the attack in more detail, let's consider a small but complete application:

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

class B {
  public static void evilMethod (A a) {
    ... // see below
  }
}

class Test {
  public static void main (String[] args) {
```

```
    A a = new A(3);
    B.evilMethod(a);
  }
}
```

The goal is to get access to the value of `a.x` from within class `B`. The first step is to fool or entice the user's application into calling `evilMethod`, perhaps using a library application that users would be willing to download and run with a minimal security manager. After all we promise the application is written in pure Java, and hence should be trusted. And of course, we'll replace `evilMethod` with a less obvious name.

From within class `B`, instead of asking directly for the value of `a.x` (which would cause an `IllegalAccessError` exception), we instead ask for the stack trace of the entire computation up to the point of the call. The stack trace is easily accessible using `Throwable` objects and then parsed to produce the name of the class whose `main` method started the computation. Then `evilMethod` simply attempts to reconstruct the state of the main JVM by invoking the meta JVM.

More concretely, the entire sequence of steps is:

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

2. Print the stack trace 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 `a.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.

Or in code:

```
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 trace = sw.toString();

    String className = parseTrace(trace); // definition omitted

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

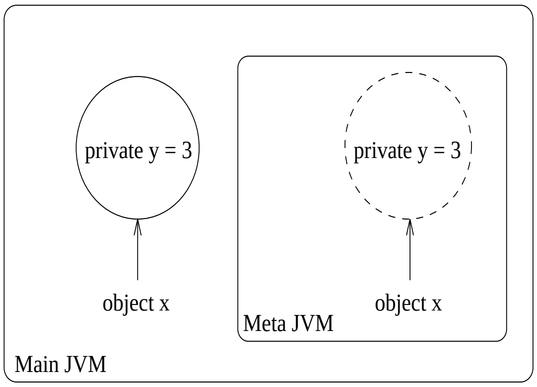

Figure 1: Recreating the State of the Main JVM

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

When executing this code, the string `trace` 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.

Figure 1 illustrates the idea. An analogy might also help: 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 (under any direct attacks). Suppose an expert banker with knowledge of banking processes and transactions was available for hire. If this banker could gain access to a trace of all transactions from the legitimate bank, the transactions 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.

5 Security Assurance

Using the meta JVM, it is easy to check that the PUA violates our security requirement (Requirement 3.1). The mismatch between the security requirement and the presence of the PUA can be resolved in two fashions.

5.1 Modifying the Language

A first attempt at stymieing the PUA might be to perform an informal analysis of the problem, find the conditions under which the attack occurs, and thwart one or more of these conditions, *e.g.*, eliminate access to the textual representation of stack traces. We briefly discuss such an approach.

The PUA rests of the following assumptions:

1. *The application's bytecodes are accessible*: The most straightforward way 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. This is a clear weakness in the attack plan, and a place where it can fortunately be foiled. In particular, if a security policy is in force that precludes reading files, as it is for most applets, then the attack fails — it cannot access the application's code to recreate state information. However this is not a general solution: preventing applications from doing any input is too restrictive.

2. *A textual representation of the stack trace is accessible*: The stack trace is invaluable for debugging purposes: limiting access to it appears too restrictive. However to be useful for an attacker, the trace should be parsed to expose the structure of the stack frames. Fortunately, the textual representation of the stack trace is not specified and hence non-portable across JVM implementations, which makes the attack itself non-portable. Going one step further, the stack trace could be made opaque in the form of a *continuation* object. But this is in general insufficient to stop all attacks. Even without access to the stack trace, an attacker might try to read all the class files that it can access on the host machine, search for all methods called `main`, and execute each of them in an attempt to reconstruct the parallel universe.

3. *It is possible to (partially) replay the stack trace*: The meta JVM attempts to reproduce the state of the main JVM by replaying the stack trace. In simplistic situations in which the computation is deterministic (*e.g.*, consists of one thread and performs no interactive input) the replay is straightforward. In more realistic situations involving several threads, a graphical interface, and interactive input, an exact replay is in general impossible. However the stack trace again comes to the rescue: it is possible to use the trace to infer the control flow implied by interactive input. For example, if the body of method `m1` consists of:

   ```
   if (someInput)
     m2();
   else m3();
   ```

   and the stack trace indicates that `m1` calls `m2` then the replay can proceed without knowing the value of `someInput`.

5.2 Relaxing the Security Requirement

Several authors have already noted that the definition of Java is weaker than it should be from a security viewpoint [3]. Our analysis re-states Abadi's observation (stated for a different reason):

> In hindsight, it is not clear whether one should base any security expectations on qualifiers like `private` [2:p.7].

Thus, despite several discussions in the Java literature that `private` variables are adequate for storing confidential information, there is growing evidence that private variables should not be used for security purposes.

5.3 Assurance

No matter which approach we choose to resolve the conflict, we are still faced with the more general problem of providing some assurance that other attacks could not slip through.

Ideally, assurance would mean a mathematical proof, which requires a formal statement of the security requirement and a formal semantics for the language. However, we note that the context the PUA uses to observe the confidential information includes the entire meta JVM implementation: it is about 15,000 lines of Java code and uses many of the facilities in the Java core API which is written in `C/C++`. Thus, an analysis of the security properties of Java needs to take into account the entire language, including the libraries, the non-portable constructs, and system issues such as the accessibility of class files. It is unlikely that any such analysis could lead to a proof (whether manual or automatic) that could be used to discover our context.

In general, in the security literature, indirect attacks like database inference are known to be troublesome and are hard to prevent or detect [10]. Preventing or detecting such attacks in the context of Java appears no easier.

A more pragmatic approach (also discussed by the designers of the Java security model [4]) is to support the high-level security requirement by lower-level security properties enforced by other system components such as the class loader, the security manager, etc. It should then be possible to demonstrate security assurance for the JDK by showing that the lower level properties compose together to imply the global security invariant. If this demonstration is to be done based on a mathematical analysis, it would still require a formal and complete model of the JVM.

6 Conclusion

We have noted that, although the Java literature includes some examples that use `private` variables to hold sensitive confidential information, the Java security model did not recognize indirect attacks that attempt to infer such information. We have discovered a security attack that can infer confidential information in some simple cases. At this point, it is not clear if one can exploit this attack to get sensitive information in the real world. We are exploring some possibilities.

Our analysis re-raises the question of whether it is appropriate to base the security of Java on qualifiers such as `private`.

Acknowledgments

We benefited from discussions with Corky Cartwright, Matthias Felleisen, Zary Segall, and Guy Steele.

References

[1] Abadi, M. Secrecy by typing in security protocols. In *Third International Conference on the Theoretical Aspects of Computer Software* (Sept. 1997), Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 611–638. Paper for an open lecture given in conjuction with the symposium.

[2] Abadi, M. Protection in programming-language translations. Tech. Rep. SRC Research Report 154, Digital Systems Research Center, Palo Alto, CA, Apr. 1998.

[3] Dean, D., Felten, E. W., Wallach, D. S., and Balfanz, D. Java security: Web browsers and beyond. In *Internet Beseiged: Countering Cyberspace Scofflaws* (1998), D. E. Denning and P. J. Denning, Eds., ACM Press, pp. 241–269.

[4] Erdos, M., Hartman, B., and Mueller, M. Security reference model for the Java developer's kit 1.0.2. Available at http://www.javasoft.com/security/SRM.html, November 1996.

[5] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, Mass., 1996.

[6] Javasoft. Secure computing with Java: Now and the future. White paper prepared for the JavaOne conference, 1997.

[7] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, Mass., 1997.

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

[9] O'Leary, D. E. Knowledge discovery as a thread to database security. In *Knowledge Discovery in Databases*, G. Piatetsky-Shapiro and W. J. Frawley, Eds. AAAI Press, Menlo Park, California, 1991, ch. 30, pp. 507–516.

[10] Olovsson, T. A structured approach to computer security. Tech. Rep. 122, Department of Computer Engineering, Chalmers University of Technology, 1992.

[11] Sabry, A., and Fickas, S. Java access modifiers in parallel universes. Tech. Rep. CIS-TR-98-03, Department of Computer and Information Science, University of Oregon, June 1998.

[12] Taivalsaari, A. Implementing a Java virtual machine in the Java programming language. Tech. Rep. SMLI TR-98-64, Sun Microsystems, Palo Alto, CA, Mar. 1998.