Low Level Security in Java

Frank Yellin

Abstract

The Java(tm) language allows Java-compatible Web browsers to download code fragments dynamically and then to execute those code fragments locally. However, users must be wary of executing any code that comes from untrusted sources or that passes through an insecure network.

This paper presents the details of the lowest-levels of the Java security mechanism. Before any downloaded code is executed, it is scanned and verified to ensure that it conforms to the specifications of the virtual machine.

Keywords

WWW; Java; HotJava; Security; Remote execution;

Introduction to the Java Language

The Java(tm) language [1] [2] is a simple, object-oriented, portable, robust language developed at Sun Microsystems.

The language was created for developing programs in a heterogenous network-wide environment. Since one of the initial goals of the language was to be used in embedded systems with a minimum amount of memory, the Java language is designed to be small and to use a small amount of hardware resources.

The Java compiler generates **class** files, which have an architecturally neutral, binary intermediate format. Embedded in the class file are **bytecodes**, which are implementations for each of the class's methods, written in the instruction set of a virtual machine. The class file format has no dependencies on byte-ordering, pointer size, or underlying operating system.

The bytecodes are executed by means of a **runtime system**, an emulator for the virtual machine's instruction set. The same bytecodes can be run on any platform.

Security

Since Java language compiled code is designed to be transported in binary format across networks, security is extremely important. No one wants to bring across any piece of code if there is a possibility that executing the code could do any of the following:

- Damage hardware, software, or information on the host machine.
- Pass unauthorized information to anyone.
- Cause the host machine to become unusable through resource depletion.

Because the Java bytecode is run on the host machine, there are special security concerns. Users who download Java class files from remote, possibly insecure (or hostile) sites must be satisfied that the downloaded code cannot subvert the Java bytecode interpreter to perform impermissible operations.

The lowest levels of the Java interpreter implement security in several ways.

Security though being published.

The complete source code for both the Java interpreter and the Java compiler are available for inspection. We do not expect users to take our word for it that Java language is secure. Security audits of the Java source are currently being performed.

Security through being well-defined.

The Java language is strict in its definition of the language:

- All primitive types in the language are guaranteed to be a specific size.
- All operations are defined to be performed in a specified order.

Two correct Java compilers will never give different results for execution of a program. This is far different from C and C++, in which the sizes of the primitive types are machine- and compiler-dependent, and the order of execution is undefined except in certain specific cases.

Security through lack of pointer arithmetic.

The Java language does not have pointer arithmetic, so Java programmers cannot forge a pointer to memory. All references to methods and instance variables in the class file are via symbolic names. The user cannot create code that has magic offsets in it that just happen to point to the "right place." Users cannot create code that bash system variables or that accesses private information.

Security through garbage collection.

Garbage collection [3] makes Java programs both more secure and more robust. Two common bugs in C/C++ programs are:

- Failing to free memory once it is no longer needed.
- Accidentally freeing the same piece of memory twice.

Failing to free memory that is no longer accessible can cause a program to use increasing amounts of memory. Accidentally freeing the same piece of memory often causes subtle memory corruption bugs that are difficult to locate. The Java language eliminates the need for programmers to be concerned with these issues.

Security through strict Compile-Time Checking.

The Java compiler performs extensive, stringent, compile-time checking so that as many errors as possible can be detected by the compiler. The Java language is strongly typed; unlike C/C++, the type system has no loopholes:

- Objects cannot be cast to a subclass without an explicit runtime check.
- All references to methods and variables are checked to make sure that the objects are of the appropriate type. In addition, the compiler checks that "security barriers" (e.g., referencing a `private` variable or method from another class) are not violated.
- Integers cannot be converted into objects. Objects cannot be converted into integers.

The compiler also strictly ensures that a program does not access the value of an uninitialized local variable.

Class File Verification

Even though the compiler performs thorough type checking, there is still the possibility of attack via the use of a "hostile" compiler. Applications such as the HotJava(tm) browser [4] do not download source code which they then compile; these applications download already-compiled class files. The HotJava browser has no way of determining whether the bytecodes were produced by a trustworthy Java compiler or by an adversary attempting to exploit the interpreter.

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: methods might have disappeared or changed arguments; variables might have changed types or changed from dynamic (per object) to static (per class). The visibility of a method or variable may have changed from `public` to `private`.

All class files brought in from "the outside" are subjected to a verifier. This verifier ensures that the class file has the correct format. The bytecodes are verified using a simple theorem prover which establishes a set of "structural constraints" on the bytecodes.

The bytecode verifier also enhances the performance of the interpreter. Runtime checks that would otherwise have to be performed for each interpreted instruction can be eliminated. Rather, the interpreter can assume that these checks have already been performed. Though each individual check may be inexpensive, several machine instructions for the execution of each bytecode instruction are eliminated.

For example, the interpreter already knows that the code will adhere to the following constraints:

- There are no stack overflows or underflows.
- All register accesses and stores are valid.
- The parameters to all bytecode instructions are correct.
- There is no illegal data conversion.

The verifier is independent of the Java compiler. Although it will certify all code generated by the current compiler, it should also certify code that the current compiler couldn't possibly generate. Any set of bytecodes that satisfy the structural criteria will be certified by the verifier.

The verifier is extremely conservative. It will refuse to certify some class files that a more sophisticated theorem prover might certify.

Other languages can be compiled into the class format. The bytecode verifier, by not being specifically tied to the Java language, allows users to import code from outside their firewall with confidence.

The Class File Format

Each Java class file is downloaded across the network as a separate entity. The class file is simply a stream of 8-bit bytes. All 16-bit and 32-bit quantities are formed by reading in two or four 8-bit bytes, respectively, and joining them together in big-endian format.

The Basic Format

Following is a brief sketch of the class file format. Complete details can be found in [5].

A class file contains:

- A magic constant
- Major and minor version information
- The "constant pool"
- Information about this class (name, superclass, etc.)
- Information about each of the fields and methods in this class
- Debugging information.

The "**constant pool**" is a heterogenous array of data. Each entry in the constant pool can be one of the following:

- A Unicode [6] string
- A class or interface name
- A reference to a field or method
- A numeric value
- A constant String value

No other part of the class file makes specific references to strings, classes, fields, or methods. All such references are through indices into the constant pool.

For each field and method in the class, the bytes in the class file indicate the field or method's name and its type. The type of a field or method is indicated by a string called its **signature**. Fields may have an additional attribute giving the field's initial value. Methods may have an additional attribute giving the code for performing that method.

Methods may, in fact, have multiple code attributes. The attribute CODE indicates bytecode to be run through the interpreter. Methods might also have attributes such as SPARC-CODE or 386-CODE which are machine-code implementations of the method. The HotJava browser will ignore the machine-code implementation of any method from an untrustworthy source, since it cannot verify that machine code is structurally sound.

The current implementation of the HotJava browser believes that any class file that comes from the network is untrustworthy. It will only run machine code that has been loaded from local class files. However, the class format can allow authors to digitally sign class files. Future browsers may be more trusting of signed machine code coming from trusted sources.

The Bytecodes and the Virtual Machine

The CODE attribute supplies information for executing the method in the machine language of a virtual machine. The information for each method includes:

- The maximum stack space needed by the method.
- The maximum number of registers used by the method.
- The actual code for executing the method. These bytecodes are for the Java virtual machine.
- A table of exception handlers. Each entry in the table gives a start and end offset into the bytecodes, an exception type, and the offset of a handler for the exception. The entry indicates that if an exception of the indicated type occurs within the code indicated by the starting and ending offset, a handler for the exception will be found at the given handler offset.

The Java virtual machine defines six primitive types:

- 32-bit integer ("**integers**")
- 64-bit integers ("**longs**" or "**long integers**")
- 32-bit floating-point numbers ("**single floats**")
- 64-bit floating-point numbers ("**double floats**")
- pointers to objects and arrays ("**handles**")
- pointers to the virtual machine code ("**return addresses**")

The Java virtual machine also defines several array types: these include arrays of integers, longs, single floats, double floats, handles, booleans, bytes (8-bit integers), shorts (16-bit integers), and Unicode characters. Arrays of handles have an additional type field indicating the class of object the array can hold.

Each method activation has a separate expression-evaluation stack and set of local registers. Each register and each stack location must be able to hold an integer, a single float, a handle, or a return address. Longs and double floats must fit into two consecutive stack locations or two consecutive registers. The virtual-machine instructions ("**opcodes**") will address longs and double floats in registers using the index of the lower-numbered register.

Objects on the stack and in registers are not (necessarily) tagged. The virtual-machine instruction set provides opcodes to operate on different primitive data types. For example, `ineg`, `fneg`, `lneg`, and `dneg` each negate the top item on the stack, but they assume that the top item on the stack is an integer, a single float, a long, or a double float, respectively.

The bytecode instructions can be divided into several categories:

- Pushing constants onto the stack
- Accessing and modifying the value of a register
- Accessing arrays
- Stack manipulation (e.g., `swap`, `dup`, `pop`)
- Arithmetic, logical, and conversion instructions
- Control transfer
- Function return
- Manipulating object fields
- Method invocation
- Object creation
- Type casting

Each bytecode consists of a one-byte opcode, followed by zero or more bytes of additional operand information. With the exception of two "table lookup" instructions, all instructions are a fixed length, based on the opcode.

The Verification Process

The Verifier operates in four passes.

Pass 1

The first pass is the simplest. It occurs when the class is first read into the interpreter.

This pass ensures that the class file has the format of a class file. The first several bytes must contain the right magic number. All recognized attributes need to be the proper length. The class file must not be truncated or have extra bytes at the end. The constant pool must not contain any unrecognized information.

Pass 2.

In the second pass, the verifier delves a little bit more deeply into the class file format. It performs all verification that can be performed without looking at the bytecodes. The errors detected by Pass 2 include:

- Ensuring that final classes are not subclassed, and that final methods are not overridden.
- Checking that every class (except Object) must have a superclass.
- Ensuring that the constant pool satisfies certain constraints. For example, class references in the constant pool must contain a field that points to a unicode string reference in the constant pool.
- Checking that all field references and method references in the constant pool must have legal names, legal classes, and a legal type signature.

Note that when looking at field and method references, this pass does not actually check to make sure that the given field or method really exists in the given class; nor does it check that the type signatures given refer to real classes. Rather, the signature must simply "look like" a legal signature. Further checking is delayed until Passes 3 and 4.

Pass 3

This is the most complex pass of the class verification. The bytecodes of each method are verified. Data-flow analysis [7] is performed on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point:

- The stack is always the same size and contains the same types of objects.
- No register is accessed unless it is known to contain a value of the appropriate type.
- Methods are called with the appropriate arguments.
- Fields are modified with values of the appropriate type.
- All opcodes have appropriate type arguments on the stack and in the registers.

Further information on this pass, see the section "The Bytecode Verifier."

Pass 4

For efficiency reasons, certain tests that could be performed in Pass 3 are delayed until the code is actually run. Pass 3 of the verifier avoids loading class files unless it has to.

For example, if a method contains a call to another method that returns an object of type foobarType, and that object is then immediately assigned to a field of the same type, the verifier doesn't bother to check if the type foobarType exists. However, if it is assigned to a field of the type anotherType, the definitions of both foobarType and anotherType must be loaded in to assure that foobarType is a subclass of anotherType.

The first time an instruction that references a class is executed, the verifier does the following:

- Loads in the definition of the class if it has not already been loaded.
- Verifies that the currently executing class is allowed to reference the given class.

The first time an instruction calls a method, or accesses or modifies a field, the verifier does the following:

- Ensures that the method or field exists in the given class.
- Checks that the method or field has the indicated signature.
- Checks that the currently executing method has access to the given method or field.

This pass of the verifier does not have to check the type of the object on the stack. That check has already been done by Pass 3.

After the verification has been performed, the instruction in the bytecode stream is replaced with an alternative form of the instruction. For example, the opcode new is replaced with new_quick. This alternative instruction indicates that the verification needed by this instruction has taken place, and need not be performed again. It is illegal for these _quick instructions to appear in Pass 3.

The Bytecode Verifier

As indicated above, Pass 3 of the verifier, the **bytecode verifier**, is the most complex pass of the class verification.

First, the bytes that make up the virtual instructions are broken up into a sequence of instructions, and the offset of the start of each instruction is kept in a bit table. The verifier then goes through the bytes a second time and parses the instructions. During this pass each instruction is converted into a structure. The arguments, if any, to each instruction are checked to make sure they are reasonable:

- All control-flow instructions go to the start of an instruction. Branches into the middle of an instruction are clearly not allowed. Similarly, branches to before the beginning of the code or to after the end of the code are not allowed.
- All register references are to a legal register. Code cannot access or modify any register greater than the number of registers that the method indicated it uses.
- All references to the constant pool must be to an entry of the appropriate type. For example, the opcode `ldc1` can only be used for integers, floats, or `String`'s. The opcode `getfield` must reference a field.
- The code does not end in the middle of an instruction.
- For each exception handler, the starting and ending point must point to the beginning of an instruction. The offset of the exception handler must be a valid instruction. The starting point must be before the ending point.

For each instruction, the verifier keeps track of the contents of the stack and the contents of the registers prior to the execution of that instruction. For the stack, it needs to know the length of the stack and the type of each element on the stack. For each register, it needs to know either the type of the contents of that register or that the register contains an illegal value. The bytecode verifier does not need to distinguish between the various normal integer types (e.g., `byte`, `short`, `char`) when determining the value types on the stack.

[Some extra information is kept about each instruction in a `finally` clause. This information is discussed further in the section Try / Finally].

Next, a data-flow analyzer is initialized. For the first instruction, the lower-numbered registers contain the types indicated by the method's type signature; the stack is empty. All other registers contain an illegal value. For all other instructions, indicate that this instruction has not yet been visited; there is yet no information on its stack or registers.

Finally, the data-flow analyzer is run. For each instruction, there is a "changed" bit indicating whether this instruction needs to be looked at. Initially, the "changed" bit is set only for the first instruction. The data-flow analyzer executes the following loop:

1. Find a virtual machine instruction whose "changed" bit is set. If no instruction remains whose "changed" bit is set, the method has successfully been verified. Turn off that "changed" bit.
2. Emulate the effect of this instruction on the stack and registers:
   - If the instruction uses values from the stack, ensure that there are sufficient elements on the stack and that the top element(s) of the stack are of the appropriate type. Otherwise, fail.
   - If the instruction uses a register, ensure that the specified register contains a value of the appropriate type. Otherwise, fail.
   - If the instruction pushes values onto the stack, add the indicated types to the top of the stack. Ensure that there is sufficient room on the stack for the new element(s).
   - If the instruction modifies a register, indicate that the register now contains the new type.
3. Determine the virtual-machine instructions that can follow this one. Successor instructions can be one of the following:
   1. The next instruction, if the current instruction isn't an unconditional `goto`, a `return`, or a `throw`. Fail if we can "fall off" the last instruction.
   2. The target of a conditional or unconditional branch.
   3. All exception handlers for this instruction.
4. Merge the state of the stack and registers at the end of the current instruction into each of the successor instructions. In the exception-handler case (2c), change the stack so that it contains a single object of the exception type indicated by the exception handler information.
   - If this is the first time the successor instruction has been visited, indicate that the stack and registers values calculated in Step 2 and Step 3 are the state of the stack and registers prior to executing the successor instruction; set the "changed" bit for the successor instruction.
   - If the instruction has been seen before, merge the stack and register values calculated in Step 2 and Step 3 into the values already there; set the "change" bit if there is any modification.
5. Go to Step 1.

To merge two stacks, the number of elements in each stack must be identical. A failure is indicated if this isn't the case. The stacks must be identical, except that differently typed handles may appear at corresponding places on the two stacks. In this case, the merged stack contains the common ancestor of the two handle types.

To merge two register states, compare each register. If the two types aren't identical, then unless both contain handles, indicate that the register contains an unknown (and unusable) value. For differing handle types, the merged state contains the common ancestor of the two types.

If the data-flow analyzer runs on the method without reporting any failures, then the method has been successfully verified by Pass 3 of the class file verifier.

Certain instructions and data types complicate the data-flow analyzer. We now examine each of these.

Long Integers and Doubles

Long integers and double floats each take two consecutive words on the stack and in the registers.

Whenever a long or double is moved into a register, the following register is marked as containing the second half of a long or double. This special value indicates that all references to the long or double must be through the lower numbered register.

Whenever any value is moved to a register, the preceding register is examined to see if it contains the first word of a long or a double. If so, that preceding register is changed to indicate that it now contains an unknown value. Since half of the long or double has been eradicated, the other half can no longer be used.

Dealing with 64-bit quantities on the stack is simpler. The verifier treats them as single units on the stack. For example, the verification code for the `dadd` opcode (add two double floats) checks that the top two items on the stack are both double floats. When calculating stack length, longs and double floats on the stack have length two.

Stack manipulation opcodes must treat doubles and longs as atomic units. For example, the verifier reports a failure if the top element of the stack is a double float and it encounters the opcodes `pop` or `dup`. The opcodes `pop2` or `dup2` must be used instead.

Constructors and Newly Created Objects

Creating a usable object in the Java interpreter is a multi-step process. The bytecodes produced for the Java code:

```
new myClass(i, j, k);
```

are roughly the following:

```
new <myClass>          # allocate uninitialized space
dup                    # duplicate object on the stack
<          push arguments   >
invokenonvirtual myClass.<init>  # initialize
```

This code leaves the newly created and initialized object on top of the stack.

The `myClass` initialization method sees the new uninitialized object as its `this` argument in register 0. It must either call an alternative `myClass` initialization method or call the initialization method of a superclass on the `this` object before it is allowed to do anything else with `this`.

In normal instance methods (what C++ calls **virtual**methods), the verifier indicates that register 0 initially contains an object of "the current class"; for constructor methods, register 0 instead contains a special type indicating an uninitialized object. After an appropriate initialization method is called (from the current class or the current superclass) on this object, all occurrences of this special type on the stack and in the registers are replaced by the current class type. The verifier prevents code from using the new object before it has been initialized and from initializing the object twice.

Similarly, a special type is created and pushed on the stack as the result of the opcode `new`. The special type indicates the instruction in which the object was created and the type of the uninitialized object created. When an initialization method is called on that object, all occurrences of the special type are replaced by the appropriate type.

The instruction number needs to be stored as part of the special type since there may be multiple instances of a non-yet-initialized type in existence on the stack at one type. For example, the code created for the following:

```
new InputStream(new Handle(),new InputStream("foo"))
```

may have two uninitialized InputStream's active at once.

Code may not have an uninitialized object on the stack or in a register during a backwards branch, or in a register in code protected by an exception handler or a finally. Otherwise, a devious piece of code could fool the verifier into thinking it had initialized an object when it had, in fact, initialized an object created in a previous pass through the loop.

Exception Handlers

Code produced from the current Java compiler always has properly nested exception handlers:

- The range of instructions protected by two different exception handlers will always either be completely disjoint or one will be a subrange of the other. There will never be a partial overlap.
- The handler for an exception will never be inside the code that is being protected.
- The only entry to an exception handler is through an exception. It is impossible to fall through or "goto" the exception handler.

These restrictions are not enforced by the verifier since they do not pose an threat to the integrity of the virtual-machine interpreter. As long as every non-exceptional path to the exception handler causes there to be a single object on the stack, and as long as all other criteria of the verifier are met, the verifier will pass the code.

Try / Finally

The Java language includes a feature called finally, which is like the similarly-named feature of Modula-3 [8] or unwind-protect in Common Lisp [9]. Given the following code:

```
try {
    startFaucet();
    waterLawn();
} finally {
    stopFaucet();
}
```

The Java language guarantees that the faucet is turned off, even if an exception occurs while starting the faucet or watering the lawn. The code inside the brackets after the try is called the **protected code**The code inside the brackets after the finally is the **cleanup code** The cleanup code is guaranteed to be executed, even if the protected code does a "return" out of the function, or contains a break or continue to outside the try/finally, or gets an exception.

To implement this construct, the Java compiler uses the exception handling facilities, together with two special instructions, jsr (jump to subroutine) and ret (return from subroutine). The cleanup code is compiled as a subroutine. When it is called, the top object on the stack will be the return address; this return address is saved in a register. At the end of the cleanup code, it performs a ret to return to whatever code called the cleanup.

To implement try/finally, a special exception handler is set up around the protected code which catches all exceptions. This exception handler:

1. Saves the exception in a register.
2. Executes a jsr to the cleanup code.
3. Upon return from the exception, re-throw's the exception.

If the protected code has a return, it performs the following code:

1. Saves the return value (if any) in a register.
2. Executes a jsr to the cleanup code.
3. Upon return from the exception, returns the value saved in the register.

Breaks or continues inside the protected code that go to outside the protected code execute a jsr to the cleanup code before performing their goto. Likewise, at the end of the protected code is a jsr to the cleanup code.

The cleanup code presents a special problem to the verifier. Usually, if a particular instruction can be reached via multiple paths and a particular register contains incompatible values through those multiple paths, then the register becomes unusable. However, a particular piece of cleanup code might be called from several different places:

- The call from the exception handler will have a certain register containing an exception.
- The call to implement "return" will have some register containing the return value.

- The call from the bottom of the protected code may have trash in that same register.

The cleanup code may pass verification, but after updating all the successors of the ret instruction, the verifier will note that the register that the exception handler expects to hold an exception or that the return code expects to hold a return value now contains trash.

Verifying code that contains finally's can be somewhat complicated. Fortunately, most code does not have finally's. The basic idea is the following:

- Each instruction keeps track of the smallest number of jsr targets needed to reach that instruction. For most code, this field will be empty. For instructions inside cleanup code, it will be of length one. For multiply-nested cleanup code (extremely rare!), it may be longer than one.
- For each instruction and each jsr needed to reach that instruction, a bit vector is maintained of all registers accessed or modified since the execution of the jsr instruction.
- When executing the ret from a subroutine, there must be only one possible subroutine target from which the instruction can be returning. Two different targets of jsr instructions cannot "merge" themselves into a single ret instruction.
- When performing the data-flow analysis on a ret instruction, modify the directions given above. Since the verifier knows the target of the jsr from which the instruction must be returning, it can find all the jsr's to the target, and merge the state of the stack and registers at the time of the ret instruction into the stack and registers of the instructions following the jsr using a special set of values for the registers:
- For any register that the bit vector (constructed above) indicates that the subroutine has accessed or modified, use the type of the register at the time of the ret.
- For other registers, use the type of the register at the time of the preceding jsr instruction.

Conclusion

The Java language has generated much excitement in its ability to allows programmers to create and compile code that can be executed on multiple platforms. The HotJava browser, in particular, has shown that portable code can bring interactivety to the World Wide Web.

However, before users will consent to bring over executable code from untrustworthy sources (i.e. most of the network!), they want assurances that the code cannot damage them. The byte-code verifier is the lowest-level of a many-tiered strategy [10].

Acknowledgments

Thanks to James Gosling, Arthur van Hoff, Bill Joy, Tim Lindholm, Chuck McManis, Mark Showalter, and Richard Tuck for comments and suggestions. Special thanks to Mark Scott Johnson for encouraging me to write this paper.

References

1. *The Java Language Overview.*

2. James Gosling and Henry McGilton. *The Java Language Overview: A White Paper*. Sun Microsystems Technical Report, May 1995.

3. Donald E Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1969.

4. *The HotJava Overview*.

5. *The Java Virtual Machine Specification.* Available via http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html

6. The Unicode Consortium. *The Unicode Standard: Worldwide Character Encoding*. Addison-Wesley, Reading, Massachusetts, 1992. Available via http://unicode.org/

7. Alfred V.Aho, Ravi Sethi,, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1988

8. Samuel P. Harbison. *Modula-3*. Prentice-Hall, Inc. 1992.

9. Guy L. Steele Jr. *Common Lisp: The Language*, Second Edition. Digital Press, Bedford, Massachusetts, 1990. Available via http://www.cs.cmu.edu/Web/Groups/AI/html/cltl/cltl2.html.

10. HotJava(tm): The Security Story.

About the Author

Frank Yellin
Sun Microsystems
Java Products Group
fy@eng.sun.com

Generated with CERN WebMaker
