Lecture 18: Code Safety and Virtual Machines

How to get more than 256 local variables!

```
wide <opcde> <byte1> <byte2>
```

- Opcode is one of `iload, fload, aload, iload, dload, istore, fstore, astore, istore, dstore, or ret`
- Modifies instruction to take 2 byte operand (`byte1 << 8 | byte2`)

Method Calls

- **invokevirtual <method>**
  - Invokes the method `<method>` on the parameters and object on the top of the stack.
  - Finds the appropriate method at run-time based on the actual type of the this object.

```java
invokevirtual <Method void println(java.lang.String)>
```

Example

```java
public class Sample1 {
    static public void main (String args[]) {
        System.err.println("Hello!");
        System.exit (1);
    }
}
```
> javap -c Sample1
Compiled from Sample1.java
public class Sample1 extends java.lang.Object {
public Sample1();
public static void main(java.lang.String[]);}
Method Sample1()
0 aload_0
1 invokespecial #1 <Method java.lang.Object>()
4 return
Method void main(java.lang.String[])
0 getstatic #2 <Field java.io.PrintStream err>
3 ldc #3 <String "Hello!">
5 invokevirtual #4 <Method void println(java.lang.String)>
8 iconst_1
9 invokestatic #5 <Method void exit(int)>
12 return

public class Sample1 {
static public void main (String args[]) {
System.err.println ("Hello!");
System.exit (1); }
}

public class Cast {
static public void main (String args[]) {
Object x;
x = (Object) args[0];
System.out.println ("result: " + (String) x);
}}

Method void main(java.lang.String[])
0 aload_0
1 iconst_0
2 aaload
3 astore_1
4 getstatic #2 <Field java.io.PrintStream out>
7 new #3 <Class java.lang.StringBuffer>
10 dup
11 invokespecial #4 <Method java.lang.StringBuffer()>
14 ldc #5 <String "result: ">
16 invokevirtual #6 <Method java.lang.StringBuffer append(java.lang.String)>
19 aload_1
20 checkcast #7 <Class java.lang.String>
23 invokevirtual #6 <Method java.lang.StringBuffer append(java.lang.String)>
26 invokevirtual #8 <Method java.lang.String toString()>
29 invokevirtual #9 <Method void println(java.lang.String)>
32 return

Method void main(java.lang.String[])
0 aload_0
1 iconst_0
2 aaload
3 astore_1
4 getstatic #2 <Field java.io.PrintStream err>
7 new #3 <Class java.lang.String>
10 dup
11 invokespecial #4 <Method java.lang.String()>
14 ldc #5 <String "result: ">
16 invokevirtual #6 <Method java.lang.StringBuffer append(java.lang.String)>
19 aload_1
20 checkcast #7 <Class java.lang.String>
23 invokevirtual #6 <Method java.lang.StringBuffer append(java.lang.String)>
26 invokevirtual #8 <Method java.lang.String toString()>
29 invokevirtual #9 <Method void println(java.lang.String)>
32 return

public class JSR {
static public void main (String args[]) {
try {
System.out.println("hello");
} catch (Exception e) {
System.out.println ("There was an exception!");
finally {
System.out.println ("I am finally here!");
}
}}

JVML Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jsr</td>
<td>The address of the opcode of the instruction immediately following this jsr instruction is pushed onto the operand stack as a value of type returnAddress. The unsignedBranchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is (branchbyte1 &lt;&lt; 8)</td>
</tr>
<tr>
<td></td>
<td>Notes</td>
</tr>
<tr>
<td></td>
<td>The jsr instruction is used with the ret instruction in the implementation of the finally clauses of the Java programming language. Note that jsr pushes the address onto the operand stack and ret gets it out of a local variable. This asymmetry is intentional.</td>
</tr>
</tbody>
</table>

The Worst Instruction

The worst instruction in the set is the jsr instruction.

What is a secure programming language?

1. Language is designed so it cannot express certain computations considered insecure.
   A few attempt to do this: PLAN, packet filters

2. Language is designed so that (accidental) program bugs are likely to be caught by the compiler or run-time environment instead of leading to security vulnerabilities.

Java™: Programming Language

compared to C++, not to C

“$A$ simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.” [Sun95]

Java: int is 32 bits
C: int is $\geq 16$ bits

Safe Programming Languages

- Type Safety
  - Compiler and run-time environment ensure that bits are treated as the type they represent
- Memory Safety
  - Compiler and run-time environment ensure that program cannot access memory outside defined storage
- Control Flow Safety
  - Can’t jump to arbitrary addresses
  Which of these does C/C++ have?
  Is Java the first language to have them?
  No way! LISP had them all in 1960.

Java™ Safety

- Type Safety
  - Most types checked statically
  - Coercions, array assignments type checked at run time
- Memory Safety
  - No direct memory access (e.g., pointers)
  - Primitive array type with mandatory run-time bounds checking
- Control Flow Safety
  - Structured control flow, no arbitrary jumps

Malicious Code

Can a safe programming language protect you from malcode?

1. Code your servers in it to protect from buffer overflow bugs
2. Only allow programs from untrustworthy origins to run if the are programmed in the safe language
Safe Languages?
- But how can you tell program was written in the safe language?
  - Get the source code and compile it (most vendors, and all malicious attackers refuse to provide source code)
  - Special compilation service cryptographically signs object files generated from the safe language (SPIN, [Bershad96])
  - Verify object files preserve safety properties of source language (Java)

Does JVML satisfy Java™’s safety properties?
iconst_2 push integer constant 2 on stack
istore_0 store top of stack in variable 0 as int
aload_0 load object reference from variable 0

No! This code violates Java™’s type rules.

Mistyped Code
.method public static main([Ljava/lang/String;])V
  ... iconst_2 istore_0 aload_0 iconst_2 iconst_3 iadd
  .end method

Verifier error before any code runs

Runtime Error
public class Cast {
  static public void main(String args[]) {
    Object o = new Object();
    String s = (String) o;
    System.out.println(s);
    return;
  }
}

Object new #2 <Class java.lang.Object>
0 new #2 <Class java.lang.Object>
3 dup 4 invokespecial #1 <Method java.lang.Object>()> 7 astore_1 8 astore_1 9 checkcast #3 <Class java.lang.String> 12 astore_2 13 getstatic #4 <Field java.io.PrintStream out> 16 astore_2 17 invokespecial #5 <Method void println(java.lang.String)> 18 return
Bytecode Verifier

- Checks class file is formatted correctly
  - Magic number: class file starts with 0xCafEBabe
  - String table, code, methods, etc.
- Checks JVML code satisfies safety properties
  - Simulates program execution to know types are correct, but doesn’t need to examine any instruction more than once

Verifying Safety Properties

- Type safe
  - Stack and variable slots must store and load as same type
  - Only use operations valid for the data type
- Memory safe
  - Must not attempt to pop more values from stack than are on it
  - Doesn’t access private fields and methods outside class implementation
- Control flow safe
  - Jumps must be to valid addresses within function, or call/return

Charge

- PS6 will be out (electronically) on Friday
- If you would like to be assigned a partner for PS6, send me email as soon as possible