(Chris Weston's take on Orwell's 1984)

Security in Java

Week 11
CS 212 – Spring 2008

Announcements

- Project Part 3
  - We've extended the due date for Part 3
  - Part 3 code is now due Sunday, April 13, at noon

Java Class Loading

- Loading = reading in class file, verifying bytecode, integrating into the JVM
- Java class loading is lazy
  - A class is loaded and initialized when it (or a subclass) is first accessed
- Classname must match filename so class loader can find it
- Superclasses are loaded and initialized before subclasses

Class Initialization

- Prepare static fields with default values
  - 0 for primitive types
  - null for reference types
- Run static initializer: <clinit>
  - Performs programmer-defined initializations
  - Only time <clinit> is ever run
  - Only the JVM can call it

Class Initialization

```
class Staff {
  static Staff Paul = new Staff();
  static Staff Kelly = new Staff();
  static Map h = new HashMap();
  static {
    h.put(Paul, INSTRUCTOR);
    h.put(Kelly, ADMINISTRATOR);
  }
  ...
}
```

- Compiled to Staff.<clinit>

Initialization Dependencies

```
class A {
  static int a = B.b + 1;  //code in A.<clinit>
}

class B {
  static int b = 42;       //code in B.<clinit>
}
```

- Initialization of A will be *suspended* while B is loaded and initialized

Initialization Dependencies

```
class A {
  public static int a = B.b + 1;     //code in A.<clinit>
}

class B {
  public static int b = A.a + 1;     //code in B.<clinit>
}
```

Q) Is this legal Java? If so, does it halt?

A) Yes and yes

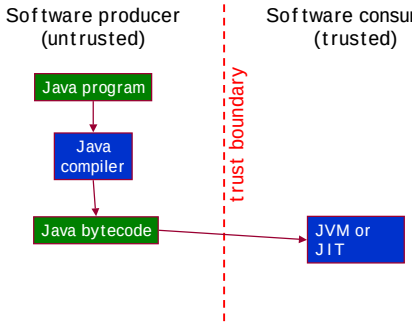
Q) So what are the values of A.a and B.b?

A) A.a = 2 B.b = 1

Object Initialization

- Object creation initiated by new
  - Sometimes implicitly (e.g., for Strings built using +)
- JVM allocates heap space for the object
  - Room for all instance (non-static) fields of the class, including inherited fields
- Instance fields prepared with default values
  - 0 for primitive types
  - null for reference types
- Object initializer: <init>(…)
  - Call to <init>(…) is explicit in the compiled code
  - Code for <init> is compiled from constructor
  - Default <init> is used, if no constructor
- First operation of <init> must be a call to the corresponding <init> of superclass
  - Either done explicitly by the programmer using super(…) or implicitly by the compiler

Mobile Code



Mobile Code

- Problem: Mobile code is *not* trustworthy!
- We often have trusted and untrusted code running together in the same virtual machine
  - Example: applets downloaded off the net and running in our browser
- Do not want untrusted code to perform critical operations (e.g., file I/O, net I/O, class loading, security management,…)
  - How do we prevent this?

Java Security Model

- Bytecode verification
  - Ensure that only legitimate bytecodes are executed by the JVM
- Secure class loading
  - Guards against substitution of malicious code for standard system classes
- Stack inspection
  - Mediates access to critical resources

Bytecode Verification

- Performed at load time
- Basis for the entire security model
  - Prevents circumvention of higher-level checks
- Enforces type safety
  - All operations are well-typed (e.g., no confusion of refs and ints)
  - Illegal data typecasts
  - Array bounds
  - Operand stack overflow, underflow
  - Consistent state over all dataflow paths
  - Private/protected/package/final annotations

Secure Class Loading

- The *class loader* is responsible for
  - Locating and fetching the class file
  - Consulting the *security policy*
  - Defining the class object with the appropriate permissions
- A *class loader* associates the following information with each class that it loads:
  - Where the code was loaded from
  - Who signed the code (if anyone)
  - Default permissions granted to the code
    - Ability to make network connections back to the originating host
    - If loaded from local file system, the ability to read files from originating directory and any subdirectories
  - Other permissions depending on local security policy

The "Primordial" Class Loader

- A class loader is a Java class—how does it get loaded?
  - There is a "primordial" class loader that bootstraps the class loading process
    - Generally written in a native language (e.g., C)
  - Base classes (i.e., java.lang) are essentially loaded by this primordial class loader

Mobile Code

- Early approach: signed applets
- Not so great
  - Everything is either *trusted* or *untrusted*, nothing in between
  - A signature can only verify an already existing relationship of trust, it cannot *create* trust
- Would like to allow untrusted code to interact with trusted code
  - Just monitor its activity somehow…

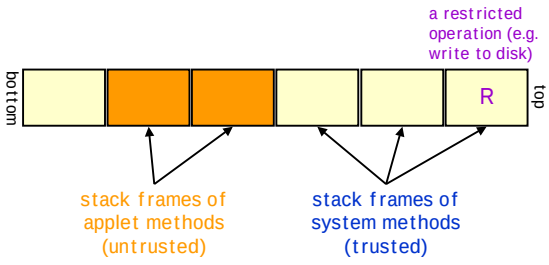
Mobile Code

Q) Why not just let trusted (system) code do anything it wants, even in the presence of untrusted code?

A) Because untrusted code calls system code to do stuff (file I/O, etc.)

System code could be operating *on behalf of* untrusted code

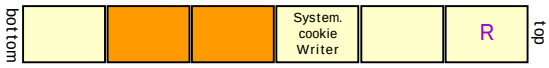
Runtime Stack



Disallow?

- Maybe we should disallow it
  - The malicious applet may be trying to erase our disk
  - It's calling system code to do that

Allow?



- Or, maybe we should allow it
  - It may just want to write a cookie
  - It called System.cookieWriter
    - System.cookieWriter "knows" it's OK to write a cookie

Allow?

- Maybe we should allow it for another reason
  - All running methods are trusted

How do we tell the difference between these scenarios?

Answer: Stack inspection!

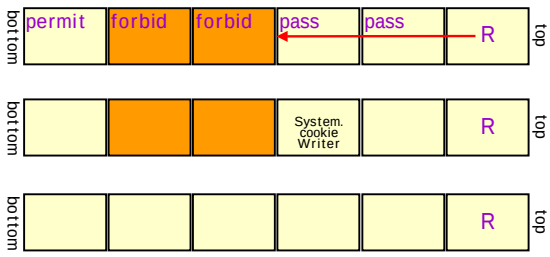
Stack Inspection

- An invocation of a trusted method, when calling another method, may either:
  - permit R on the stack above it
  - forbid R on the stack above it
  - pass permission from below (be transparent)
- An instantiation of an *untrusted* method must forbid R on the stack above it

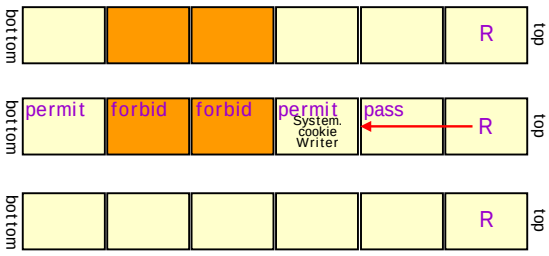
Stack Inspection

- When about to execute R, look down through the stack until we see either
  - A system method permitting R — do it
  - A system method forbidding R — don't do it
  - An untrusted method — don't do it
- If we get all the way to the bottom, do it (IE, Sun JDK) or don't do it (Netscape)

Case A: R is not executed



Case B: R is executed



Case C: R is executed