A Secure Implementation of Java Inner Classes

By
Anasua Bhowmik and William Pugh
Department of Computer Science
University of Maryland

More info at:
http://www.cs.umd.edu/~pugh/java
Motivation and Overview

- Present implementation of Java inner classes provides a security hole in order to allow inner classes access the private fields of the outer class and vice versa
- We designed a secure technique for allowing access to private fields and methods
- No need to change the JVM
- Very little overhead
- Developed a byte code transforming tool which modify the class files and make the inner classes safe
Java Inner Classes

- *Inner class* is a new feature added in Java 1.1
- *Inner classes* are classes defined as member of other class
- *Inner classes* are allowed to access the private members of the enclosing class and vice versa
- For each instance of an outer class there is a corresponding instance of the inner classes

```java
class A {
    private a;
    class B {
        private b;
        void f() {
            b = a+a;  // accessing pvt. var of A
        }
        public g(){
            B myObj = new B();
            myObj.f();
            int x = myObj.b; // accessing pvt. var of A
        }
    }
}
```
Inner Classes Aren’t Understood By JVMs

- Inner classes are implemented as a compiler transformation
- JVM do not need to understand inner classes
  - code will run on 1.0 JVM’s
- JVM prohibits access to private members from outside the class
- Compiler transforms the class, containing inner classes, to a number of non-nested classes
Implementation of Inner Classes

- Access$0() of class A has package level visibility.
- The class A$B also has package level visibility.
Security Threats with Present Implementation

- The private data members of classes get exposed through access functions
- Other classes belonging to the same package can call the access functions and tamper the private data member

```
private int m;
public void g() {
    A$B ob = new A$B();
    ob.f();
}

int access$0() {
    return m;
}
```

```
class C {
    fun() {
        A a = new A();
        ..
        int x = a.access$0();
    }
}
```

Class C and A belongs to the same package
Is This A Problem?

- Lots of Java code uses inner classes
- Using new 1.2 security model, all privileged code is put in inner classes
- Still requires attacker get inside package
- One security barrier down
  - Prefer defense in depth
- Ed Felton recommends against using current version of inner classes
New Implementation of Inner Classes

- The access to the private members are restricted only to the intended classes
- The new implementation is built on top of the current implementation
  - class files are rewritten
- No need to change the JVM
- A secret key is shared between all the classes that need access to each others private data members
  - Class B wants to access a class A’s private member \( m \)
  - invokes A’s access function
  - \( B \) passes its shared secret key to A’s access function
  - A verifies whether \( B \)’s secret key and A’s secret key are the same object
    - if yes, give access to its private variable \( m \)
    - otherwise, throw a security exception
The secret key is an object allocated dynamically during runtime.

Class A allocates an object in its static initializer and stores it in its own private static field A.sharedSecret.

Class A passes down the secret key by invoking the receiveSecretKey(A.sharedSecret) of class B.

In receiveSecretKey(Object) B stores A’s secret key in its own private static field, B.sharedSecret.

Whenever B tries to access A’s private field it passes its shared secret key for authentication.
New Implementation of Inner Classes

Initialization Phase

A passes the secret key object to B

A allocates a new object and stores it in A.sharesSecret

B stores the secret key passed by A in B.sharedSecret

B wants to access A’s private Field

B invokes A’s access method with B.sharedSecret as an argument

B passes the secret key for verification

In access method A verifies B’s secret key

A grants access if B’s secret key matches with A’s

A throws security exception if secret keys not match
Class A {
    static private final Object sharedSecret = new Object();
    static {
        A$B.receiveSecretForA(sharedSecret);
    }
    private int x;
    int access$1(Object secretForA) {
        if (secretForA != sharedSecret) throw new SecurityException();
        return x;
    }
}

Class A$B {
    private A this$0;
    static private Object sharedSecret;
    static void receiveSecretForA(Object secretKey) {
        if (sharedSecret != null) throw new VerifyError();
        sharedSecret = secretKey;
    }
    … invoke this$0.access$1(sharedSecret)…
}
Advantages of the New Implementation

- Access is permitted only to the desired classes
- No need to change the existing JVMs
- The secret key value is a pointer to memory, allocated dynamically
  - Absolutely impossible to forge
- The additional overhead for initialization and validation of the secret keys are small
- Very small increase in the size of the class files
Overhead Due to Modification

- For each class allowing/needing access
  - One static field
- For each set of objects needing mutual access
  - One object created
- All initializations are done in static initializer
- One additional argument in each access method
- Few additional instructions are executed for each access call to
  - pass the extra argument
  - verify the secret key
A Rewriting Tool For Jar Files

- Developed a tool to transform the byte codes
- Takes a *jar* file, examines the class files and finds out the sets of classes which need mutual access
- modify all the class files which are either defining access$\$ methods or invoking access$\$ methods
- All the classes in the jar file are made safe in the presence of inner classes
- Used our tool to modify several *jar* files - rt.jar, swing.jar etc.
Experimental Result for swing.jar

Static Evaluation:

- % increase in the code size: 2.9%
- # of class files in swing.jar: 1498
- # of inner classes: 898
- # of inner classes needing access: 139
- # of objects created: 53
- # of new fields added: 195
- # of access methods: 145
- # of places access methods are invoked: 439
Experimental Result for swing.jar

Runtime Performance

For a trial run of SwingSet demo, which tests all the functionalities

Total number of calls to access$ functions - 46,638
Total user time - 59.44 sec
Total system time - 3.91 sec

Note: The user and system times are comparable when we run the demo with original swing.jar file. Although it is not possible to run the demo exactly the same way and compare precisely.
Even Better Security

- Before A gives the secret to A$B
  - Check signatures on A$B imply the signatures on A
- Prevents situation where an attacker tries to combine a signed version of A with a modified (and unsigned) version of A$B
Conclusion

- Designed a new implementation for inner classes to fix the security hole of the current implementation
- Little additional overhead
  - regarding both code size and execution time
- Implemented a byte code rewriter to incorporate the changes by transforming the byte code
- Can be implemented in the compiler
- Can extend this idea to have friend classes like C++