Object Serialization in Java

Java's object serialization mechanism is a convenient way to store Java objects on disk. It is also tempting to use this mechanism as a building block for network protocols. This mini-lecture describes the pitfalls of using object serialization in this way, and offers some suggestions on how you could begin to use serialization in your own networking protocols.

Disclaimer: These notes are not a complete discussion of the problems inherant in object serialization, and are simply a few problems that the T.A.'s and professors have noticed over the last week. We do not know of a comprehensive source that describes how Java's object serialization mechanism can be used sercurely.

1 What Can Go Wrong

The biggest problem with using serializable over a network connection is that readObject() returns an instantiated object of a type controlled by the sender of the message. The implementation of this object must be somewhere on your classpath for the deserialization to succeed.

Java ships with many classes that implement serializable, and many third-party libraries also implement serializable. Worse, Java allows objects to implement their own deserialize methods. If you use the default ObjectInputStream, the creator of the objects you are deserializing can trick you into running any code in any of these custom deserialize methods.

The best you can do without subclassing ObjectInputStream is to immediately cast objects that it returns to the expected type, and to only accept instances of final classes over the network.

You might be able to do a bit better. ObjectInputStream has a protected method called readClassDescriptor() that you could imagine overriding as follows:

```
------------   Begin SaferObjectInputStream --------------------------
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectStreamClass;
import java.util.Set;

public class SaferObjectInputStream extends ObjectInputStream {
   private Set whitelist;
   public SaferObjectInputStream(InputStream in, Set whitelist)
    throws IOException {
      super(in);
```

```
      this.whitelist = whitelist;
   }
   protected ObjectStreamClass readClassDescriptor()
    throws IOException, ClassNotFoundException {
      ObjectStreamClass ret = super.readClassDescriptor();
      if(!whitelist.contains(ret.forClass())) {
         throw new ClassNotFoundException
            ("Unexpected class encountered on input stream.");
      }
      return ret;
   }
}

------------   End SaferObjectInputStream --------------------------
```

The problem with this is that we don't know when readClassDescriptor is called or what happens when the ClassNotFoundException is thrown. It's unclear whether ObjectInputStream is designed to be used in this way.

Even if we could restrict the adversary to a whitelisted set of classes, the sender of the message could send you malformed objects, causing your code to throw exceptions in unexpected places.

For example, Sun's String implementation has three fields:

```
class String {
 private char[] buf;
 private int start;
 private int stop;
 // more stuff here....
}
```

start and stop index into buf, and describe which region of buf represents the string object. This lets Sun implement substring() without copying:

```
public String substring(int start, int stop) {
  assert(stop-start <= this.start-this.stop && stop-start >= 0);
  // The next line calls a private constructor
  return new String(buf, this.start+start, this.start+stop);
}
```

Anyway, imagine what would happen if start or stop were initialized to negative numbers or past the end of buf. When you call a method on such a String, Java's internal array bounds checks could throw an ArrayIndexOutOfBounds exception, even though your code would not throw that exception if it were given a valid String object.

Therefore you need to be prepared to handle strange exceptions throughout your codebase or you need to make sure that the necesssary consistency checks are executed against each object as it is being deserialized.

We covered the problem with cycles of objects[1] in lecture, (ObjectInputStream.readUnshared() will return cycle-free objects) but cycles are a special case of a more general problem. You need to validate each object

[1] Imagine what would happen if you deserialized a binary tree that contained unexpected cycles!

graph that readObject() returns to make sure that it is something that could occur during normal operation. One common practice (which may have problematic corner cases) is to implement a special version of clone() that copies objects by only calling public methods that the two objects provide. That way, you know that clone()'s output is an instance that you could have created, but you still don't know whether clone() itself has strange side effects or if clone()'s output actually makes sense.

Worse, imagine what happens with data structures that have bad worst-case performance. You could imagine writing code that relies upon TreeSet's logarithmic lookup time, but having an adversary pass you a poorly balanced tree, causing code that once ran in O(log n) time to run in O(n) time... This could turn O(n log n) algorithms into O(n2) algorithms. You can imagine worse problems with other data structures...

If you do a good job of convincing us that your deserialization code is secure, (ie: Implement whitelisting, audit the classes that you use, and convince us that the method that deserializes each class is secure) it will be acceptable, but there are a lot of pitfalls that stem from using this approach, especially if the security of your system depends on internal implementation details of the java api!