Look-ahead Java deserialization

How to secure deserialization from untrusted input without using encryption or sealing

Pierre Ernst

January 15, 2013

When Java™ serialization is used to exchange information between a client and a server, attackers can try to replace the legitimate serialized stream with malicious data. This article explains the nature of this threat and describes a simple way to protect against it. Find out how to stop the deserialization process as soon as an unexpected Java class is found in the stream.

Java serialization enables developers to save a Java object to a binary format so that it can be persisted to a file or transmitted over a network. Remote Method Invocation (RMI) uses serialization as a communication medium between a client and a server. Several security problems can arise when a service accepts binary data from a client and deserializes the input to construct a Java instance. This article focuses on one of them: An attacker could serialize an instance of another class and send it to the service. The service would then deserialize the malicious object and most probably cast it to the legitimate class the service is expecting, causing an exception to be thrown. However, that exception might come too late to ensure that the data is secure. This article explains why and shows how to implement a secure alternative. (See the Other deserialization pitfalls sidebar for a brief overview of other security issues relating to Java deserialization.)

Other deserialization pitfalls

Deserialization is subject to three additional threats:

• An attacker could eavesdrop on the communication and obtain potentially sensitive data. Transport Layer Security (TLS) can be used to prevent this type of attack.
• A malicious user could tamper with data that was legitimately serialized by the client application and change values to subvert the service's business logic. As with other types of services, input validation must be applied at the server, even if the same validation has already taken place at the client. Object sealing can also be an effective countermeasure in this scenario.
• An attacker can set private members of the object, which might not be the behavior that the developers intended. The attacker might be able to change the object's internal state using that technique. Marking such members transient can be part of the solution.
Further discussion of these issues and countermeasures is outside the scope of this article.

Vulnerable classes

Your service shouldn't deserialize objects of arbitrary class. Why not? The short answer is: because you likely have vulnerable classes in the server's classpath that an attacker can leverage. These classes contain code that let the attacker cause a denial-of-service condition or — in extreme cases — to inject arbitrary code.

You might believe that this kind of attack is impossible, but consider how many classes can be found in the classpath of a typical server. They include not only your own code, but also the Java Class Library, third-party libraries, and any middleware or framework libraries. Additionally, the classpath might change over an application's lifetime or be modified in response to environmental changes to the system that extend beyond a single application. When trying to leverage such a weakness, an attacker can combine several operations by sending multiple serialized objects.

I should emphasize that the service will deserialize a malicious object only if:

- The malicious object's class exists in the server's classpath. The attacker cannot simply send a serialized object of any class, because the service will be unable to load the class.
- The malicious object's class is either serializable or externalizable. (That is, the class on the server must implement either the java.io.Serializable interface or the java.io.Externalizable interface.)

Also, the deserialization process populates the object tree by copying data from the serialized stream without calling the constructor. So an attacker can't execute Java code residing inside the constructor of the serializable object class.

But the attacker has other ways of executing some code on the server. Whenever the JVM deserializes an object of a class that implements one of the following three methods, it calls the method and executes the code inside it:

- The readObject() method, typically used by developers when standard serialization cannot be used, such as when a transient member needs to be set.
- The readResolve() method, typically used to serialize singleton instances.
- The readExternal() method, used for externalizable objects.

So if you have classes in your classpath that use any of these methods, you must be aware that an attacker can call the methods remotely. This kind of attack has been used in the past to break out of the Applet sandbox (see Related topics); the same technique can also be applied against a server.

Read on to see how to allow deserialization only of the class (or classes) that you expect for your service.
Java serialization binary format

Whitelisting

Even if you're absolutely certain that your service is immune to the attack discussed in this article, remember that input validation against a list of known good values (whitelisting) is always part of good security practices.

After an object is serialized, the binary data contains both metadata (information about the structure of the data, such as class name, number of members, and type of members) and the data itself. I'll use a simple Bicycle class as an example. The class, shown in Listing 1, contains three members (id, name, and nbrWheels) and their corresponding setters and getters:

Listing 1. The Bicycle class

```java
package com.ibm.ba.scg.LookAheadDeserializer;

public class Bicycle implements java.io.Serializable {
    private static final long serialVersionUID = 5754104541168329730L;
    private int id;
    private String name;
    private int nbrWheels;

    public Bicycle(int id, String name, int nbrWheels) {
        this.id = id;
        this.name = name;
        this.nbrWheels = nbrWheels;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public int getNbrWheels() {
        return nbrWheels;
    }

    public void setNbrWheels(int nbrWheels) {
        this.nbrWheels = nbrWheels;
    }
}
```

After an instance of the class presented in Listing 1 has been serialized, the data stream looks like Listing 2:
Listing 2. Serialized data stream for the Bicycle class

```
000000: AC ED 00 05 73 72 00 2C 63 6F 6D 2E 69 62 6D 2E |········com.ibm.|
000016: 62 61 2E 73 63 67 2E 4C 6F 6F 6B 41 68 65 61 64 |ba.scg.LookAhead|
000032: 79 63 6C 65 4F DA AF 97 F8 CC C0 DA 02 00 03 49 |ycle............I|
000048: 00 02 69 64 49 00 09 6E 62 72 57 68 65 65 6C 73 |··idI··nbrWheels|
000064: 6C 00 04 6E 61 6D 65 74 00 12 4C 6A 61 76 61 2F |L···name···Ljava/|
000080: 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 78 70 00 00 |lang/String;····|
000112: 00 00 00 00 00 01 74 00 08 55 6E 69 63 79 63 6C |········Unicycl|
000128: 65                                              |e|
```

By applying the standardized Object Serialization Stream protocol to this data (see Related topics), you can see the details of the serialized object, shown in Listing 3:

### Listing 3. Details of the serialized Bicycle object

STREAM_MAGIC (2 bytes) 0xACED
STREAM_VERSION (2 bytes) 5

```
newObject
  newClassDesc
    TC_CLASSDESC (1 byte) 0x72
      className
        length (2 bytes) 0x2C = 44
        text (59 bytes) com.ibm.ba.scg.LookAheadDeserializer.Bicycle
      serialVersionUID (8 bytes) 0x4FDAAF97F8CCC0DA = 575416451168320730
      classDescFlags (1 byte) 0x02 = SC_SERIALIZABLE
      fields
        count (2 bytes) 3
        field[0]
          primitiveDesc
            prim_typecode (1 byte) I = integer
            fieldName
              length (2 bytes) 2
              text (2 bytes) id
        field[1]
          primitiveDesc
            prim_typecode (1 byte) I = integer
            fieldName
              length (2 bytes) 9
              text (9 bytes) nbrWheels
        field[2]
          objectDesc
            obj_typecode (1 byte) L = object
            fieldName
              length (2 bytes) 4
              text (4 bytes) name
            className1
              TC_STRING (1 byte) 0x74
                length (2 bytes) 0x12 = 18
                text (18 bytes) Ljava/lang/String;
            classAnnotation
              TC_ENDBLOCKDATA (1 byte) 0x78
          superClassDesc
            TC_NULL (1 byte) 0x70
          classdata[]
            classdata[0] (4 bytes) 0 = id
            classdata[1] (4 bytes) 1 = nbrWheels
            classdata[2]
              TC_STRING (1 byte) 0x74
              length (2 bytes) 8
```
You can see from Listing 3 that this serialized object is a
com.ibm.ba.scg.LookAheadDeserializer.Bicycle, that its ID is zero, that it has one wheel, and
that it's a unicycle.

The important point here is that the binary format contains headers of a sort that enable you to
perform input validation.

Look-ahead class validation

As you can see in Listing 3, when the stream is read, the class description of the serialized object
appears before the object itself. This structure enables you to implement your own algorithm to
read the class description and decide whether to continue reading the stream, depending on the
class name. Fortunately, you can do this easily by using a hook Java provides that's normally used
for custom class loading — namely, overriding the resolveClass() method. This hook fits the bill
perfectly for providing custom validation, because you can use it to throw an exception whenever
the stream contains an unexpected class. You need to subclass java.io.ObjectInputStream and
override the resolveClass() method. Listing 4 uses this technique to allow only instances of the
Bicycle class to be deserialized:

Listing 4. Custom validation hook

```java
package com.ibm.ba.scg.LookAheadDeserializer;
import java.io.IOException;
import java.io.InputStream;
import java.io.InvalidClassException;
import java.io.ObjectInputStream;
import java.io.ObjectStreamClass;
import com.ibm.ba.scg.LookAheadDeserializer.Bicycle;

public class LookAheadObjectInputStream extends ObjectInputStream {
    public LookAheadObjectInputStream(InputStream inputStream)
        throws IOException {
        super(inputStream);
    }

    @Override
    protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException,
        ClassNotFoundException {
        if (!desc.getName().equals(Bicycle.class.getName())) {
            throw new InvalidClassException(
                "Unauthorized deserialization attempt",
                desc.getName());
        }
        return super.resolveClass(desc);
    }
}
```

By calling the readObject() method on your com.ibm.ba.scg.LookAheadDeserializer instance,
you prevent an unexpected object from being deserialized.
As a demonstration, Listing 5 serializes two objects — an instance of the expected class (com.ibm.ba.scg.LookAheadDeserializer.Bicycle) and an unexpected object (a java.lang.File instance) — then tries to deserialize them using the custom validation hook from Listing 4:

**Listing 5. Deserializing two objects using a custom validation hook**

```java
package com.ibm.ba.scg.LookAheadDeserializer;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import com.ibm.ba.scg.LookAheadDeserializer.Bicycle;

public class LookAheadDeserializer {
    private static byte[] serialize(Object obj) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(obj);
        byte[] buffer = baos.toByteArray();
        oos.close();
        baos.close();
        return buffer;
    }

    private static Object deserialize(byte[] buffer) throws IOException, ClassNotFoundException {
        ByteArrayInputStream bais = new ByteArrayInputStream(buffer);
        // We use LookAheadObjectInputStream instead of InputStream
        ObjectInputStream ois = new LookAheadObjectInputStream(bais);
        Object obj = ois.readObject();
        ois.close();
        bais.close();
        return obj;
    }

    public static void main(String[] args) {
        try {
            // Serialize a Bicycle instance
            byte[] serializedBicycle = serialize(new Bicycle(0, "Unicycle", 1));

            // Serialize a File instance
            byte[] serializedFile = serialize(new File("Pierre Ernst"));

            // Deserialize the Bicycle instance (legitimate use case)
            Bicycle bicycle0 = (Bicycle) deserialize(serializedBicycle);
            System.out.println(bicycle0.getName() + " has been deserialized.");

            // Deserialize the File instance (error case)
            Bicycle bicycle1 = (Bicycle) deserialize(serializedFile);
        } catch (Exception ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```
When you run the application, the JVM throws an exception before trying to deserialize the `java.lang.File` object, as shown in Figure 1:

**Figure 1. Application output**

```
Terminate the Java deserialization attempt: java.io.File
```

**Conclusion**

This article has shown you how to stop the Java deserialization process as soon as an unexpected Java class is found on the stream, without needing to perform encryption, sealing, or simple input validation on the members of the newly deserialized instance. See Download to get the full source code for the examples.

Remember that the entire object tree (the root object with all its members) gets constructed during deserialization. In more-complex configurations, you might need to allow more than one class to be deserialized.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code for this article's examples</td>
<td>look-ahead-java-deserialization.src.zip</td>
<td>4KB</td>
</tr>
</tbody>
</table>
Related topics

- "5 things you didn't know about... Java Object Serialization": (developerWorks, April 2010): This article covers some of the security issues related to serialization.
- **Java Remote Method Protocol**: The JRMP, a protocol for Java-to-Java remote calls, uses serialization.
- **Java Security, Chapter 2, Section 2.1.1**: Read a discussion on the security implications of serialization/deserialization.
- **An interesting case of JRE sandbox breach (CVE-2012-0507)**: The kind of attack described in this article has been used to break out of the Applet sandbox.
- **CWE-502: Deserialization of Untrusted Data**: MITRE Common Weakness reference for the kind of attack described in this article.
- Known vulnerabilities relating to Java serialization:
  - **CVE-2004-2540**: `readObject` in JRE allows remote attackers to cause a denial of service using crafted serialized data.
  - **CVE-2008-5353**: The JRE does not properly enforce context of `ZoneInfo` objects during deserialization, which allows remote attackers to run untrusted applets and applications in a privileged context, as demonstrated by "deserializing calendar objects."
  - **CVE-2010-0094**: Unspecified vulnerability in the JRE allows remote attackers to affect confidentiality, integrity, and availability through unknown vectors related to deserialization of `RMIConnectionImpl` objects, which allows remote attackers to call system-level Java functions using the `ClassLoader` of a constructor that is being deserialized.
  - **CVE-2011-3521**: Unspecified vulnerability in the JRE allows remote untrusted Java Web Start applications and untrusted Java applets to affect confidentiality, integrity, and availability using unknown vectors related to deserialization.
  - **CVE-2012-0505**: Unspecified vulnerability in the JRE allows remote untrusted Java Web Start applications and untrusted Java applets to affect confidentiality, integrity, and availability using unknown vectors related to serialization.