Defending against Java Deserialization Vulnerabilities

Bay Area OWASP Meetup - September 2016

Luca Carettoni - @lucacarettoni
Agenda

This talk is about defense and how to protect your application against this new old class of vulnerabilities.

- Intro to Java Deserialization bugs
- A real-life bug (SJWC serialized object injection via JSF view state)
- Discovery
- Defense
Intro to Java Deserialization bugs
From object graph data to byte stream
Serialization in Code

// Instantiate the Serializable class
String myPreso = "OWASP Bay Area";

// Write to disk
FileOutputStream fileOut = new FileOutputStream("serial.data");

// Write object
ObjectOutputStream objOut = new ObjectOutputStream(fileOut);
objOut.writeObject(myPreso);
Deserialization in Code

// Read from disk
FileInputStream fileIn = new FileInputStream("serial.data");

// Read object
ObjectInputStream objIn = new ObjectInputStream (fileIn);
String myPreso = (String) objIn.readObject();
Deserialization in Bytecode

[...]

aload ois

invokevirtual Object ObjectInputStream.readObject()

checkcast String

[...]
Callback methods

- Developers can override the following methods to customize the deserialization process
  - readObject()
  - readResolve()
  - readObjectNoData()
  - validateObject()
  - finalize() Invoked by the Garbage Collector
What if....

1. A remote service accepts Java serialized objects
2. In the classpath of the remote application, there are unrelated classes that are Serializable AND implement one of the callbacks
3. The callback’s method implements something interesting*

* File I/O operations, system commands, socket operations, etc.
Unlikely?

//org.apache.commons.fileupload.disk.DiskFileItem
Private void readObject(ObjectInputStream in) {
703   in.defaultReadObject();
704
705   OutputStream output = getOutputStream();
...
709   FileInputStream input = new FileInputStream(dfosFile);
710   IOUtils.copy(input, output);
711   dfosFile.delete();
...

The Forgotten Bug Class @matthias_kaiser™

2005 - Marc Schonefeld

Java & Secure Programming
(Bad Examples found in JDK)

Marc Schönefeld, University of Bamberg
Illegalaccess.org

2015 - Steve Breen

What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common?
This Vulnerability.
By @breenmachine

And many more...
A real-life bug, back from 2010
Sun Java Web Console serialized object injection via JSF view state
Sun Java Web Console

README - SJWC_3.0

“The Sun Java (TM) Web Console is a web application that provides a single point on entry for many of Sun's systems management applications. The console application provides a single-sign on capability and a secure home page for many of Solaris”
JSF ViewState

- JSF ViewState uses Java deserialization to restore the UI state

HTML Page

```html
<form>
  <input type="hidden" name="javax.faces.ViewState" value=""/>
</form>
```
ViewState saved client-side only
  ○ javax.faces.STATE_SAVING_METHOD="client" before SJWC < 3.1

No encryption
A good bug

- Attractive target, as SJWC was the admin web interface for Solaris
- At the time of discovery (Jan 2010), I created a Proof-of-Concept using a known gadget based on Hashtable collisions (Crosby & Wallach, 2003)
  - https://www.ikkisoft.com/stuff/SJWC_DoS.java
- Back then, I had no idea about the infamous Apache Common Collections gadget (Gabriel Lawrence, Chris Frohoff)
  - /opt/sun/webconsole/private/container/shared/lib/commons-collections.jar
- However, I was able to leverage an Expression Language (EL) Injection-like to perform arbitrary file read
- Soon after, SJWC started using server-side ViewState
  - “Beware of Serialized GUI Objects Bearing Data” July 2010, Black Hat Vegas
In practice
Discovery
Code Review - Entry Points

Look for occurrences of:

- `java.io.ObjectInputStream.readObject()`
- `java.io.ObjectInputStream.readUnshared()`

And perform manual review to determine whether they use user-supplied data

$ egrep -r "readObject\|readUnshared\"
Code Review - Gadgets

- This is the interesting (and complex) part of exploiting Java deserialization vulnerabilities.
- As a defender, assume that there are multiple game-over gadgets available in the classpath.
  - For example, SJWC uses 58 dependency JARs.

- If you want to learn more on how to discover gold and silver gadgets:
  - Marshalling Pickles - Gabriel Lawrence, Chris Frohoff
  - Java Deserialization Vulnerabilities, The Forgotten Bug Class - Matthias Kaiser
  - Surviving the Java serialization apocalypse - Alvaro Muñoz, Christian Schneider
Discovery with no code...

- Decompile :)  
- Magic bytes in the network traffic
  - 0xAC 0xED
  - rO0
  - FvzFgDff9
  - ...
- Passive and active tools
  - https://github.com/johndekroon/serializekiller
  - <--ADD your favourite web scanner vendor HERE-->
Defense
Things that do NOT work

- Patching Apache Commons
- Removing dependencies from the classpath
- Black-listing only
- Using a short-lived Java Security Manager during deserialization
Your best option. All other mitigations are suboptimal.

Do not use serialization when receiving untrusted data.

It’s 2016, there are better options.
Option #1 - Add authentication

- Add a layer of authentication to ensure that Java serialization can be invoked by trusted parties only
  - At the network layer, using client-side TLS certs
  - At the application layer, encryption/signing of the payload

<table>
<thead>
<tr>
<th>Pro</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network layer solutions can be implemented with no application changes (e.g. stunnel)</td>
<td>Additional operational complexity</td>
</tr>
<tr>
<td></td>
<td>If enc/dec is implemented by the application, secure keys management is crucial</td>
</tr>
<tr>
<td></td>
<td>Trusted parties can still abuse the application</td>
</tr>
</tbody>
</table>
Option #2 - Use Java Agent-based solutions

- Install a Java Agent solution to perform JVM-wide validation (blacklisting/whitelisting)
  - https://github.com/Contrast-Security-OSS/contrast-rO0
  - https://github.com/kantega/notsoserial

<table>
<thead>
<tr>
<th>Pro</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>- No application changes</td>
<td>- Performance hit</td>
</tr>
<tr>
<td>- Easy to deploy and use</td>
<td>- In certain environment, not usable (e.g. software engineer with no access to the underlying JVM container)</td>
</tr>
</tbody>
</table>
Option #3 - Use safe ObjectInputStream implementation

- Replace calls to ObjectInputStream with calls to a safe implementation
  - Based on look-ahead techniques
  - [https://github.com/ikkisoft/SerialKiller](https://github.com/ikkisoft/SerialKiller)
  - [https://github.com/Contrast-Security-OSS/contrast-rO0](https://github.com/Contrast-Security-OSS/contrast-rO0) (SafeObjectInputStream)

<table>
<thead>
<tr>
<th>Pro</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Full control for developers</td>
<td>● Requires re-factoring</td>
</tr>
<tr>
<td></td>
<td>● To be bulletproof*, whitelisting must be used</td>
</tr>
<tr>
<td></td>
<td>(which requires profiling, good understanding of the app)</td>
</tr>
</tbody>
</table>

* Still affected by DoS gadgets
# Mitigations in real-life

<table>
<thead>
<tr>
<th>Vendor / Product</th>
<th>Type of Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlassian Bamboo</td>
<td>Removed Usage of Serialization</td>
</tr>
<tr>
<td>Apache ActiveMQ</td>
<td>LAOIS Whitelist</td>
</tr>
<tr>
<td>Apache Batchee</td>
<td>LAOIS Blacklist + optional Whitelist</td>
</tr>
<tr>
<td>Apache JCS</td>
<td>LAOIS Blacklist + optional Whitelist</td>
</tr>
<tr>
<td>Apache openjpa</td>
<td>LAOIS Blacklist + optional Whitelist</td>
</tr>
<tr>
<td>Apache Owlb</td>
<td>LAOIS Blacklist + optional Whitelist</td>
</tr>
<tr>
<td>Apache TomEE</td>
<td>LAOIS Blacklist + optional Whitelist</td>
</tr>
</tbody>
</table>

Full credit to Alvaro Muñoz and Christian Schneider
SerialKiller is an easy-to-use look-ahead Java deserialization library to secure application from untrusted input.

https://github.com/ikkisoft/SerialKiller
How to protect your application with SerialKiller

1. Download the latest version of the SerialKiller's Jar
   a. This library is also available on Maven Central
2. Import SerialKiller's Jar in your project
3. Replace your deserialization ObjectInputStream with SerialKiller
4. Tune the configuration file, based on your application requirements
// Read from disk
FileInputStream fileIn = new FileInputStream("serial.data");

// Read object
ObjectInputStream objIn = new ObjectInputStream(fileIn);
String myPreso = (String) objIn.readObject();
In practice 2/2

// Read from disk
FileInputStream fileIn = new FileInputStream("serial.data");

// Read object
ObjectInputStream objIn = new SerialKiller(fileIn, "/etc/sk.conf");
String myPreso = (String) objIn.readObject();
SK’s configuration 1/2

SerialKiller config supports the following settings:

- **Refresh**: The refresh delay in milliseconds, used to hot-reload the configuration file
- **BlackList**: A Java regex to define malicious classes
  - Provides a default configuration against known gadgets
- **WhiteList**: A Java regex to define classes used by your application
- **Profiling**: To trace classes being deserialized
- **Logging**: Java’s core logging facility
SK’s configuration 2/2

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <refresh>6000</refresh>
    <mode>
        <profiling>true</profiling>
    </mode>
    <logging>
        <enabled>true</enabled>
        <logfile>/tmp/serialkiller.log</logfile>
    </logging>
    <blacklist>
        <!-- ysoserial's Spring1 payload -->
        <regexp>org\.springframework\.beans\.factory\.ObjectFactory\$</regexp>
    </blacklist>
    <whitelist>
        <regexp>.*</regexp>
    </whitelist>
</config>
```
SerialKiller v0.4 Demo
Thanks!

Luca Carettoni - @lucacarettoni