nccgroup

NCC Group Whitepaper

Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS)

June 15, 2017

Prepared by
Robert C. Seacord

Abstract
Java Serialization is an important and useful feature of Core Java that allows developers to transform a graph of Java objects into a stream of bytes for storage or transmission and then back into a graph of Java objects. Unfortunately, the Java Serialization architecture is highly insecure and has led to numerous vulnerabilities, including remote code execution (RCE) and denial-of-service (DoS) attacks. Any Java program that deserializes a stream is susceptible to such vulnerabilities unless proper mitigations are taken. One such mitigation strategy is look-ahead deserialization or look-ahead object input streams (LAOIS). This whitepaper examines Java deserialization vulnerabilities and evaluates various LAOIS solutions including JDK Enhancement Proposal (JEP) 290.

Table of Contents

1 Introduction

Java Serialization was introduced in JDK 1.1. It is an important and useful feature of Core Java that allows developers to transform a graph of Java objects into a stream of bytes for storage or transmission and then back into a graph of Java objects. Unfortunately, the Java serialization architecture is highly insecure and has led to numerous vulnerabilities, including remote code execution (RCE) and denial-of-service (DoS) attacks. Any Java program that deserializes a stream is susceptible to such vulnerabilities unless proper mitigations are taken. The following code from Java Object Serialization Specification [JOSS 2010] that deserializes an object stream is vulnerable to attack:

```
// Deserialize a string and date from a file.
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

The inherent problem with Java deserialization is the underlying callback architecture that makes it vulnerable to attacks. As shown in the code above, the application code invokes the `readObject` method of an `ObjectInputStream` to read an object from a stream. The stream may include any objects, not just the anticipated `String` and `Date` objects. If the object read is not a `String` and `Date` object, the cast operation will result in a `ClassCastException` or `ClassNotFoundException` if no definition for the class with the specified name could be found. Unfortunately, by the time the type checking happens, platform code has already executed significant logic that could easily result in a successful exploit.

Deserialization of untrusted data has proven to be almost universally dangerous regardless of language, platform, or serialization format. This class of vulnerability has been widely recognized by the security community for many years and is described by CWE-502: Deserialization of Untrusted Data[1] and OWASP vulnerability classifications Deserialization of Untrusted Data[2] and Object Injection.[3]

Security researchers have demonstrated a wide variety of attacks against Java deserialization code that takes advantage of executing code prior to the type check. This code is often referred to as *gadgets* because they are similar to gadgets in return-oriented programming [Shacham 2007]. Gadgets consist of existing, executable code present in the vulnerable processes that can be maliciously repurposed by an attacker. In the case of Java deserialization vulnerabilities, this code is executed when an object is deserialized.

Specific examples of such gadgets include:

- Apache Commons Collection [Lawrence 2015]
- A proof-of-concept gadget that only uses classes included in JRE versions 7u21 [Frohoff 2016].
- A pure JRE 8 RCE Deserialization gadget[4] discovered by Alvaro Muñoz.
- SerialDOS[5] developed by Wouter Coekaerts.

There are many others; these are simply a few of the more interesting gadgets. Java deserialization vulnerabilities are further exacerbated by the large number of systems that rely on it including: Remote Method Invocation (RMI), Java Management Extension (JMX), and the Java Messaging System (JMS).

[1] https://cwe.mitre.org/data/definitions/502.html
[2] https://www.owasp.org/index.php/Deserialization_of_untrusted_data
[3] https://www.owasp.org/index.php/PHP_Object_Injection
[4] https://github.com/pwntester/JRE8u20_RCE_Gadget
[5] https://gist.github.com/coekie/a27cc406fc9f3dc7a70d

2 Deserialization

The most significant security risks (of executing gadgets specified by an attacker) occur during the deserialization process. This section examines the deserialization process to better understand the mechanics of deserialization and associated exploits. The deserialization process is defined in Chapter 3 of the *Java Object Serialization Specification* [JOSS 2010].

Class `ObjectInputStream` implements object deserialization. It maintains the state of the stream including the set of objects already deserialized. Its methods allow primitive types and objects to be read from a stream written by `ObjectOutputStream`. It restores the object and its references from the stream.

The `readObject` method is used to deserialize an object from the stream. It reads from the stream to reconstruct an object. If the object in the stream is a `Class` its `ObjectStreamClass` descriptor is read, and the corresponding `Class` object returned. The `ObjectStreamClass` contains the name and `serialVersionUID` of the class.

If the class descriptor is a dynamic proxy class, the `resolveProxyClass` method is called on the stream to get the local class for the descriptor. If the class descriptor is not a dynamic proxy class, the `resolveClass` method is called on the stream to get the local class.

If the class cannot be resolved, a `ClassNotFoundException` is thrown. By default, the JVM walks the stack and uses the first non-bootstrap class loader it finds to locate classes. RMI can also use a remote codebase. The precise semantics of loading are specified in Chapter 5 of The Java Virtual Machine Specification, Java SE 8 Edition [JVMS 2015].

If the object in the stream is not a `String`, an array, or an `enum` constant, the `ObjectStreamClass` of the object is read from the stream. The local class for that `ObjectStreamClass` is retrieved. If the class is not serializable or externalizable (or an `enum` type) an `InvalidClassException` is thrown.

The retrieved local class is then instantiated and, for serializable objects, the no-argument constructor for the first non-serializable supertype is executed. For externalizable objects, the no-argument constructor for the class is run and then the `readExternal` method is called to restore the contents of the object.

This is the first opportunity an attacker has to run code associated with the supertype of the object being deserialized. This code could theoretically provide a gadget, especially because the supertype does not need to be a serializable type. However, the risk of a no-argument constructor providing a gadget is low because a non-serializable superclass is initialized with default values.

Object fields for serializable classes are initialized to the default value appropriate for its type. Each field of each object is restored by calling the corresponding class-specific `readObject` methods. The `readObject` method to be called back during deserialization and is widely-documented as a source for gadgets, for example, by OWASP.[6]

If a class-specific `readObject` method is not defined, the `defaultReadObject` method is called. The `defaultReadObject` method is a public method that reads the non-static and non-transient fields of the current class from a stream and can be consequently subclassed. This provides an additional opportunity for creating a gadget. Field initializers and constructors are not executed for serializable classes during deserialization.

For objects of externalizable classes, the no-argument constructor for the class is run and then the `readExternal` method is called to restore the contents of the object.

If the class of the object defines a `readResolve` method, the method is called to allow the object to re-

[6] https://www.owasp.org/index.php/Deserialization_of_untrusted_data

place itself. If previously enabled by `enableResolveObject`, the `resolveObject` method is called to allow subclasses of the stream to examine and replace the object. If the original object was replaced, the `resolveObject` method is called with the replacement object. The replacement object is then returned from `readObject`. The `resolveObject` method can also provide gadgets.

The `registerValidation` method is used to register a callback that is invoked when the entire object graph has been restored but before the object is returned to the original caller of `readObject`. The object to be validated must support the `ObjectInputValidation` interface and implement the `validateObject` method. Again, this is another opportunity for gadgets. Furthermore, the `validateObject` method cannot be used to mitigate deserialization attacks because it is called after deserialization has completed.

In addition to the callback mechanisms implemented as part of the serialization process, classes may also implement `finalize` methods. Finalizers are invoked by the garbage collector on an object when garbage collection determines that there are no more references to the object. Finalizers also offer an opportunity to create gadgets. These gadgets may be potentially more dangerous because the `AccessControlContext` installed when the finalizer is executed can be more permissive than the `AccessControlContext` installed during deserialization. For example, the `finalize` method of `org.apache.commons.fileupload.disk.DiskFileItem` allows an attacker to delete a file upon deserialization, as shown by the following method:

```
@Override
protected void finalize() {
  File outputFile = dfos.getFile();
  if (outputFile != null && outputFile.exists()) {
    outputFile.delete();
  }
} // end finalize
```

3 Look-Ahead Object Input Streams

Pierre Ernst had the original idea for look-ahead Java deserialization in his 2013 paper [Ernst 2013]. Look-ahead deserialization is a deserialization validation technique which allows the content of a serialized stream to be type-checked (and otherwise validated) prior to actual deserialization. Look-ahead deserialization is possible because the serialized binary data contains both metadata and the data itself, as specified and required by the *Java Object Serialization Specification* [JOSS 2010]. This metadata includes information about the structure of the data, such as class name, number of members, and type of members.

Ernst uses a simple bicycle class to illustrate this point:

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

public class Bicycle implements java.io.Serializable {
  private static final long serialVersionUID = 5754104541168320730L;

  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;
  }

  // Setters and getters omitted.
}
```

Serializing an instance of this class generates the following serialized data stream:

```
000000: AC ED 00 05 73 72 00 2C 63 6F 6D 2E 69 62 6D 2E |····sr·,com.ibm.|
000016: 62 61 2E 73 63 67 2E 4C 6F 6F 6B 41 68 65 61 64 |ba.scg.LookAhead|
000032: 44 65 73 65 72 69 61 6C 69 7A 65 72 2E 42 69 63 |Deserializer.Bic|
000048: 79 63 6C 65 4F DA AF 97 F8 CC C0 DA 02 00 03 49 |ycleO··········I|
000064: 00 02 69 64 49 00 09 6E 62 72 57 68 65 65 6C 73 |··idI··nbrWheels|
000080: 4C 00 04 6E 61 6D 65 74 00 12 4C 6A 61 76 61 2F |L··name···Ljava/|
000096: 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 78 70 00 00 |lang/String;····|
000112: 00 00 00 00 01 74 00 08 55 6E 69 63 79 63 6C |·····t··Unicycl|
000128: 65                                              |e|
```

The serialized stream begins with a magic number and version written by the `writeStreamHeader` when called by `ObjectOutputStream` during serialization:

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

The name and `serialVersionUID` of the class from the serialization's descriptor for classes `ObjectStreamClass` is recorded in the data stream for all classes present in the stream:

```
className
  length (2 bytes) 0x2C = 44
  text (59 bytes) com.ibm.ba.scg.LookAheadDeserializer.Bicycle
```

```
serialVersionUID (8 bytes) 0x4FDAAF97F8CCC0DA = 5754104541168320730
```

Because this information is used to deserialize the objects, it must be present and accurate for deserialization to succeed. Consequently, it will always identify which classes will be deserialized, even if the data stream has been tampered with.

The serialized data stream also includes instance data for the object, named `Unicycle` in this example.

The existence of metadata in the serialized data stream allows an object stream to preview the contents of the stream before invoking potentially dangerous callback methods that might contain RCE, DoS, or other gadgets. The metadata can then be validated as expected and shown to be non-malicious before deserialization begins in earnest. A number of look-ahead object input stream (LAOIS) implementations have been developed that vary in what information they can validate and how the developer specifies what constitutes a valid or invalid object. Examples of LAOIS implementations include:

- SerialKiller
- Apache Commons Class IO `ValidatingObjectInputStream`
- Contrast Security `contrast-r00`

SerialKiller

SerialKiller[7] is a LAOIS developed by Luca Carettoni to detect malicious payloads or whitelist valid application classes. `SerialKiller` is used in place of the standard `java.io.ObjectInputStream`. This is accomplished with a one-line change:

```
ObjectInputStream ois = new SerialKiller(is, "/etc/serialkiller.conf");
String msg = (String) ois.readObject();
```

The second argument is the location of SerialKiller's configuration file.

SerialKiller supports blacklisting, whitelisting, profiling, and logging. Both list types are specified as a Java regular expression. The default configuration file already includes several known payloads so that an application is protected by default against known attacks. A profiling mode enumerates classes deserialized by the application. Deserialization is not blocked in this mode. To protect your application, ensure this option is set to `false` for production (default value). Profiling simplifies the identification of objects necessary for proper deserialization.

SerialKiller also provides basic logging capability. Out of the box, this capability is configured for Linux platforms. Modify the configuration file to prevent a `NoSuchFileException` exception on Windows platforms:

```
<logging>
  <enabled>true</enabled>
  <logfile>/tmp/serialkiller.log</logfile>
</logging>
```

SerialKiller requires source code changes and is consequently not a useful solution for administrators lacking access to source code.

[7] https://github.com/ikkisoft/SerialKiller

Apache Commons Class IO `ValidatingObjectInputStream`

The Apache Commons Class IO Serialization package[8] provides a framework for controlling the deserialization of objects and includes the `ValidatingObjectInputStream` class. Similar to SerialKiller, `ValidatingObjectInputStream` extends `ObjectInputStream`, adding a variety of methods for specifying which classes can be deserialized. These methods either accept the specified classes for deserialization (unless they are rejected), or reject the specified classes for deserialization, even if they are otherwise accepted. The following `deserialize` method, for example, will only deserialize objects of `Bicycle.class`.

```
private static Object deserialize(byte[] buffer) throws IOException,
    ClassNotFoundException, ConfigurationException {
  Object obj;
  try (ByteArrayInputStream bais = new ByteArrayInputStream(buffer);
    // Use ValidatingObjectInputStream instead of InputStream
    ValidatingObjectInputStream ois = new ValidatingObjectInputStream(bais);) {
    ois.accept(Bicycle.class);
    obj = ois.readObject();
  }
  return obj;
}
```

As was the case with SerialKiller, `ValidatingObjectInputStream` supports whitelisting and blacklisting methods but has no real defense against DoS attacks. The Apache Commons Class IO LAOIS solution does not offer profiling and logging out of the box. Apache Commons Class IO requires source code changes and is consequently not a useful solution for administrators lacking access to source code.

Contrast Security `contrast-r00`

The Contrast Security `contrast-r00` solution[9] is somewhat different from the two LAOIS already described.

A lightweight Java agent, `contrast-r00` can be used to hot patch an application against Java deserialization vulnerabilities by modifying the behavior of `ObjectInputStream` via rewriting it. `contrast-r00` can also be used in an analogous manner to SerialKiller and `ValidatingObjectInputStream` by modifying the source code. For example, the following code uses the `SafeObjectInputStream` to perform whitelisting.

```
SafeObjectInputStream in = new SafeObjectInputStream(inputStream, true);
in.addToWhitelist(SafeClass.getName());
in.addToWhitelist("com.my.SafeDeserializable");
// Everything else is the same
in.readObject();
```

`contrast-r00` also supports blacklists but has no defense against DoS attacks.

[8] https://commons.apache.org/proper/commons-io/
[9] https://github.com/Contrast-Security-OSS/contrast-rO0

JEP 290: Filter Incoming Serialization Data is a new Java core library feature created through the JDK Enhancement Proposals (JEP) process and released with Java 9. JEP 290 allows incoming streams of object-serialization data to be filtered to improve both security and robustness. The filter mechanism allows object-serialization clients to validate their inputs and exported RMI objects to validate invocation arguments.

The core mechanism is a filter interface implemented by serialization clients and set on an ObjectInputStream. The filter interface methods are called during the deserialization process to validate the classes being deserialized, the sizes of arrays being created, the stream length, graph depth, and number of references as the stream is being decoded. A filter determines whether the arguments are ALLOWED or REJECTED and should return the appropriate status. If the filter cannot determine the status it should return UNDECIDED. Filters are designed for the specific use case and expected types. A filter designed for a particular use may be passed a class that is outside of the scope of the filter. If, for example, the purpose of the filter is to black-list classes then it can reject a candidate class that matches and report UNDECIDED for others.

Dissimilar to the other look-ahead object serialization solutions discussed in this whitepaper, JEP 290 can be used to defeat DoS attacks by limiting the sizes of arrays being created, the stream length, stream depth, and number of references to those values required for normal operations.

Java 9 supports custom filters, process-wide filters, and built-in filters [Giannakidis 2016]. Configurable process-wide filters are also supported in JDK 8, Update 121 (JDK 8u121), JDK 7, Update 131 (JDK 7u131), and JDK 6, Update 141 (JDK 6u141). Additionally, JEP 290 provides a logging facility.

Custom Filters

Custom filters are created by implementing the ObjectInputFilter interface and overriding the checkInput method. The following BikeFilter class provides a custom filter for deserializing a single Bicycle object:

```
import java.util.List;
import java.util.Optional;
import java.util.function.Function;
import java.io.ObjectInputFilter;

class BikeFilter implements ObjectInputFilter {
  private long maxStreamBytes = 78; // Maximum allowed bytes in the stream.
  private long maxDepth = 1; // Maximum depth of the graph allowed.
  private long maxReferences = 1; // Maximum number of references in a graph.

  @Override
  public Status checkInput(FilterInfo filterInfo) {
    if (filterInfo.references() < 0 || filterInfo.depth() < 0 || filterInfo.
        streamBytes() < 0
        || filterInfo.references() > maxReferences || filterInfo.depth() > maxDepth
        || filterInfo.streamBytes() > maxStreamBytes) {
      return Status.REJECTED;
    }
    Class<?> clazz = filterInfo.serialClass();
    if (clazz != null) {
      if (Bicycle.class == filterInfo.serialClass()) {
        return Status.ALLOWED;
      }
      else {
```

```
        return Status.REJECTED;
      }
    }
    return Status.UNDECIDED;
  } // end checkInput
} // end class BikeFilter
```

In many calls to the filter, the class is null. For example, only the reference count changes when a back reference is encountered.

This is a relatively simple example that does not deal with multiple filter patterns or allow arrays. Classes are rejected if their metrics are negative or exceed the established limits. Classes with valid metrics are deserialized only if they are the `Bicycle.class`; otherwise, they are rejected.

The checks for negative values in this filter is defensive coding for the pattern-based filters to avoid making assumptions about the range of values and to protect against wraparound. Initially, the only caller of filters is `ObjectInputStream`, and it does not produce negative values without wraparound.

Java 9 adds additional methods to `ObjectInputStream` to set and get the current filter:

```
public class ObjectInputStream ... {
    public final void setObjectInputFilter(ObjectInputFilter filter);
    public final ObjectInputFilter getObjectInputFilter(ObjectInputFilter filter);
}
```

If no filter is set for an `ObjectInputStream` then the global filter is used, if any.

Process-wide Filters

Process-wide filters provide an effective emergency measure to mitigate against Java deserialization vulnerabilities. A process-wide filter is configured via a system property or a configuration file. If supplied, the system property (`jdk.serialFilter`) supersedes the security property value (`jdk.serialFilter` in `conf/security/java.properties`). For example, the system property can be set programmatically, as shown by the following code:

```
Properties props = System.getProperties();
props.setProperty("jdk.serialFilter", "Bicycle;!*;maxdepth=1;maxrefs=1;maxbytes=78;
    maxarray=10");
```

Patterns are separated by ";" (semicolon). Whitespace is significant and is considered part of the pattern. If a pattern includes an equals assignment ("="), it sets a limit. If a limit appears more than once, the last value is used.

- `maxdepth=value` - maximum graph depth
- `maxrefs=value` - maximum number of internal references
- `maxbytes=value` - maximum number of bytes in the input stream
- `maxarray=value` - maximum array length allowed

These limits should be set to the minimum values required for the successful execution of the program to mitigate against DoS attacks.

Other patterns match or reject class or package names as returned from `Class.getName()` and, if an optional module name is present, `class.getModule().getName()`. The element type is used in the pattern for arrays, and not the array type. If the pattern:

- Starts with "`!`", the class is rejected if the remaining pattern is matched; otherwise, the class is allowed if the pattern matches.
- Ends with "`.**`", it matches any class in the package and all subpackages.
- Ends with "`.*`", it matches any class in the package.
- Ends with "`*`", it matches any class with the pattern as a prefix.
- Is equal to the class name, it matches.

Otherwise, the pattern is not matched. The resulting filter performs the limit checks and then tries to match the class, if any. If any of the limits are exceeded, the filter returns `Status.REJECTED`. If the class is an array type, the class to be matched is the element type. Arrays of any number of dimensions are treated the same as the element type. For example, a pattern of `!example.Foo`, rejects creation of any instance or array of `example.Foo`. The first pattern that matches, working from left to right, determines the `Status.ALLOWED` or `Status.REJECTED` result. If the limits are not exceeded and no pattern matches the class, the result is `Status.UNDECIDED`. It is informative to examine the source code for `ObjectInputFilter` [10]

For example, the following code is used to construct a new filter from a pattern. In this case, the pattern is a class name:

```
// Pattern is a class name
if (negate) {
  // A Function that fails if the class equals the pattern, otherwise don't care
  patternFilter = c -> c.getName().equals(name) ? Status.REJECTED : Status.UNDECIDED;
} else {
  // A Function that succeeds if the class equals the pattern, otherwise don't care
  patternFilter = c -> c.getName().equals(name) ? Status.ALLOWED : Status.UNDECIDED;
}
```

In the `negate` case, a matching pattern means that the class is rejected. Otherwise, the filter returns `Status.UNDECIDED`. This means that unless another filter explicitly allows or rejects the class, it will be *allowed*. Consequently, the filter pattern from our earlier example:

`"Bicycle;!*;maxdepth=1;maxrefs=1;maxbytes=78;maxarray=10"`

deserializes objects where the class name equals `ser05j.Bicycle` and rejects all other classes. Reversing the order of the first two filters:

`"!*;Bicycle;maxdepth=1;maxrefs=1;maxbytes=78;maxarray=10"`

results in the `ObjectInputFilter` rejecting *all* classes. The remaining filters in this pattern are to mitigate against DoS attacks by setting the various limits to the maximum required to deserialize a single `Bicycle` object.

The Java documentation for `ObjectInputFilter` [11] suggests that a custom filter should check if a process-

[10] https://github.com/netroby/jdk9-dev/blob/master/jdk/src/java.base/share/classes/java/io/ObjectInputFilter.java
[11] http://download.java.net/java/jdk9/docs/api/java/io/ObjectInputFilter.html

wide filter is configured and, if so, defer to it as in the following example:

```
ObjectInputFilter.Status checkInput(FilterInfo info) {
    ObjectInputFilter serialFilter = ObjectInputFilter.Config.getSerialFilter();
    if (serialFilter != null) {
        ObjectInputFilter.Status status = serialFilter.checkInput(info);
        if (status != ObjectInputFilter.Status.UNDECIDED) {
            // The process-wide filter overrides this filter
            return status;
        }
    }
    if (info.serialClass() != null &&
        Remote.class.isAssignableFrom(info.serialClass())) {
        return Status.REJECTED;      // Do not allow Remote objects
    }
    return Status.UNDECIDED;
}
```

The advantage of deferring to a process-wide filter is that an administrator can modify the filter patterns post deployment. However, a custom filter can be specific to a particular `ObjectInputStream.readObject` call and can precisely allow only what is needed for a particular `ObjectInputStream.readObject` invocation.

Built-in Filters

RMI Registry and Distributed Garbage Collection (DGC) use JEP 290 Serialization Filtering to improve service security and robustness. RMI Registry and DGC implement built-in whitelist filters for the typical classes expected to be used with each service. These built-in filters implement pre-configured whitelists of classes and limits that are typical for the RMI Registry and DGC use cases:

```
java.rmi.server.ObjID;
java.rmi.server.UID;
java.rmi.dgc.VMID;
java.rmi.dgc.Lease;
maxdepth=5;
maxarray=10000;
```

Developers can configure additional filter patterns to the built-in filters using the `sun.rmi.registry.registryFilter` and `sun.rmi.transport.dgcFilter` system or security properties.

5 LAOIS Benefits and Limitations

Java deserialization vulnerabilities represent an extremely serious class of vulnerability that can allow exploits from DoS to RCE for deserialization code that might otherwise be considered defect-free. The primary mitigation is to not deserialize untrusted data. This idea is echoed by *The CERT Oracle Secure Coding Standard for Java* [Long 2012], which contains Rule "SER12-J. Prevent deserialization of untrusted data." In cases where deserializing untrusted data cannot be avoided, a secondary mitigation is to protect the application via hardening and authentication, including:

- Using a look-ahead object input stream (LAOIS) to only enable the deserialization of necessary classes
- Using a security manager and security policy
- Network listeners used for/with deserialization should listen only on loopback adapters or ports that are protected by local/network firewall rules
- Data channels or streams used for deserialization should be authenticated with credentials and/or cryptography before object deserialization starts

This whitepaper is solely focused on the secondary mitigation strategy of LAOIS, although other secondary mitigations should be applied in concert.

Blacklisting

Blacklisting has detractors and supporters. Blacklisting can be beneficial to defeat a specific threat in a short time-frame with the least likelihood of collateral damage. A limitation of blacklisting is that new gadgets are continually being discovered, and of course, blacklisting cannot protect against unknown gadgets. Even in the case where no new gadgets are found, bypass gadgets can be used to defeat blacklisting [Muñoz 2016]. Classes similar to the following example have been discovered in the JRE, third-party libraries, and application servers:

```
public class NestedProblems implements Serializable {
  byte[] bytes … ;…

  private void readObject(ObjectInputStream in) throws IOException,
      ClassNotFoundException {
    ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(bytes));
    ois.readObject();
  }
}
```

This code allows an attacker to provide their own unprotected `ObjectInputStream` and then exploit it using one or more of the previously described deserialization exploits described. The gadget bypass does not work if LAOIS is applied via instrumentation, because the nested deserialization will also be an instance of a LAOIS rather than just a regular OIS. Consequently, `contrast-r0` and similar solutions successfully protect against bypass gadgets. Similarly, the JEP 290 process-wide filter will also protect against bypass gadgets as these are applied against all OIS instances.

Whitelisting and Profiling

The application is secure if the intersection of this set of objects and the sets of gadgets is the empty set (∅). Realistically, whitelisting alone cannot prevent DoS attacks, such as deserializing an `ArrayList` with a size of `Integer.MAX_VALUE`. The effective use of whitelisting requires a whitelist that is as restrictive as possible. This whitelist can best be determined by some form of profiling.

There is some disagreement among experts as to the use of profiling in Java deserialization (in private emails to the author). The original intent of Java serialization was that it was not necessary to know which classes needed to be serialized. This allowed these classes to evolve over time or just to change slightly with a minor update. Profiling may also fail to account for slightly different data than expected. For a given class, it may not be possible to know which classes need to be available. For example, many classes will store objects of subclassable types.

In the simple example in this paper, print statements were used to determine the necessary values for the filter. This works if you are only deserializing a single instance of a single class, but real applications are seldom this simple. Consequently, a profiling tool like the one included in SerialKiller is useful for creating a restrictive whitelist, even if a different LAOIS implementation is used at runtime. If the restrictive whitelist does not allow any gadgets, and the metrics are sufficiently limited, deserialization vulnerabilities can be prevented. Secure coding of serializable classes and identifying vulnerabilities in existing classes will be discussed in a future whitepaper.

The application of a restrictive whitelist can easily conflict with the original intent of Java serialization and prevent class evolution, but may be necessary in cases where untrusted data is deserialized.

Denial of Service

Denial-of-service attacks are among the hardest to defend against, and doing so may be impossible. LAOIS solutions such as SerialKiller, Apache Commons Class IO `ValidatingObjectInputStream`, and Contrast Security `contrast-rO0` do not mitigate against denial-of-service attacks. JEP 290 attempts to defend against these attacks by allowing the developer or administrator to establish various limits. The filter pattern `"jdk.serialFilter", "ser05j.Bicycle;!*;maxdepth=1;maxrefs=1;maxbytes=78;maxarray=10"`, for example, can effectively prevent the SerialDOS exploit:

```
May 26, 2017 4:18:09 PM java.io.ObjectInputStream filterCheck
INFO: ObjectInputFilter REJECTED: class java.util.HashSet, array length: -1, nRefs:
    1, depth: 1, bytes: 36, ex: n/a
java.io.InvalidClassException: filter status: REJECTED
```

But cannot prevent other attacks including: generic heap DoS inside `ObjectInputStream`; heap DoS using nested `Object[]`, `ArrayList`, and `HashMap`; collision attacks on `Hashtable`; and collision attacks on `HashMap` (Oracle Java 1.7)[12] even with the following filter in place `"maxdepth=1;maxrefs=1;maxbytes=100;maxarray=1"`.

Deserialization in Libraries

Developers do not always have access to `ObjectInputStream` implementations in libraries, and administrators do not have direct access to any of them. Process-wide filters provide a way to constrain the behavior of these implementations.

[12] https://github.com/topolik/ois-dos

6 Conclusion

Look-ahead object input streams can be used as a mitigation for Java deserialization issues in cases where deserializing untrusted data cannot be avoided. There are several different LAOIS implementations; most are lacking in their defense against DoS attacks. The most promising solution is JEP 290 Serialization Filtering (shipping with Java 9). Configurable process-wide filters are also available in recent updates to Java 6, Java 7, and Java 8. However, even JEP 290 (as currently implemented) is deficient in its defense against DoS attacks.

LAOIS does not address the development of serializable classes. If these classes are not coded securely and are whitelisted by the LAOIS, the overall system will still be vulnerable. A future whitepaper will examine the more complicated issue of securely implementing serializable classes.

7 Acknowledgments

Thanks to Jeremy Brandt-Young, David Goldsmith, Andy Grant, Heather Overcash, and Audrey Saunders for supporting this effort. Thanks to the technical reviewers Daniele Costa, Jake Heath, Fred Long (Aberystwyth University), Alvaro Muñoz (HPE), Pierre Ernst, Thomas Hawtin, Apostolos Giannakidis (Waratek), Arshan Dabirsiaghi (Contrast Security), Will Klieber (SEI/CERT), and Roger Riggs (Oracle).

8 Author Bio

Robert C. Seacord, a renowned computer scientist and author, known as the "father of secure coding." Robert is a Principal Security Consultant with NCC Group where he works with software developers and software development organizations to eliminate vulnerabilities resulting from coding errors before they are deployed. Previously, Robert led the secure coding initiative in the CERT Division of Carnegie Mellon University's Software Engineering Institute (SEI). Robert is also an adjunct professor in the School of Computer Science and the Information Networking Institute at Carnegie Mellon University. Robert is the author of six books, including *The CERT C Coding Standard, Second Edition* (Addison-Wesley, 2014), *Secure Coding in C and C++, Second Edition* (Addison-Wesley, 2013), *The CERT Oracle Secure Coding Standard for Java* (Addison-Wesley, 2012), and *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* (Addison-Wesley, 2014). Robert is on the Advisory Board for the Linux Foundation and an expert on the ISO/IEC JTC1/SC22/WG14 international standardization working group for the C programming language.

9 References

[API 2014] Java Platform, Standard Edition 8 API Specification, Oracle (2014).

[Bloch 2008] Bloch, Joshua. *Effective Java*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley (2008).

[Ernst 2013] Pierre Ernst. Look-ahead Java deserialization, January 15, 2013.

[Frohoff 2016] Chris Frohoff. Security Advisory – Java SE, January 26, 2016.

[Giannakidis 2016] Apostolos Giannakidis. A First Look Into Java's New Serialization Filtering, January 20, 2017.

[JSO 2016] Java Security Overview, Oracle (2016).

[JOSS 2010] Java Object Serialization Specification version 6.0, Oracle (2016).

[JLS 2015] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. Java Language Specification: Java SE 8 Edition. Oracle America (2016).

[JVMS 2015] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley. The Java Virtual Machine Specification: Java SE 8 Edition. Oracle America (2015).

[Lawrence 2015] Gabriel Lawrence and Chris Frohoff Marshalling Pickles how deserializing objects can ruin your day. Qualcomm (2015).

[Long 2012] Long, Fred, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*, SEI Series in Software Engineering. Boston: Addison-Wesley (2012).

[Long 2013] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. 2013. *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* (1st ed.). Addison-Wesley Professional.

[Muñoz-Schneider 2016] Alvaro Muñoz and Christian Schneider. Serial Killer: Silently Pwning Your Java Endpoints. RSA Conference (2016).

[Muñoz 2016] Alvaro Muñoz. The perils of Java deserialization. HPE Security Research Technical Report (2016).

[SCG 2015] Secure Coding Guidelines for Java SE, version 5.1 Oracle (2015).

[Shacham 2007] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and communications security (CCS '07). ACM, New York, NY, USA, 552-561.

[Tutorials 2016] The Java Tutorials. Oracle (2016).