Java安全漫谈 - 18.原生反序列化利用链 JDK7u21

> 这是代码审计知识星球中Java安全的第十八篇文章。

攻克了前面介绍过的第三方的反序列化利用链，自然我们会想到，没有合适的第三方库存在时，Java反序列化是否还能利用。这是一件幸运又不幸的事情，幸运的是，的确存在不依赖第三方库的Java反序列化利用链，不幸的是，新的Java版本没有这样的问题。

这条利用链就是JDK7u21，顾名思义，它适用于Java 7u21及以前的版本。所以，在阅读并复现本文章前，需要先安装Java 7u21，并在项目中将版本设置为7u21。

JDK7u21的核心原理

相信学习了CommonsCollections的这些利用链后，大家心里对反序列化有自己的认识。如果问，什么是某条反序列化利用链的核心点，有的同学可能会说是readObject或TemplatesImpl。不过我的理解是，核心在于触发**"动态方法执行"**的地方，而不是TemplatesImpl或某个类的readObject方法。

举几个例子：

- CommonsCollections系列反序列化的核心点是那一堆`Transformer`，特别是其中的`InvokerTransformer`、`InstantiateTransformer`
- CommonsBeanutils反序列化的核心点是`PropertyUtils#getProperty`，因为这个方法会触发任意对象的getter

而JDK7u21的核心点就是`sun.reflect.annotation.AnnotationInvocationHandler`，记性好的同学应该还对此有印象。我们曾经在《Java安全漫谈》的第10和11章中介绍过（不记得的同学回看一下），但当时只用到了这个类会触发`Map#put`、`Map#get`的特点。

其实，我们看到AnnotationInvocationHandler类中的equalsImpl方法：

```
1  private Boolean equalsImpl(Object o) {
2      if (o == this)
3          return true;
4
5      if (!type.isInstance(o))
6          return false;
7      for (Method memberMethod : getMemberMethods()) {
8          String member = memberMethod.getName();
9          Object ourValue = memberValues.get(member);
10         Object hisValue = null;
11         AnnotationInvocationHandler hisHandler = asOneOfUs(o);
12         if (hisHandler != null) {
13             hisValue = hisHandler.memberValues.get(member);
14         } else {
15             try {
16                 hisValue = memberMethod.invoke(o);
17             } catch (InvocationTargetException e) {
18                 return false;
19             } catch (IllegalAccessException e) {
20                 throw new AssertionError(e);
21             }
22         }
```

```
        if (!memberValueEquals(ourValue, hisValue))
            return false;
    }
    return true;
}

private transient volatile Method[] memberMethods = null;
private Method[] getMemberMethods() {
    if (memberMethods == null) {
        memberMethods = AccessController.doPrivileged(
            new PrivilegedAction<Method[]>() {
                public Method[] run() {
                    final Method[] mm = type.getDeclaredMethods();
                    AccessibleObject.setAccessible(mm, true);
                    return mm;
                }
            });
    }
    return memberMethods;
}
```

这个方法中有个很明显的反射调用`memberMethod.invoke(o)`，而`memberMethod`来自于`this.type.getDeclaredMethods()`。

也就是说，`equalsImpl`这个方法是将`this.type`类中的所有方法遍历并执行了。那么，假设`this.type`是Templates类，则势必会调用到其中的`newTransformer()`或`getOutputProperties()`方法，进而触发任意代码执行。

这就是JDK7u21的核心原理。

如何调用equalsImpl

那么，现在的任务就是通过反序列化调用`equalsImpl`，`equalsImpl`是一个私有方法，在`AnnotationInvocationHandler#invoke`中被调用。

`AnnotationInvocationHandler#invoke`这个名字好像又很熟悉，也是《Java安全漫谈》提到过的一个知识点，在第11篇中：

> 作为一门静态语言，如果想劫持一个对象内部的方法调用，实现类似PHP的魔术方法`__call`，我们需要用到`java.reflect.Proxy`：
>
> ```
> Map proxyMap = (Map) Proxy.newProxyInstance(Map.class.getClassLoader(),
> new Class[] {Map.class}, handler);
> ```
>
> `Proxy.newProxyInstance`的第一个参数是ClassLoader，我们用默认的即可；第二个参数是我们需要代理的对象集合；第三个参数是一个实现了InvocationHandler接口的对象，里面包含了具体代理的逻辑。
>
> ...
>
> 我们回看`sun.reflect.annotation.AnnotationInvocationHandler`，会发现实际上这个类实际就是一个InvocationHandler，我们如果将这个对象用Proxy进行代理，那么在readObject的时候，只要调用任意方法，就会进入到`AnnotationInvocationHandler#invoke`方法中，进而触发我们的`LazyMap#get`。
>
> 十分美妙。

InvocationHandler是一个接口，他只有一个方法就是invoke：

```
public interface InvocationHandler {
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

在使用`java.reflect.Proxy`动态绑定一个接口时，如果调用该接口中任意一个方法，会执行到`InvocationHandler#invoke`。执行invoke时，被传入的第一个参数是这个proxy对象，第二个参数是被执行的方法名，第三个参数是执行时的参数列表。

而`AnnotationInvocationHandler`就是一个`InvocationHandler`接口的实现，我们看看它的invoke方法：

```
public Object invoke(Object proxy, Method method, Object[] args) {
    String member = method.getName();
    Class<?>[] paramTypes = method.getParameterTypes();

    // Handle Object and Annotation methods
    if (member.equals("equals") && paramTypes.length == 1 &&
        paramTypes[0] == Object.class)
        return equalsImpl(args[0]);
    assert paramTypes.length == 0;
    if (member.equals("toString"))
        return toStringImpl();
    if (member.equals("hashCode"))
        return hashCodeImpl();
    if (member.equals("annotationType"))
        return type;
    // ...
```

可见，当方法名等于“equals”，且仅有一个Object类型参数时，会调用到`equalImpl`方法。

所以，现在的问题变成，我们需要找到一个方法，在反序列化时对proxy调用equals方法。

找到equals方法调用链

比较Java对象时，我们常用到两个方法：

- equals
- compareTo

任意Java对象都拥有`equals`方法，它通常用于比较两个对象是否是同一个引用；而compareTo实际上是`java.lang.Comparable`接口的方法，我在前一篇介绍`java.util.PriorityQueue`时也介绍过，通常被实现用于比较两个对象的值是否相等。

所以，我第一时间想到使用`java.util.PriorityQueue`，但实际上其中用的是compareTo，而非equals。

另一个常见的会调用equals的场景就是集合set。set中储存的对象不允许重复，所以在添加对象的时候，势必会涉及到比较操作。

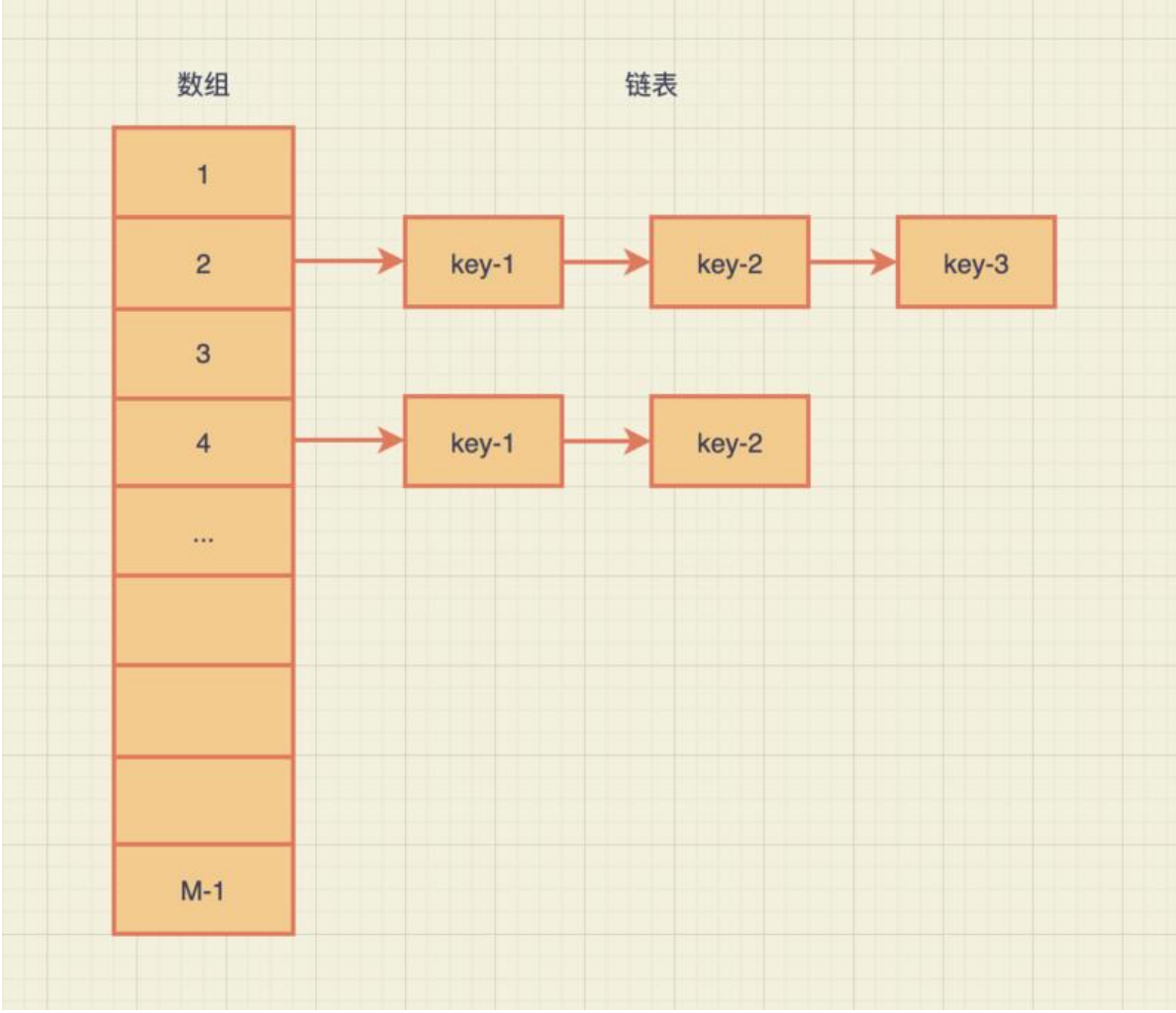
我们查看HashSet的readObject方法：

```
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Read in any hidden serialization magic
```

```
    s.defaultReadObject();

    // Read in HashMap capacity and load factor and create backing HashMap
    int capacity = s.readInt();
    float loadFactor = s.readFloat();
    map = (((HashSet)this) instanceof LinkedHashSet ?
           new LinkedHashMap<E,Object>(capacity, loadFactor) :
           new HashMap<E,Object>(capacity, loadFactor));

    // Read in size
    int size = s.readInt();

    // Read in all elements in the proper order.
    for (int i=0; i<size; i++) {
        E e = (E) s.readObject();
        map.put(e, PRESENT);
    }
}
```

可见，这里使用了一个HashMap，将对象保存在HashMap的key处来做去重。

HashMap，就是数据结构里的哈希表，相信上过数据结构课程的同学应该还记得，哈希表是由数组+链表实现的——哈希表底层保存在一个数组中，数组的索引由哈希表的`key.hashCode()`经过计算得到，数组的值是一个链表，所有哈希碰撞到相同索引的key-value，都会被链接到这个链表后面。



所以，为了触发比较操作，我们需要让比较与被比较的两个对象的哈希相同，这样才能被连接到同一条链表上，才会进行比较。

跟进下HashMap的put方法：

```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key);
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    modCount++;
    addEntry(hash, key, value, i);
    return null;
}
```

变量`i`就是这个所谓的"哈希"。两个不同的对象的`i`相等时，才会执行到`key.equals(k)`，触发前面说过的代码执行。

所以，我们接下来的目的就是为了让proxy对象的"哈希"，等于TemplateImpl对象的"哈希"。

巧妙的Magic Number

计算"哈希"的主要是下面这两行代码：

```
int hash = hash(key);
int i = indexFor(hash, table.length);
```

将其中的关键逻辑提权出来，可以得到下面这个函数：

```
public static int hash(Object key) {
    int h = 0;
    h ^= key.hashCode();

    h ^= (h >>> 20) ^ (h >>> 12);
    h = h ^ (h >>> 7) ^ (h >>> 4);
    return h & 15;
}
```

除了`key.hashCode()`外再没有其他变量，所以proxy对象与TemplateImpl对象的"哈希"是否相等，仅取决于这两个对象的`hashCode()`是否相等。TemplateImpl的`hashCode()`是一个Native方法，每次运行都会发生变化，我们理论上是无法预测的，所以想让proxy的`hashCode()`与之相等，只能寄希望于`proxy.hashCode()`。

`proxy.hashCode()`仍然会调用到`AnnotationInvocationHandler#invoke`，进而调用到`AnnotationInvocationHandler#hashCodeImpl`，我们看看这个方法：

```
private int hashCodeImpl() {
    int result = 0;
    for (Map.Entry<String, Object> e : memberValues.entrySet()) {
        result += (127 * e.getKey().hashCode()) ^
            memberValueHashCode(e.getValue());
    }
    return result;
}
```

遍历`memberValues`这个Map中的每个key和value，计算每个`(127 * key.hashCode()) ^ value.hashCode()`并求和。

JDK7u21中使用了一个非常巧妙的方法：

- 当`memberValues`中只有一个key和一个value时，该哈希简化成`(127 * key.hashCode()) ^ value.hashCode()`
- 当`key.hashCode()`等于0时，任何数异或0的结果仍是他本身，所以该哈希简化成`value.hashCode()`。
- 当`value`就是TemplateImpl对象时，这两个哈希就变成完全相等

所以，我们找到一个hashCode是0的对象作为`memberValues`的key，将恶意TemplateImpl对象作为value，这个proxy计算的hashCode就与TemplateImpl对象本身的hashCode相等了。

找一个hashCode是0的对象，我们可以写一个简单的爆破程序来实现：

```
public static void bruteHashCode()
{
    for (long i = 0; i < 9999999999L; i++) {
        if (Long.toHexString(i).hashCode() == 0) {
            System.out.println(Long.toHexString(i));
        }
    }
}
```

运行了挺长时间，跑出来第一个是`f5a5a608`，这个也是ysoserial中用到的字符串。

利用链梳理

所以，整个利用的过程就清晰了，按照如下步骤来构造：

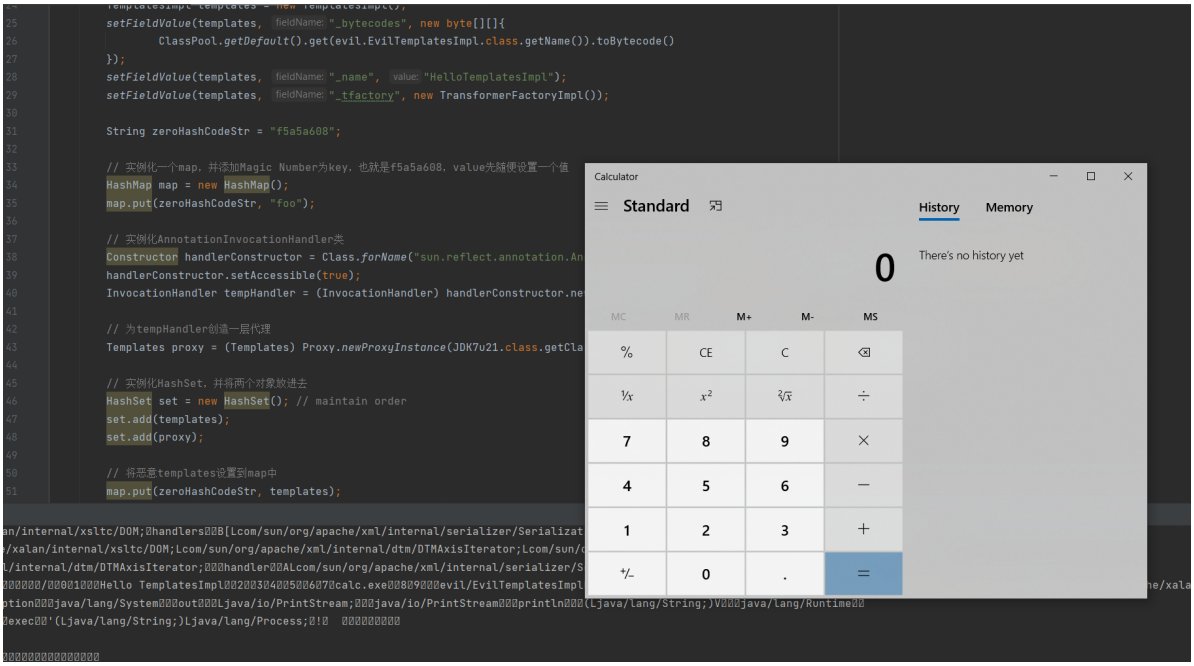
- 首先生成恶意`TemplateImpl`对象
- 实例化`AnnotationInvocationHandler`对象
    - 它的type属性是一个TemplateImpl类
    - 它的memberValues属性是一个Map，Map只有一个key和value，key是字符串`f5a5a608`，value是前面生成的恶意TemplateImpl对象
- 对这个`AnnotationInvocationHandler`对象做一层代理，生成proxy对象
- 实例化一个HashSet，这个HashSet有两个元素，分别是：
    - TemplateImpl对象
    - proxy对象
- 将HashSet对象进行序列化

这样，反序列化触发代码执行的流程如下：

- 触发HashSet的readObject方法，其中使用HashMap的key做去重

- 去重时计算HashSet中的两个元素的`hashCode()`，因为我们的静心构造二者相等，进而触发`equals()`方法
- 调用`AnnotationInvocationHandler#equalsImpl`方法
- `equalsImpl`中遍历`this.type`的每个方法并调用
- 因为`this.type`是TemplatesImpl类，所以触发了`newTransform()`或`getOutputProperties()`方法
- 任意代码执行

按照这个步骤，我简化了一个比较好理解的JDK7u21代码：https://github.com/phith0n/JavaThings/blob/master/general/src/main/java/com/govuln/deserialization/JDK7u21.java



修复

这个利用链俗名是JDK7u21，可以认为它可以在7u21及以前的版本中使用，这就牵扯出两个问题：

- 这个利用链是否影响JDK6和JDK8，具体影响哪些小版本
- JDK7u21以上的版本如何修复这个问题

第一个问题，Java的版本是多个分支同时开发的，并不意味着JDK7的所有东西都一定比JDK6新，所以，当看到这个利用链适配7u21的时候，我们不能先入为主地认为JDK6一定都受影响。

Oracle JDK6一共发布了30多个公开的版本，最后一个公开版本是6u45，在2013年发布。此后，Oracle公司就不再发布免费的更新了，但是付费用户仍然可以获得Java 6的更新，最新的Java 6版本是6u221。

其中，公开版本的最新版6u45仍然存在这条利用链，大概是6u51的时候修复了这个漏洞，但是这个结论不能肯定，因为免费用户下载不到这个版本。

JDK8在发布时，JDK7已经修复了这个问题，所以JDK8全版本都不受影响。

我们来看看官方在JDK7u25中是怎样修复这个问题的：https://github.com/openjdk/jdk7u/commit/b3dd6104b67d2a03b94a4a061f7a473bb0d2dc4e

```
9 jdk/src/share/classes/sun/reflect/annotation/AnnotationInvocationHandler.java
...  ...  @@ -1,5 +1,5 @@
1    1    /*
2        - * Copyright (c) 2003, 2011, Oracle and/or its affiliates. All rights reserved.
     2    + * Copyright (c) 2003, 2013, Oracle and/or its affiliates. All rights reserved.
3    3    * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
4    4    *
5    5    * This code is free software; you can redistribute it and/or modify it
     @@ -337,12 +337,15 @@ private void readObject(java.io.ObjectInputStream s)
337  337          try {
338  338              annotationType = AnnotationType.getInstance(type);
339  339          } catch(IllegalArgumentException e) {
340      -            // Class is no longer an annotation type; all bets are off
341      -            return;
     340  +            // Class is no longer an annotation type; time to punch out
     341  +            throw new java.io.InvalidObjectException("Non-annotation type in annotation serial stream");
342  342          }
343  343
344  344          Map<String, Class<?>> memberTypes = annotationType.memberTypes();
345  345
     346  +
     347  +        // If there are annotation members without values, that
     348  +        // situation is handled by the invoke method.
346  349          for (Map.Entry<String, Object> memberValue : memberValues.entrySet()) {
347  350              String name = memberValue.getKey();
348  351              Class<?> memberType = memberTypes.get(name);
```

在`sun.reflect.annotation.AnnotationInvocationHandler`类的readObject函数中，原本有一个对`this.type`的检查，在其不是AnnotationType的情况下，会抛出一个异常。但是，捕获到异常后没有做任何事情，只是将这个函数返回了，这样并不影响整个反序列化的执行过程。

新版中，将`return;`修改成`throw new java.io.InvalidObjectException("Non-annotation type in annotation serial stream");`，这样，反序列化时会出现一个异常，导致整个过程停止：

这个修复方式看起来击中要害，实际上仍然存在问题，这也导致后面的另一条原生利用链JDK8u20。我们下会再来分析。