安全漏洞分析

CVE-2021-2394：Oralce7 月补丁日二次序列化漏洞分析

360CERT
北京奇虎科技有限公司

报告信息

报告名称	CVE-2021-2394：Oralce7 月补丁日二次序列化漏洞分析		
报告类型	安全漏洞分析	报告编号	B6-2021-072301
报告版本	1	报告日期	2021-07-21
报告作者	360CERT	联系方式	g-cert-report@360.cn
提供方	北京鸿腾智能科技有限公司-360CERT		
接收方			

报告修订记录

报告版本	日期	修订	审核	描述
1	2021-07-21	360CERT	360CERT	撰写报告

目录

1 漏洞简述

2021 年 07 月 21 日，360CERT 监测发现 Oracle官方 发布了 2021年7月份 的安全更新，本次分析报告选取的是其中一个反序列化漏洞，CVE 编号为 CVE-2021-2194，漏洞等级：严重，漏洞评分：9.8 。

2 风险等级

评定方式	等级
威胁等级	严重
影响面	广泛
攻击者价值	高
利用难度	低
360CERT 评分	9.8

3 漏洞详情

diff 补丁后发现 WebLogicFilterConfig 的新增了两个黑名单 package：

```
"oracle.eclipselink.coherence.integrated.internal.querying",
↪ "oracle.eclipselink.coherence.integrated.internal.cache"
```

这次的漏洞依然是一个二次序列化漏洞，前半部分的序列化思路和 CVE-2020-14756 类似，后面代码执行的思路来自 CVE-2020-14841 。**(要注意的是这个漏洞不能过 'Weblogic ' 新增的白名单机制，但是可以走 iiop)**在 oracle.eclipselink.coherence.integrated.internal.querying 中发现了 FilterExtractor 这个类，他的 readExternal 方法如下：

```
public void readExternal(DataInput in) throws IOException {
    this.attributeAccessor = SerializationHelper.readAttributeAccessor(in);
}
```

跟到 readAttributeAccessor 方法里，当 id 为 1 的时候会返回一个 MethodAttributeAccessor 对象，这个对象是CVE-2020-14841 之后被加入了黑名单的，不过在这里返回的话就不会走反序列化的流程。

```
public static AttributeAccessor readAttributeAccessor(DataInput in) throws IOException {
    int id = ExternalizableHelper.readInt(in);
    if (id == 0) {
        InstanceVariableAttributeAccessorExtended accessor = new InstanceVariableAttributeAccessorExtended();
        accessor.setAttributeName((String)ExternalizableHelper.readObject(in));
        return accessor;
    } else if (id == 1) {
        MethodAttributeAccessor accessor = new MethodAttributeAccessor();
        accessor.setAttributeName((String)ExternalizableHelper.readObject(in));
        accessor.setGetMethodName((String)ExternalizableHelper.readObject(in));
        accessor.setSetMethodName((String)ExternalizableHelper.readObject(in));
        return accessor;
    } else {
        return null;
    }
}
```

也就是说我们依然可以利用MethodAttributeAccessor 对象，并且最终会赋值到 attributeAccessor 属性，这个对象可以调用任意方法（也不是任意，后面会讲）。继续看到 FilterExtractor 的 extract 方法。

```
public Object extract(Object obj) {
    if (obj instanceof Wrapper) {
        obj = ((Wrapper)obj).unwrap();
    }

    if (!this.attributeAccessor.isInitialized()) {
        this.attributeAccessor.initializeAttributes(obj.getClass());
    }

    try {
        return this.attributeAccessor.getAttributeValueFromObject(obj);
    } catch (Exception var3) {
        return new FilterExtractor.InvalidObject();
    }
}
```

主动去调用了 attributeAccessor#getAttributeValueFromObject 方法，我们看到 MethodAttributeAccessor#getAttributeValueFromObject 。该方法会通过反射执行 anObject 的 getMethod 方法，这两个变量都是攻击者可控的，于是就能够进行利用，不过这里的 parameters 为 null ，所以只能调用无参方法。

```
public Object getAttributeValueFromObject(Object anObject) throws DescriptorException {
    return this.getAttributeValueFromObject(anObject, (Object[])null);
}

protected Object getAttributeValueFromObject(Object anObject, Object[] parameters) throws DescriptorException {
    try {
        if (PrivilegedAccessHelper.shouldUsePrivilegedAccess()) {
            try {
                return AccessController.doPrivileged(new PrivilegedMethodInvoker(this.getGetMethod(), anObject, parameters));
            } catch (PrivilegedActionException var5) {
                Exception throwableException = var5.getException();
                if (throwableException instanceof IllegalAccessException) {
                    throw DescriptorException.illegalAccessWhileGettingValueThruMethodAccessor(this.getGetMethodName(), anObject.getClass().getName(), throwableException
                } else {
                    throw DescriptorException.targetInvocationWhileGettingValueThruMethodAccessor(this.getGetMethodName(), anObject.getClass().getName(), throwableExcept
                }
            }
        } else {
            return this.getMethod.invoke(anObject, parameters);
        }
```

不难想到利用 JdbcRowSetImpl 。接着，我们还需要找到一个地方来调用 FilterExtractor#extract 。这里利用的又是 CVE-2020-14756 里使用过的 com.tangosol.util.aggregator.TopNAggregator.PartialResult ，这里就不细说了，最终可以调用一个 Comparator 的 compare 方法。不难想到 ExtractorComparator 。这里只需要给 m_extractor 赋值为 FilterExtractor 。

```
public int compare(T o1, T o2) {
    Comparable a1 = o1 instanceof Entry ? (Comparable)((Entry)o1).extract(this.m_extractor) : (Comparable)this.m_extractor.extract(o1);
    Comparable a2 = o2 instanceof Entry ? (Comparable)((Entry)o2).extract(this.m_extractor) : (Comparable)this.m_extractor.extract(o2);
    if (a1 == null) {
        return a2 == null ? 0 : -1;
    } else {
        return a2 == null ? 1 : a1.compareTo(a2);
    }
}
```

而他的 readExternal 方法，刚好又回从输入流中读取 m_extractor 值。

```
public void readExternal(DataInput in) throws IOException {
    this.m_extractor = (ValueExtractor)ExternalizableHelper.readObject(in);
}
```

3.1 构造 POC 需要注意的点

在给MethodAttributeAccessor 对象进行赋值的时候，仅仅是赋值了 gettername 和 settername 。

```
public static AttributeAccessor readAttributeAccessor(DataInput in) throws IOException {
    int id = ExternalizableHelper.readInt(in);
    if (id == 0) {
        InstanceVariableAttributeAccessorExtended accessor = new InstanceVariableAttributeAccessorExtende
        accessor.setAttributeName((String)ExternalizableHelper.readObject(in));
        return accessor;
    } else if (id == 1) {
        MethodAttributeAccessor accessor = new MethodAttributeAccessor();
        accessor.setAttributeName((String)ExternalizableHelper.readObject(in));
        accessor.setGetMethodName((String)ExternalizableHelper.readObject(in));
        accessor.setSetMethodName((String)ExternalizableHelper.readObject(in));
        return accessor;
    } else {
        return null;
    }
}
```

于是，当执行到 FilterExtractor#extract 时，会先执行 isInitialized 。

```
public Object extract(Object obj) {
    if (obj instanceof Wrapper) {
        obj = ((Wrapper)obj).unwrap();
    }

    if (!this.attributeAccessor.isInitialized()) {
        this.attributeAccessor.initializeAttributes(obj.getClass());
    }

    try {
        return this.attributeAccessor.getAttributeValueFromObject(obj);
    } catch (Exception var3) {
        return new FilterExtractor.InvalidObject();
    }
}
```

这里会返回 false 。

```
public boolean isInitialized() {
    return (this.getMethod != null || this.isReadOnly()) && (this.setMethod != null || this.isWriteOnly());
}
```

于是调用 initializeAttributes ，这里会根据我们设置的 getMethodName 、setMethodName 通过反射获取具体的方法。

```
protected void initializeAttributes(Class theJavaClass, Class[] getParameterTypes) throws DescriptorException {
    if (this.getAttributeName() == null) {
        throw DescriptorException.attributeNameNotSpecified();
    } else {
        DescriptorException descriptorException;
        try {
            this.setGetMethod(Helper.getDeclaredMethod(theJavaClass, this.getGetMethodName(), getParameterTypes));
            if (!this.isWriteOnly()) {
                this.setSetMethod(Helper.getDeclaredMethod(theJavaClass, this.getSetMethodName(), this.getSetMethodParameterTypes()));
            }

        } catch (NoSuchMethodException var5) {
            descriptorException = DescriptorException.noSuchMethodWhileInitializingAttributesInMethodAccessor(this.getSetMethodName(), this.getGetMethodName(), theJ
            descriptorException.setInternalException(var5);
            throw descriptorException;
        } catch (SecurityException var6) {
            descriptorException = DescriptorException.securityWhileInitializingAttributesInMethodAccessor(this.getSetMethodName(), this.getGetMethodName(), theJavaC
            descriptorException.setInternalException(var6);
            throw descriptorException;
        }
    }
}
```

这里 isWriteOnly 默认为 false ， 不能像之前 CVE-2020-14841 一样通过反射进行修改改值，因为 MethodAttributeAccessor 在黑名单里。

```
public boolean isWriteOnly() {
    return this.isWriteOnly;
}
```

这里的逻辑是获取我们要调用的 get 方法，这里是一个无参方法，反射之后，获取 get 方法的返回值类型，然后传给 set 方法，所以这里 set 方法的参数是根据 get 方法来的（不一定必须是 getter/setter ）。

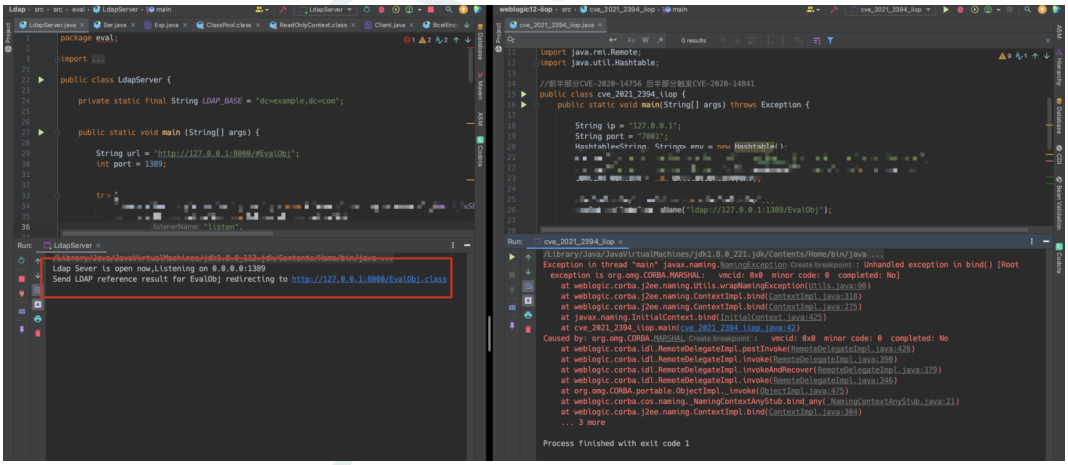
最终找到的JdbcRowSetImpl方法：

```
get -> connect()
set -> setConnection()
```

这样，刚好满足 connect 返回值类型是 set 方法的形参类型。

3.2 漏洞证明

iiop 利用：

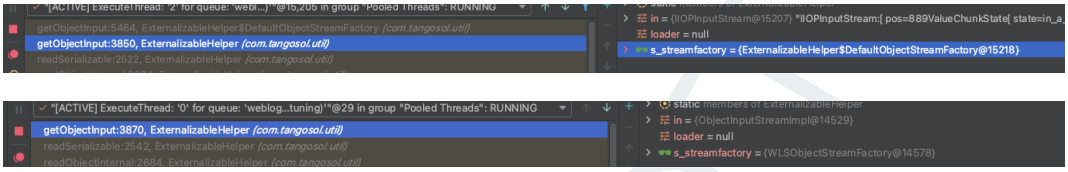


3.3 漏洞修复

第一个就是把利用中使用的两个 package 加入了黑名单里。

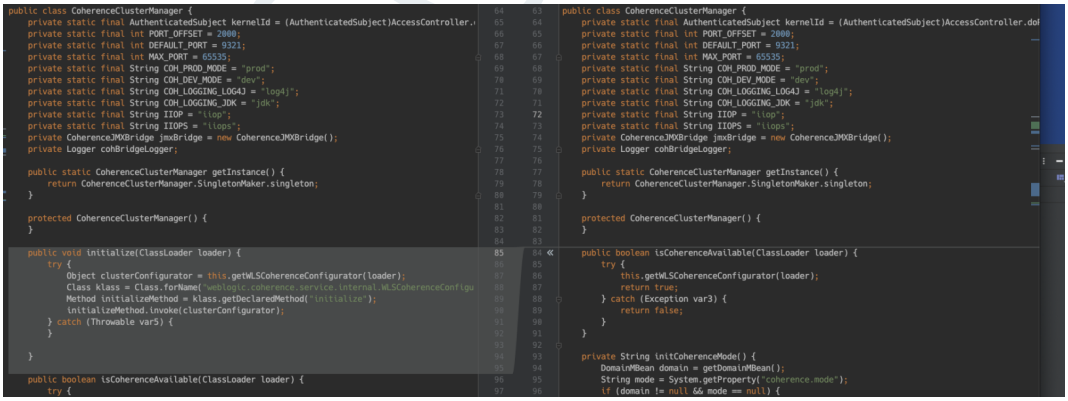
第二个就是基于 iiopinputstream 进行修复，这里只是简略的跟进 diff 了一下，在进

入序列化 IIOP 流程之后，第一幅图是没修复之前，s_streamfactory 是 DefaultObjectStreamFactory，而第二幅图是修复之后，s_streamfactory 被设置为了 WLSObjectStreamFactory



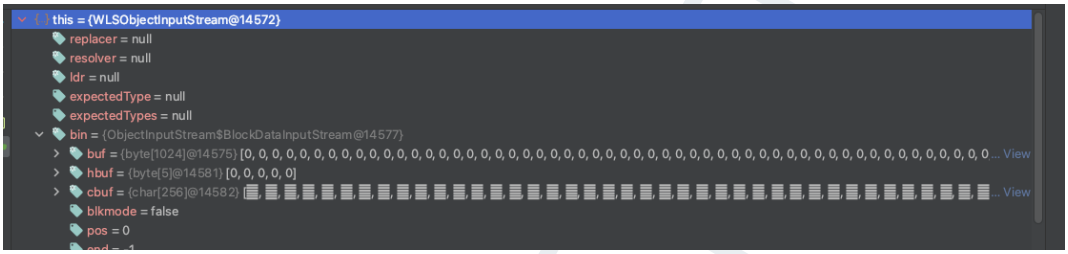
具体设置流程在 CoherenceClusterManager，这是 Weblogic 初始化的流程里

这里由于新增了 WLSCoherenceConfiguator 的初始化，就会给 s_streamfactory 赋值 WLSObjectStreamFactory

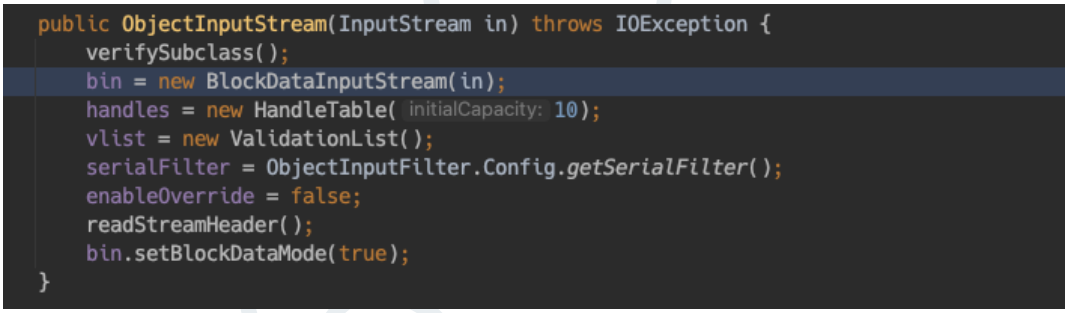


如果 s_streamfactory 是 WLSObjectStreamFactory，那么就会调用他的 getObject，会实例化一个 WLSObjectInputStream

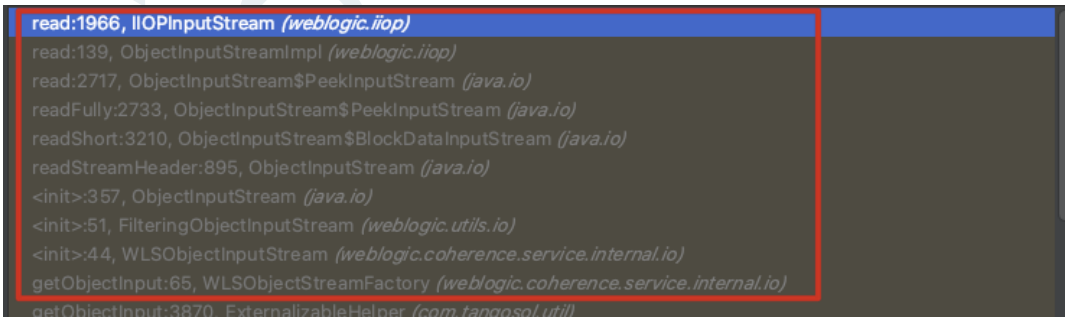
后续就会去读 WLSObjectInputStream 的 byte 数据



这个 byte 数据在初始化WLSObjectInputStream 的过程中是不会进行赋值的，于是是为空。



接着，后面在 read 的时候就会爆 eof 的错误。




public final int read(byte[] b, int off, int len) throws IOException {
if (this.eof()) {
return -1;
} else {
this.checkForRoomInChunk(bytesToRead: 0);
if (this.chunkingState.state == IIOPInputStream.State.in_a_chunk) {
len = min(len, this.chunkingState.getBytesLeftInChunk());
this.chunkingState.reduceChunkLength(len);
}
return this.rawInput.read(b, off, len);
}

4 时间线

2021-07-20 Oracle 发布安全更新通告

2021-07-21 360CERT 发布通告

2021-07-23 360CERT 发布分析

5 参考链接

Oracle Critical Patch Update Advisory - July 2021