安全漏洞分析

CVE-2021-2135：Weblogic 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-2021-2135：Weblogic 二次序列化漏洞分析		
报告类型	安全漏洞分析	报告编号	B6-2021-071903
报告版本	1	报告日期	2021-04-21
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	2021-04-21	360CERT	360CERT	撰写报告

目录

1 漏洞简述

2021 年 04 月 21 日，360CERT 监测发现 Oracle官方 发布了 2021年4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-2021-2135 ，漏洞等级：严重，漏洞评分：9.8。

未经身份验证的攻击者通过 T3 或 IIOP 协议发送恶意请求，最终接管服务器。

对此，360CERT 建议广大用户及时将 Weblogic 升级到最新版本。与此同时，请做好资产自查以及预防工作，以免遭受黑客攻击。

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

通过官方通告得知该漏洞依然是基于 Coherence 的反序列化，于是去 diffWeblogicFilterConfig，发现黑名单新增一个 coherence 类 com.tangosol.internal.util.SimpleBinaryEntry。

CVE-2021-2135	Oracle WebLogic Server	Coherence Container	T3, IIOP	Yes	9.8	Network	Low	None	None	Un-changed	High	High	High	12.1.3.0.0, 12.2.1.3.0, 12.2.1.4.0, 14.1.1.0.0

此次漏洞是基于 CVE-2020-14756 漏洞的绕过，采用的思路依然是进行二次反序列化，绕过 Weblogic 自身 jep290 的黑名单限制。在 SimpleBinaryEntry 里存在 getKey/getValue 方法，这两个方法都调用了 ExternalizableHelper.fromBinary 方法。

```
public K getKey() {
    K key = this.m_key;
    if (key == null) {
        key = this.m_key = ExternalizableHelper.fromBinary(this.m_binKey, this.getContextSerializer());
    }

    return key;
}
```

在 fromBinary 方法里又调用了 deserializeInternal，在该方法里会将 ReadBuffer 类型的输入流转换为 BufferInput 类型，传入 readObjectInternal 方法，

```
private static <T> T deserializeInternal(Serializer serializer, ReadBuffer buf, Function<BufferInput, BufferInput> supplierBufferIn, Class<T> clazz) throws IOException {
    BufferInput in = buf.getBufferInput();
    int nType = in.readUnsignedByte();
    switch(nType) {
    case 13:
        readInt(in);
        nType = in.readUnsignedByte();
        break;
    case 18:
    case 19:
        long nMask = nType == 18 ? (long)in.readByte() : in.readPackedLong();
        if ((nMask & 1L) == 0L) {
            throw new EOFException("Decorated value is missing a value");
        }

        int cb = in.readPackedInt();
        in = buf.getReadBuffer(in.getOffset(), cb).getBufferInput();
        nType = in.readUnsignedByte();
    }

    if (supplierBufferIn != null) {
        in = (BufferInput)supplierBufferIn.apply(in);
    }

    Object o = nType == 21 ? serializer.deserialize(in, clazz) : readObjectInternal(in, nType, ((ClassLoaderAware)serializer).getContextClassLoader());
    return realize(o, serializer);
}
```

接着，调用 readExternalizableLite 方法。

```
case 10:
    return readExternalizableLite(in, loader);
```

在 CVE-2020-14756 中，对该方法进行了修复。

```
public static ExternalizableLite readExternalizableLite(DataInput in, ClassLoader loader) throws IOException {
    ExternalizableLite value;
    if (in instanceof PofInputStream) {
        value = (ExternalizableLite)((PofInputStream)in).readObject();
    } else {
        String sClass = readUTF((DataInput)in);
        WrapperDataInputStream inWrapper = in instanceof WrapperDataInputStream ? (WrapperDataInputStream)in : null;

        try {
            Class<?> clz = loadClass(sClass, loader, inWrapper == null ? null : inWrapper.getClassLoader());
            if (in instanceof ObjectInputStream) {
                ObjectInputStream ois = (ObjectInputStream)in;
                if (!checkObjectInputFilter(clz, ois)) {
                    throw new InvalidClassException("Deserialization of class " + sClass + " was rejected");
                }
            }

            value = (ExternalizableLite)clz.newInstance();
        } catch (InstantiationException var7) {
            throw new IOException("Unable to instantiate an instance of class '" + sClass + "'; this is most likely due to a missing public no-args constructor: " +
        } catch (Exception var8) {
            throw new IOException("Class initialization failed: " + var8 + "\n" + getStackTrace(var8) + "\nClass: " + sClass + "\nClassLoader: " + loader + "\nConte
        }

        if (loader != null) {
            if (inWrapper == null) {
                in = new WrapperDataInputStream((DataInput)in, loader);
            } else if (loader != inWrapper.getClassLoader()) {
                inWrapper.setClassLoader(loader);
            }
        }

        value.readExternal((DataInput)in);
        if (value instanceof SerializerAware) {
            ((SerializerAware)value).setContextSerializer(ensureSerializer(loader));
        }
    }

    return value;
}
```

修复的原理是，只要输入流的类型是 ObjectInputStream，那么会去调用 checkObjectInputFilter 方法，而 checkObjectInputFilter 的具体实现其实就是基于 jep290 的检测。

```
protected static boolean checkObjectInputFilter(Class<?> clz, ObjectInputStream ois) {
    try {
        Object filter = HANDLE_GET_FILTER == null ? null : HANDLE_GET_FILTER.invoke(ois);
        if (filter == null) {
            return true;
        }

        ExternalizableHelper.DynamicFilterInfo dynamic = (ExternalizableHelper.DynamicFilterInfo)s_tloHandler.get();
        dynamic.setClass(clz);
        Object oFilterInfo = dynamic.getFilterInfo();
        Enum status = HANDLE_CHECKINPUT.invoke(filter, oFilterInfo);
        dynamic.setClass((Class)null);
        return status != null && !status.name().equals("REJECTED");
    } catch (IllegalAccessException | InvocationTargetException | ClassNotFoundException var6) {
        err( s: "Unable to invoke checkInput on objectInputFilter ");
        err(var6);
    } catch (Throwable var7) {
    }

    return false;
}
```

但是，当我们通过 BufferInput 流进行反序列化的时候，是不会被调用到 checkObjectInputFilter 方法的，于是绕过了之前的补丁。

3.1 POC 构造

SimpleBinaryEntry 的 getKey 方法在自身重写的 toString 方法内进行了调用。

```
public String toString() {
    return "SimpleBinaryEntry(key=\"" + this.getKey() + "\", value=\"" + this.getValue() + "\")";
}
```

那么首先要想办法触发 toString 方法，最开始我想到的是 BadAttributeValueExpException，直接反序列化的时候就会调用对象的 toString 方法，不过这样就会有问题。在最终调用 serializer.deserialize 进行二次序列化的时候， serializer 会因为没有赋值而报错，SimpleBinaryEntry 自身有一个 setContextSerializer 方法，需要有地方能调用。

```
private static <T> T deserializeInternal(Serializer serializer, ReadBuffer buf, Function<BufferInput, BufferInput> supplierBufferIn, Class<T> clazz) throws IOException {
    BufferInput in = buf.getBufferInput();  in: AbstractByteArrayReadBuffer$ByteArrayBufferInput@1256
    int nType = in.readUnsignedByte();  nType: 172
    switch(nType) {
    case 13:
        readInt(in);
        nType = in.readUnsignedByte();
        break;
    case 18:
    case 19:
        long nMask = nType == 18 ? (long)in.readByte() : in.readPackedLong();
        if ((nMask & 1L) == 0L) {
            throw new EOFException("Decorated value is missing a value");
        }

        int cb = in.readPackedInt();
        in = buf.getReadBuffer(in.getOffset(), cb).getBufferInput();  buf: "Binary(length=771, value=0xACED00057372004D636F6D2E6265612E636F72652E726570061636B616765642E737072696E
        nType = in.readUnsignedByte();
    }

    if (supplierBufferIn != null) {
        in = (BufferInput)supplierBufferIn.apply(in);  supplierBufferIn: null
    }

    Object o = nType == 21 ? serializer.deserialize(in, clazz) : readObjectInternal(in, nType, ((ClassLoaderAware)serializer).getContextClassLoader());  serializer: null    cla
    return realize(o, serializer);
}
```

serializer 的赋值在 ExternalizableHelper#readExternalizableLite 。

```
public static ExternalizableLite readExternalizableLite(DataInput in, ClassLoader loader) throws IOException {
    ExternalizableLite value;
    if (in instanceof PofInputStream) {
        value = (ExternalizableLite)((PofInputStream)in).readObject();
    } else {
        String sClass = readUTF((DataInput)in);
        WrapperDataInputStream inWrapper = in instanceof WrapperDataInputStream ? (WrapperDataInputStream)in : null;

        try {
            value = (ExternalizableLite)loadClass(sClass, loader, inWrapper == null ? null : inWrapper.getClassLoader()).newInsta
        } catch (InstantiationException var6) {
            throw new IOException("Unable to instantiate an instance of class '" + sClass + "'; this is most likely due to a miss
        } catch (Exception var7) {
            throw new IOException("Class initialization failed: " + var7 + "\n" + getStackTrace(var7) + "\nClass: " + sClass + "\
        }

        if (loader != null) {
            if (inWrapper == null) {
                in = new WrapperDataInputStream((DataInput)in, loader);
            } else if (loader != inWrapper.getClassLoader()) {
                inWrapper.setClassLoader(loader);
            }
        }

        value.readExternal((DataInput)in);
        if (value instanceof SerializerAware) {
            ((SerializerAware)value).setContextSerializer(ensureSerializer(loader));
        }
    }
```

所以，还是需要先有流程走到 ExternalizableHelper#readExternalizableLite，这里只需要 value 为 SimpleBinaryEntry 就能设置 serializer 。参考了漏洞发现者的文章，toString 方法能够在 com.sun.org.apache.xpath.internal.objects.XString 重写的 equals 方法里调用。

```
public boolean equals(Object obj2)
{

  if (null == obj2)
    return false;

    // In order to handle the 'all' semantics of
    // nodeset comparisons, we always call the
    // nodeset function.
  else if (obj2 instanceof XNodeSet)
    return obj2.equals(this);
  else if(obj2 instanceof XNumber)
      return obj2.equals(this);
  else
    return str().equals(obj2.toString());
}
```

而ExternalizableHelper#readMap 方法里调用了 map.put，熟悉反序列化的应该都很熟悉 put 方法，在 map.put 方法里会触发 key 的 equals 方法的调用，而这里的 oKey 和 oVal 都是可以通过序列化进行控制的，只需要 oKey 是 XString 类型即可。

```
public static int readMap(DataInput in, Map map, ClassLoader loader) throws IOException {
    int cEntries;
    if (in instanceof PofInputStream) {
        PofInputStream inPof = (PofInputStream)in;
        inPof.getPofReader().readMap(inPof.nextIndex(), map);
        cEntries = map.size();
    } else {
        cEntries = in.readInt();

        for(int i = 0; i < cEntries; ++i) {
            Object oKey = readObject(in, loader);
            Object oVal = readObject(in, loader);
            map.put(oKey, oVal);
        }
    }

    return cEntries;
}
```

接着还需要有一个调用 readMap 的地方，com.tangosol.util.processor.ConditionalPutAll 的 readExternal 方法中刚好就存在 readMap 的调用。

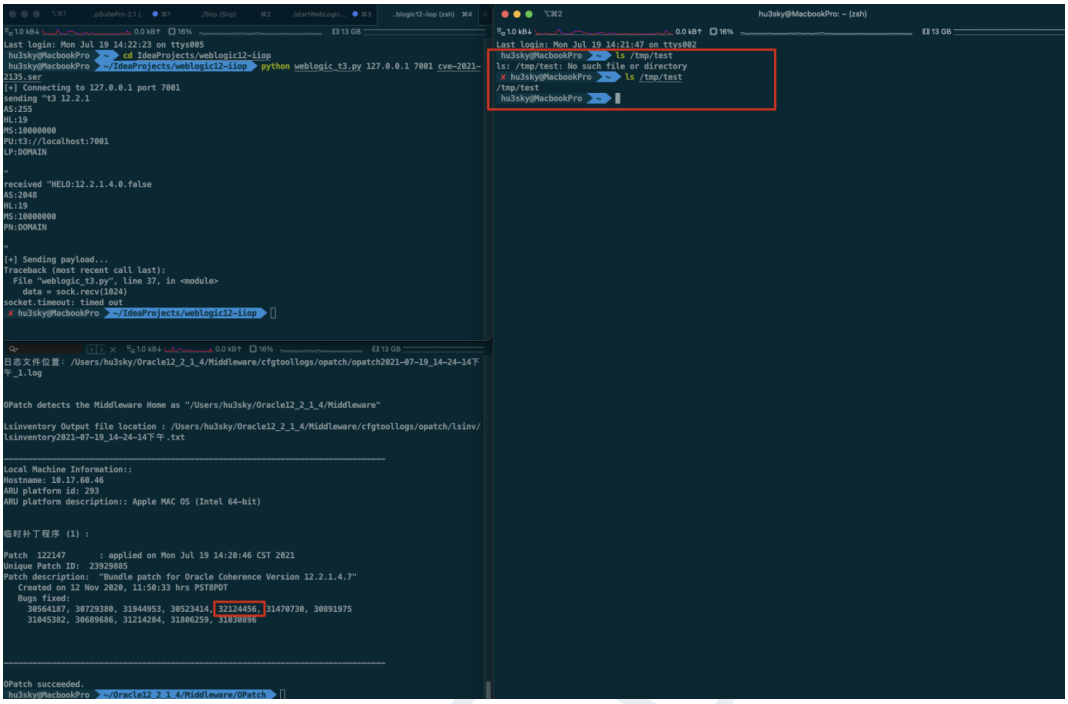
```
public void readExternal(DataInput in) throws IOException {   in: ObjectInputStream@1134
    this.m_filter = (Filter)ExternalizableHelper.readObject(in);   m_filter: "MapEventFilter(mask)"
    Map map = this.m_map = new LiteMap();   map:  size = 1    m_map:  size = 1
    ExternalizableHelper.readMap(in, map, (ClassLoader)null);   in: ObjectInputStream@1134    map:  size = 1
}
```

看似已经没有问题了，但是这个类并不是 Externalizable 的子类，所以在反序列化的调用中，无法自动调用到 readExternal 方法，所以还是需要找另外一个入口来主动调用他的 readExternal 。这里就可以参考 CVE-2020-14756 了，入口类 com.tangosol.coherence.servlet.AttributeHolder 没有加到黑名单里。

完整的调用栈如下：

3.2 漏洞证明

PatchID:32124456



Oracle WebLogic Server and Coherence Oracle Fusion Middleware Infrastructure (WebLogic Server for FMW)	Coherence 12.2.1.4.7 Patch 32124456 or later	CVE-2020-14756

3.3 漏洞修复

第一个修复的点依旧是针对黑名单，加入了 SimpleBinaryEntry 类。第二个修复点就是本次修复大改的地方，加入了白名单机制，只允许白名单里的类或其子类通过白名单。打了补丁后，再次发送攻击的包会报错，调用栈在 log 里清晰可见：

```
java.io.InvalidClassException: Expected type of [ , weblogic.rmi.spi.ServiceContext, weblogic.rjvm.ClassTableEntry,
weblogic.rjvm.JVMID, weblogic.security.acl.internal.AuthenticatedUser, weblogic.rmi.extensions.server.RuntimeMethodDescriptor,
weblogic.utils.io.Immutable] but actual type was com.tangosol.coherence.servlet.AttributeHolder
    at weblogic.utils.io.FilteringObjectInputStream.validateReturnType(FilteringObjectInputStream.java:172)
    at weblogic.utils.io.FilteringObjectInputStream.resolveClass(FilteringObjectInputStream.java:66)
    at weblogic.rjvm.InboundMsgAbbrev$ServerChannelInputStream.resolveClass(InboundMsgAbbrev.java:143)
    at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1922)
    at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1805)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2096)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1624)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:464)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:422)
    at weblogic.utils.io.FilteringObjectInputStream.readObjectValidated(FilteringObjectInputStream.java:128)
    at weblogic.rjvm.InboundMsgAbbrev.readObject(InboundMsgAbbrev.java:89)
    at weblogic.rjvm.InboundMsgAbbrev.read(InboundMsgAbbrev.java:55)
    at weblogic.rjvm.MsgAbbrevJVMConnection.readMsgAbbrevs(MsgAbbrevJVMConnection.java:335)
    at weblogic.rjvm.MsgAbbrevInputStream.init(MsgAbbrevInputStream.java:233)
    at weblogic.rjvm.MsgAbbrevJVMConnection.dispatch(MsgAbbrevJVMConnection.java:569)
    at weblogic.rjvm.t3.MuxableSocketT3.dispatch(MuxableSocketT3.java:666)
    at weblogic.socket.BaseAbstractMuxableSocket.dispatch(BaseAbstractMuxableSocket.java:397)
    at weblogic.socket.SocketMuxer.readReadySocketOnce(SocketMuxer.java:993)
    at weblogic.socket.SocketMuxer.readReadySocket(SocketMuxer.java:929)
    at weblogic.socket.NIOSocketMuxer.process(NIOSocketMuxer.java:599)
    at weblogic.socket.NIOSocketMuxer.processSockets(NIOSocketMuxer.java:563)
    at weblogic.socket.SocketReaderRequest.run(SocketReaderRequest.java:30)
    at weblogic.socket.SocketReaderRequest.execute(SocketReaderRequest.java:43)
    at weblogic.kernel.ExecuteThread.execute(ExecuteThread.java:147)
    at weblogic.kernel.ExecuteThread.run(ExecuteThread.java:119)
```

于是我们看到 weblogic.rjvm.InboundMsgAbbrev ， 新增了一个白名单字段。

```
1 private static final Class[] ABBREV_CLASSES = new Class[]{String.class,
  ↪ ServiceContext.class, ClassTableEntry.class, JVMID.class,
  ↪ AuthenticatedUser.class, RuntimeMethodDescriptor.class,
  ↪ Immutable.class};
```

在InboundMsgAbbrev里的 readObject 方法将白名单传入了 readObjectValidated 方法

```
private Object readObject(MsgAbbrevInputStream in) throws IOException, ClassNotFoundException {
    int typecode = in.read();
    switch(typecode) {
    case 0:
        return (new InboundMsgAbbrev.ServerChannelInputStream(in)).readObjectValidated(ABBREV_CLASSES);
    case 1:
        return in.readASCII();
    default:
        throw new StreamCorruptedException("Unknown typecode: '" + typecode + "'");
    }
}
```

会设置 expectedType ， 即白名单 List。

```
public Object readObjectValidated(Class[] returnTypes) throws IOException, ClassNotFoundException {
    Object var2;
    try {
        this.expectedTypes = returnTypes;
        var2 = this.readObject();
    } catch (Exception var6) {
        throw var6;
    } finally {
        this.expectedTypes = null;
    }

    return var2;
}
```

T3 自身的序列化流程走完之后，就开始走 ObjectInputStream ，在调用到 ServerChannelInputStream 重写的 resolveClass 的时候，先进行黑名单检测。

```
protected Class resolveClass(ObjectStreamClass descriptor) throws ClassNotFoundException, IOException {
    try {
        this.checkLegacyBlacklistIfNeeded(descriptor.getName());
    } catch (InvalidClassException var4) {
        throw var4;
    }

    Class c = super.resolveClass(descriptor);
    if (c == null) {
        throw new ClassNotFoundException("super.resolveClass returns null.");
    } else {
        ObjectStreamClass localDesc = ObjectStreamClass.lookup(c);
        if (localDesc != null && localDesc.getSerialVersionUID() != descriptor.getSerialVersionUID()) {
            throw new ClassNotFoundException("different serialVersionUID. local: " + localDesc.getSerialVersionUID() + " remote: " + descriptor.getSerialVersionUID());
        } else {
            return c;
        }
    }
}
```

如果不在黑名单里，继续调用 super.resolveClass ，这个时候会调用 validateReturnType 方法。

```
protected Class<?> resolveClass(ObjectStreamClass descriptor) throws ClassNotFoundException, IOException {
    this.checkLegacyBlacklistIfNeeded(descriptor.getName());
    Class cls = super.resolveClass(descriptor);
    this.validateReturnType(cls);
    return cls;
}
```

这里，进行了白名单检测，从expectedType 取值，只要不是白名单里的子类型，那么就会直接报错。

```
public void validateReturnType(Class returnType) throws InvalidClassException {
    if (this.expectedType != null || this.expectedTypes != null) {
        try {
            if (returnType != null) {
                if (this.expectedType == null) {
                    for(int i = 0; i < this.expectedTypes.length; ++i) {
                        if (this.expectedTypes[i].isAssignableFrom(returnType)) {
                            return;
                        }
                    }

                    String classTypes = "[ ";

                    for(int i = 0; i < this.expectedTypes.length; ++i) {
                        if (i > 0) {
                            classTypes = classTypes + ", " + this.expectedTypes[i].getName();
                        }
                    }

                    classTypes = classTypes + "]";
                    throw new InvalidClassException("Expected type of " + classTypes + " but actual type was " + returnType.getName());
                }

                if (!this.expectedType.isAssignableFrom(returnType)) {
                    throw new InvalidClassException("Expected type of " + this.expectedType.getName() + " but actual type was " + returnType.getName());
                }
            }
        } finally {
            this.expectedType = null;
            this.expectedTypes = null;
        }

    }
}
```

4 时间线

2021-04-20 Oracle 发布安全更新通告

2021-04-21 360CERT 发布通告

2021-07-19 360CERT 发布分析

5 参考链接

Oracle Critical Patch Update Advisory - April 2021

How Did I Find Weblogic T3 RCE