Java反序列化漏洞之殇

cryin / 2018-02-08 14:58:00 / 浏览数 7700 技术文章 技术文章 顶(0) 踩(0)

关于反序列化漏洞分析及利用研究的文章不少，但鲜有检测及修复方面的介绍，本文旨站在应用安全的角度，从安全编码、代码审计、漏洞检测及修复方案对反序列化漏

概述

序列化是让Java对象脱离Java运行环境的一种手段，可以有效的实现多平台之间的通信、对象持久化存储。

Java 序列化是指把 Java 对象转换为字节序列的过程便于保存在内存、文件、数据库中，ObjectOutputStream类的 writeObject()
方法可以实现序列化。反序列化是指把字节序列恢复为 Java 对象的过程，ObjectInputStream 类的 readObject() 方法用于反序列化。

漏洞成因

序列化和反序列化本身并不存在问题。但当输入的反序列化的数据可被用户控制，那么攻击者即可通过构造恶意输入，让反序列化产生非预期的对象，在此过程中执行构造的

漏洞代码示例如下：

```
......
//■■■■■,■■■■■
InputStream in=request.getInputStream();
ObjectInputStream ois = new ObjectInputStream(in);
//■■■■
ois.readObject();
ois.close();
```

这里特别要注意的是非预期的对象，正因为此java标准库及大量第三方公共类库成为反序列化漏洞利用的关键。安全研究人员已经发现大量利用反序列化漏洞执行任意代码的
Lawrence和Chris Frohoff在《Marshalling Pickles how deserializing objects can ruin your day》中提出的利用Apache Commons
Collection实现任意代码执行。此后安全研究人员也陆续爆出XML、Json、Yaml等反序列化的相关漏洞。

除了commons-collections 3.1可以用来利用java反序列化漏洞，还有更多第三方库同样可以用来利用反序列化漏洞并执行任意代码，部分如下：

- commons-fileupload 1.3.1
- commons-io 2.4
- commons-collections 3.1
- commons-logging 1.2
- commons-beanutils 1.9.2
- org.slf4j:slf4j-api 1.7.21
- com.mchange:mchange-commons-java 0.2.11
- org.apache.commons:commons-collections 4.0
- com.mchange:c3p0 0.9.5.2
- org.beanshell:bsh 2.0b5
- org.codehaus.groovy:groovy 2.3.9
- ......

Java反序列化详解

序列化数据结构

通过查看序列化后的数据，可以看到反序列化数据开头包含两字节的魔术数字，这两个字节始终为十六进制的0xAC ED。接下来是两字节的版本号0x00
05的数据。此外还包含了类名、成员变量的类型和个数等。

这里以类SerialObject示例来详细进行介绍Java对象序列化后的数据结构：

```
public class SerialObject implements Serializable{
   private static final long serialVersionUID = 5754104541168322017L;

   private int id;
   public String name;

   public SerialObject(int id,String name){
       this.id=id;
       this.name=name;
   }
   ...
}
```

序列化SerialObject实例后以二进制格式查看：

```
00000000: aced 0005 7372 0024 636f 6d2e 7878 7878  ....sr.$com.xxxx
00000010: 7878 2e73 6563 2e77 6562 2e68 6f6d 652e  xx.sec.web.home.
00000020: 5365 7269 616c 4f62 6a65 6374 4fda af97  SerialObjectO...
00000030: f8cc c5e1 0200 0249 0002 6964 4c00 046e  .......I..idL..n
00000040: 616d 6574 0012 4c6a 6176 612f 6c61 6e67  amet..Ljava/lang
00000050: 2f53 7472 696e 673b 7870 0000 07e1 7400  /String;xp....t.
00000060: 0563 7279 696e 0a                        .cryin.
```

序列化的数据流以魔术数字和版本号开头，这个值是在调用ObjectOutputStream序列化时，由writeStreamHeader方法写入：

```
protected void writeStreamHeader() throws IOException {
    bout.writeShort(STREAM_MAGIC);//STREAM_MAGIC (2 bytes) 0xACED
    bout.writeShort(STREAM_VERSION);//STREAM_VERSION (2 bytes) 5
   }
```

序列化后的SerialObject对象详细结构：

```
STREAM_MAGIC (2 bytes) 0xACED
STREAM_VERSION (2 bytes) 0x0005
   TC_OBJECT (1 byte) 0x73
       TC_CLASSDESC (1 byte) 0x72
       className
           length (2 bytes) 0x24 = 36
           text (36 bytes) com.xxxxxx.sec.web.home.SerialObject
       serialVersionUID (8 bytes) 0x4FDAAF97F8CCC5E1 = 5754104541168322017
       classDescInfo
           classDescFlags (1 byte) 0x02 = SC_SERIALIZABLE
           fields
               count (2 bytes) 2
               field[0]
                   primitiveDesc
                       prim_typecode (1 byte) I = integer
                       fieldName
                           length (2 bytes) 2
                           text (2 bytes) id
               field[1]
                   objectDesc
                       obj_typecode (1 byte) L = object
                       fieldName
                           length (2 bytes) 4
                           text (4 bytes)  name
                       className1
                           TC_STRING (1 byte) 0x74
                               length (2 bytes) 0x12 = 18
                               text (18 bytes) Ljava/lang/String;

           classAnnotation
               TC_ENDBLOCKDATA (1 byte) 0x78

           superClassDesc
               TC_NULL (1 byte) 0x70
   classdata[]
       classdata[0] (4 bytes) 0xe107 = id = 2017
       classdata[1]
           TC_STRING (1 byte) 0x74
           length (2 bytes) 5
           text (8 bytes) cryin
```

反序列化过程详解

Java程序中类ObjectInputStream的readObject方法被用来将数据流反序列化为对象，如果流中的对象是class，则它的ObjectStreamClass描述符会被读取，并返回相应的

如果类描述符是动态代理类，则调用resolveProxyClass方法来获取本地类。如果不是动态代理类则调用resolveClass方法来获取本地类。如果无法解析该类，则抛出ClassN

如果反序列化对象不是String、array、enum类型，ObjectStreamClass包含的类会在本地被检索，如果这个本地类没有实现java.io.Serializable或者externalizable接口，则

反序列化漏洞检测方案

代码审计

反序列化操作一般在导入模版文件、网络通信、数据传输、日志格式化存储、对象数据落磁盘或DB存储等业务场景,在代码审计时可重点关注一些反序列化操作函数并判断输

```
ObjectInputStream.readObject
ObjectInputStream.readUnshared
XMLDecoder.readObject
Yaml.load
XStream.fromXML
ObjectMapper.readValue
JSON.parseObject
...
```

同时也要关注存在漏洞的第三方库及版本是否安全。

进阶审计

对于直接获取用户输入进行反序列化操作这种点比较好审计并发现，目前反序列化漏洞已经被谈起太多次了，所以有经验的开发都会在代码中有相应的修复。但并不是所有

代码中有使用到反序列化操作，那自身项目工程中肯定存在可以被反序列化的类，包括Java自身、第三方库有大量这样的类，可被反序列化的类有一个特点，就是该类必定实
接口是启用其序列化功能的接口，实现 java.io.Serializable 接口的类才是可序列化的。一个典型的示例如下：

```
public class SerialObject implements Serializable{
   private static final long serialVersionUID = 5754104541168322017L;

   private int id;
   public String name;

   public SerialObject(int id,String name){
       this.id=id;
       this.name=name;
   }

   public void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException{
       //■■■■■readObject()■■
       in.defaultReadObject();
   }
}
```

所以在代码审计时对这些类也可进行特别关注，分析并确认是否有可能被发序列化漏洞利用执行任意代码。发现新的可利用的类即可突破使用黑名单进行校验的一些应用。

白盒检测

大型企业的应用很多，每个都人工去审计不现实，往往都有相应的自动化静态代码审计工具，这里以ObjectInputStream.readObject()为例，其它原理也相似。在自动化检
检测方式可参考lgtm.com对于Deserialization of user-controlled data的实现:

```
/**
* @name Deserialization of user-controlled data
* @description Deserializing user-controlled data may allow attackers to
*              execute arbitrary code.
* @kind problem
* @problem.severity error
* @precision high
* @id java/unsafe-deserialization
* @tags security
*       external/cwe/cwe-502
*/
import java
import semmle.code.java.security.DataFlow
import semmle.code.java.frameworks.Kryo
import semmle.code.java.frameworks.XStream
import semmle.code.java.frameworks.SnakeYaml

class ObjectInputStreamReadObjectMethod extends Method {
 ObjectInputStreamReadObjectMethod() {
   this.getDeclaringType().getASourceSupertype*().hasQualifiedName("java.io", "ObjectInputStream") and
   (this.hasName("readObject") or this.hasName("readUnshared"))
 }
}

class XMLDecoderReadObjectMethod extends Method {
 XMLDecoderReadObjectMethod() {
   this.getDeclaringType().hasQualifiedName("java.beans", "XMLDecoder") and
```

```
   this.hasName("readObject")
 }
}

class SafeXStream extends FlowSource {
 SafeXStream() {
   any(XStreamEnableWhiteListing ma).getQualifier().(VarAccess).getVariable().getAnAccess() = this
 }
}

class SafeKryo extends FlowSource {
 SafeKryo() {
   any(KryoEnableWhiteListing ma).getQualifier().(VarAccess).getVariable().getAnAccess() = this
 }
}

predicate unsafeDeserialization(MethodAccess ma, Expr sink) {
 exists(Method m | m = ma.getMethod() |
   m instanceof ObjectInputStreamReadObjectMethod and
   sink = ma.getQualifier()
   or
   m instanceof XMLDecoderReadObjectMethod and
   sink = ma.getQualifier()
   or
   m instanceof XStreamReadObjectMethod and
   sink = ma.getAnArgument() and
   not exists(SafeXStream sxs | sxs.flowsTo(ma.getQualifier()))
   or
   m instanceof KryoReadObjectMethod and
   sink = ma.getAnArgument() and
   not exists(SafeKryo sk | sk.flowsTo(ma.getQualifier()))
   or
   ma instanceof UnsafeSnakeYamlParse and
   sink = ma.getArgument(0)
 )
}

class UnsafeDeserializationSink extends Expr {
 UnsafeDeserializationSink() {
   unsafeDeserialization(_, this)
 }
 MethodAccess getMethodAccess() { unsafeDeserialization(result, this) }
}

from UnsafeDeserializationSink sink, RemoteUserInput source
where source.flowsTo(sink)
select sink.getMethodAccess(), "Unsafe deserialization of $@.", source, "user input"
```

黑盒检测

调用ysoserial并依次生成各个第三方库的利用payload(也可以先分析依赖第三方包量，调用最多的几个库的paylaod即可)，该payload构造为访问特定url链接的payload，

```
java -jar ysoserial.jar CommonsCollections1 'curl " + URL + " '
```

也可通过DNS解析记录确定漏洞是否存在。现成的轮子很多，推荐NickstaDB写的SerialBrute，还有一个针对RMI的测试工具BaRMIe，也很不错~。.

RASP检测

Java程序中类ObjectInputStream的readObject方法被用来将数据流反序列化为对象，如果流中的对象是class，则它的ObjectStreamClass描述符会被读取，并返回相应的

类的名称及serialVersionUID的ObjectStreamClass描述符在序列化对象流的前面位置，且在readObject反序列化时首先会调用resolveClass读取反序列化的类名，所以RA

百度的开源RASP产品就是使用的这种方法，具体可参考其DeserializationHook.java的实现:

```
@Override
   protected MethodVisitor hookMethod(int access, String name, String desc,
                                      String signature, String[] exceptions, MethodVisitor mv) {
       if ("resolveClass".equals(name) && "(Ljava/io/ObjectStreamClass;)Ljava/lang/Class;".equals(desc)) {
           return new AdviceAdapter(Opcodes.ASM5, mv, access, name, desc) {
               @Override
               protected void onMethodEnter() {
```

```
                    loadArg(0);
                    invokeStatic(Type.getType(HookHandler.class),
                            new Method("checkDeserializationClass", "(Ljava/io/ObjectStreamClass;)V"));
                }
            };
        }
        return mv;
    }
```

其中检测覆盖的反序列化类黑名单如下:

```
plugin.register('deserialization', function (params, context) {
    var deserializationInvalidClazz = [
        'org.apache.commons.collections.functors.InvokerTransformer',
        'org.apache.commons.collections.functors.InstantiateTransformer',
        'org.apache.commons.collections4.functors.InvokerTransformer',
        'org.apache.commons.collections4.functors.InstantiateTransformer',
        'org.codehaus.groovy.runtime.ConvertedClosure',
        'org.codehaus.groovy.runtime.MethodClosure',
        'org.springframework.beans.factory.ObjectFactory',
        'xalan.internal.xsltc.trax.TemplatesImpl'
    ]

    var clazz = params.clazz
    for (var index in deserializationInvalidClazz) {
        if (clazz === deserializationInvalidClazz[index]) {
            return {
                action:     'block',
                message:    '[redacted]',
                confidence: 100
            }
        }
    }
    return clean
})
```

攻击检测

通过查看反序列化后的数据，可以看到反序列化数据开头包含两字节的魔术数字，这两个字节始终为十六进制的0xAC
ED。接下来是两字节的版本号。我只见到过版本号为5（0x00
05）的数据。考虑到zip、base64各种编码，在攻击检测时可针对该特征进行匹配请求post中是否包含反序列化数据，判断是否为反序列化漏洞攻击。

```
xxxdeMacBook-Pro:demo xxx$ xxd objectexp
    00000000: aced 0005 7372 0032 7375 6e2e 7265 666c  ....sr.2sun.refl
    00000010: 6563 742e 616e 6e6f 7461 7469 6f6e 2e41  ect.annotation.A
    00000020: 6e6e 6f74 6174 696f 6e49 6e76 6f63 6174  nnotationInvocat
    00000030: 696f 6e48 616e 646c 6572 55ca f50f 15cb  ionHandlerU.....
```

但仅从特征匹配只能确定有攻击尝试请求，还不能确定就存在反序列化漏洞，还要结合请求响应、返回内容等综合判断是否确实存在漏洞。

Java反序列化漏洞修复方案

通过Hook resolveClass来校验反序列化的类

通过上面序列化数据结构可以了解到包含了类的名称及serialVersionUID的ObjectStreamClass描述符在序列化对象流的前面位置，且在readObject反序列化时首先会调用r
Ernst在2013年提出《Look-ahead Java deserialization》，具体实现代码示例如下:

```
public class AntObjectInputStream extends ObjectInputStream{
    public AntObjectInputStream(InputStream inputStream)
            throws IOException {
        super(inputStream);
    }

    /**
     * [redacted]SerialObject class
     */
    @Override
    protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException,
            ClassNotFoundException {
        if (!desc.getName().equals(SerialObject.class.getName())) {
```

```
            throw new InvalidClassException(
                    "Unauthorized deserialization attempt",
                    desc.getName());
        }
        return super.resolveClass(desc);
    }
}
```

通过此方法，可灵活的设置允许反序列化类的白名单，也可设置不允许反序列化类的黑名单。但反序列化漏洞利用方法一直在不断的被发现，黑名单需要一直更新维护，且无

SerialKiller 是由Luca Carettoni利用上面介绍的方法实现的反序列化类白/黑名单校验的jar包。具体使用方法可参考其代码仓库。

contrast-rO0是一个轻量级的agent程序，通过通过重写ObjectInputStream来防御反序列化漏洞攻击。使用其中的SafeObjectInputStream类来实现反序列化类白/黑名单

```
SafeObjectInputStream in = new SafeObjectInputStream(inputStream, true);
in.addToWhitelist(SerialObject.class);

in.readObject();
```

使用ValidatingObjectInputStream来校验反序列化的类

使用Apache Commons IO
Serialization包中的ValidatingObjectInputStream类的accept方法来实现反序列化类白/黑名单控制，具体可参考ValidatingObjectInputStream介绍；示例代码如下:

```
private static Object deserialize(byte[] buffer) throws IOException,
ClassNotFoundException , ConfigurationException {
   Object obj;
   ByteArrayInputStream bais = new ByteArrayInputStream(buffer);
   // Use ValidatingObjectInputStream instead of InputStream
   ValidatingObjectInputStream ois = new   ValidatingObjectInputStream(bais);

   //■■■■■■■SerialObject class
   ois.accept(SerialObject.class);
   obj = ois.readObject();
   return obj;
}
```

使用ObjectInputFilter来校验反序列化的类

Java
9包含了支持序列化数据过滤的新特性，开发人员也可以继承java.io.ObjectInputFilter类重写checkInput方法实现自定义的过滤器，，并使用ObjectInputStream对象的se

```
import java.util.List;
import java.util.Optional;
import java.util.function.Function;
import java.io.ObjectInputFilter;
class BikeFilter implements ObjectInputFilter {
   private long maxStreamBytes = 78; // Maximum allowed bytes in the stream.
   private long maxDepth = 1; // Maximum depth of the graph allowed.
   private long maxReferences = 1; // Maximum number of references in a graph.
   @Override
   public Status checkInput(FilterInfo filterInfo) {
       if (filterInfo.references() < 0 || filterInfo.depth() < 0 || filterInfo.streamBytes() < 0 || filterInfo.references() >
           return Status.REJECTED;
       }
       Class<?> clazz = filterInfo.serialClass();
       if (clazz != null) {
           if (SerialObject.class == filterInfo.serialClass()) {
               return Status.ALLOWED;
           }
           else {
               return Status.REJECTED;
           }
       }
       return Status.UNDECIDED;
   } // end checkInput
} // end class BikeFilter
```

上述示例代码，仅允许反序列化SerialObject类对象，上述示例及更多关于ObjectInputFilter的均参考自NCC Group Whitepaper由Robert C. Seacord写的《Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS)》

黑名单校验修复

在反序列化时设置类的黑名单来防御反序列化漏洞利用及攻击，这个做法在源代码修复的时候并不是推荐的方法，因为你不能保证能覆盖所有可能的类，而且有新的利用pay

但有某些场景下可能黑名单是一个不错的选择。写代码的时候总会把一些经常用到的方法封装到公共类，这样其它工程中用到只需要导入jar包即可，此前已经见到很多提供

- org.apache.commons.collections.functors.InvokerTransformer
- org.apache.commons.collections.functors.InstantiateTransformer
- org.apache.commons.collections4.functors.InvokerTransformer
- org.apache.commons.collections4.functors.InstantiateTransformer
- org.codehaus.groovy.runtime.ConvertedClosure
- org.codehaus.groovy.runtime.MethodClosure
- org.springframework.beans.factory.ObjectFactory
- com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl
- org.apache.commons.fileupload
- org.apache.commons.beanutils
- ...

安全编码建议

- 更新commons-collections、commons-io等第三方库版本；
- 业务需要使用反序列化时，尽量避免反序列化数据可被用户控制，如无法避免建议尽量使用白名单校验的修复方式；

总结

关于反序列化漏洞分析及利用研究的文章不少，但鲜有检测及修复方面的介绍，本文旨站在应用安全的角度，从安全编码、代码审计、漏洞检测及修复方案对反序列化漏洞进

参考

- https://www.nccgroup.trust/us/our-research/combating-java-deserialization-vulnerabilities-with-look-ahead-object-input-streams-laois/
- https://dzone.com/articles/a-first-look-into-javas-new-serialization-filterin
- https://docs.oracle.com/javase/7/docs/platform/serialization/spec/protocol.html
- https://www.owasp.org/index.php/Deserialization_of_untrusted_data
- https://github.com/Cryin/Paper/blob/master/Java%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90%
- https://www.ibm.com/developerworks/library/se-lookahead/
- https://github.com/baidu/openrasp

上一篇：Java反序列化备忘录 下一篇：【Struts2-命令-代码执行漏...

1. 3 条回复


b5mali4 2018-02-09 10:56:48

收藏，深度好文

0 回复Ta

[71428****@qq.com](#) 2018-09-28 17:02:32

■■■■ObjectInputStream■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

这里我是存疑的，使用了readobject，流的来源可被污染并不能说明存在反序列化漏洞。反序列化过去的流只有类名和对象状态而不是字节码，这里需要借用common collections等组件构建利用和发现新“gadget ”。而不是说没有合理实现resolveClass等修复方案就是漏洞（spring-tx.jar这个漏洞特点是流可控）。

另外审计时，也可以看看readStreamHeader、readClassDescriptor是否被修改，这样会导致序列化poc失败，需要进行对应的调整。

1 回复Ta

 [niexinming](#) 2018-11-02 01:04:59

马克

0 回复Ta

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)