TONGDAO
同 道
KCon
2024
JAVA反序列化新思路
汇报人：2rpang
时间：2024.08.24

目录

CONTENT

01 Java反序列化简介
MANDAMUS MEDIOCREM

02 常见防御思路与思考
MANDAMUS MEDIOCREM

03 以点带面深入分析
MANDAMUS MEDIOCREM

04 代理模式下的利用
MANDAMUS MEDIOCREM

KCon 2024

KCon
PART ONE
01
Java反序列化简介
MANDAMUS MEDIOCREM REREHENDUNT

Java反序列化简介

MANT EUM E

01
Kcon 2024

序列化

将Java对象转化为字节存储于内存或磁盘中，保存对象的过程

02
Kcon 2024

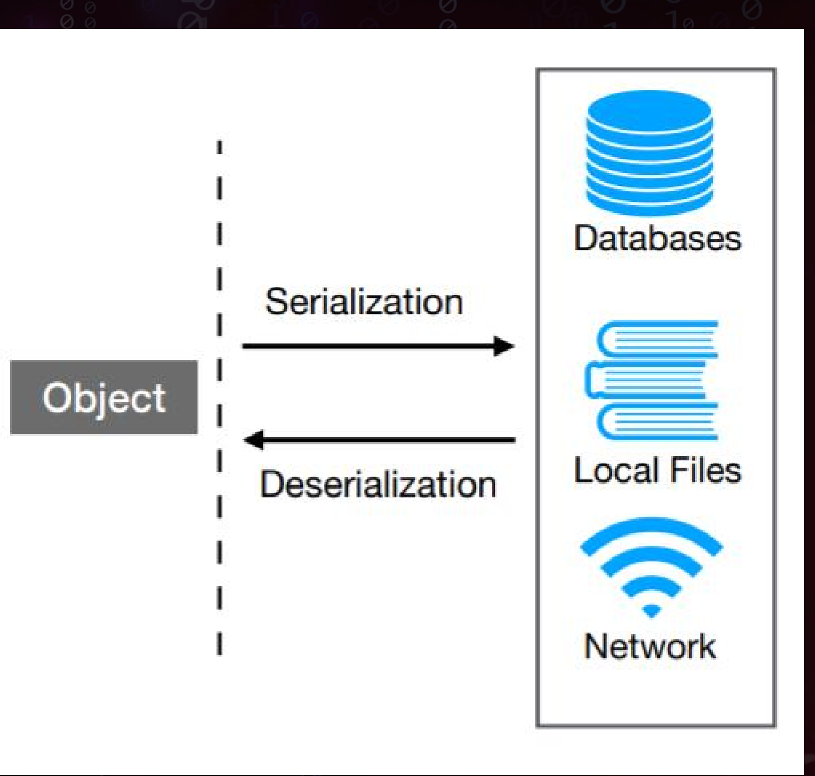
反序列化

将存储于内存或磁盘中的Java字节码还原为对象的过程

03
Kcon 2024

用途

远程网络对象传输
远程方法调用
不同环境间对象存储



Java反序列化简介

MANT EUM E

ObjectInputStream.readObject
readObject0
TC_OBJECT
是
readOrdinaryObject
ObjectStreamClass desc = readClassDesc(false)
resolveProxyClass（限制接口方法）
readNonProxyDesc->resolveClass
desc.isExternalizable()
否
readSerialData
是
readExternalData
obj.readExternal(ObjectInput in)
hasData（类描述符）
否
readObjectNoDataMethod
是
desc.hasReadObjectMethod
否
defaultReadFields
是
readObjectMethod
读取类中的信息并保存初始化

writeObjectMethod = getPrivateMethod(cl, "writeObject", new Class<?>[] { ObjectOutputStream.class }, Void.TYPE);

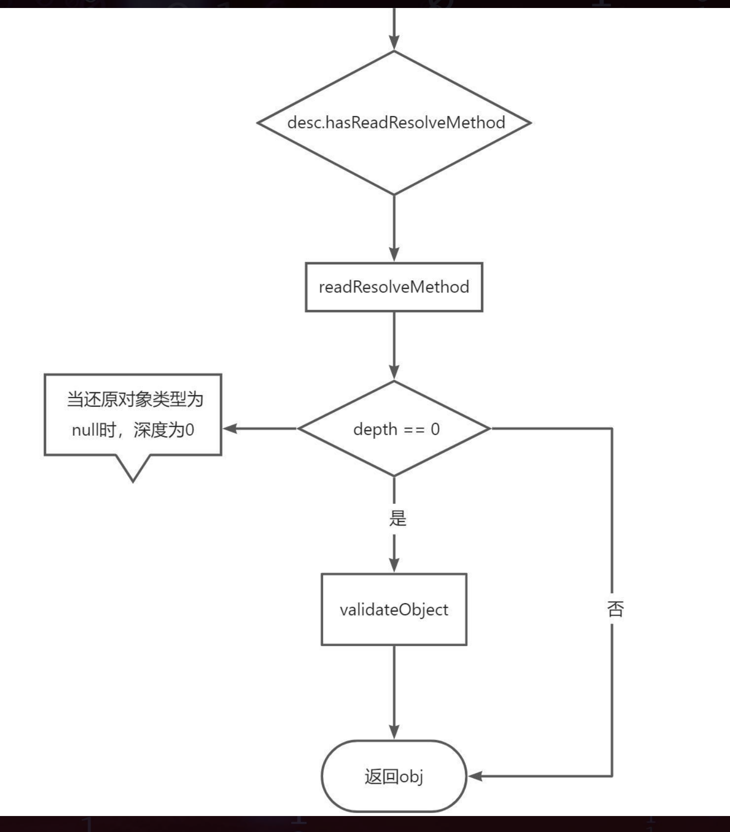
readObjectMethod = getPrivateMethod(cl, "readObject", new Class<?>[] { ObjectInputStream.class }, Void.TYPE);

readObjectNoDataMethod = getPrivateMethod(cl, "readObjectNoData", null, Void.TYPE);

writeReplaceMethod = getInheritableMethod(cl, "writeReplace", null, Object.class);

readResolveMethod = getInheritableMethod(cl, "readResolve", null, Object.class);

Serializable.class.isAssignableFrom(cl)
!Externalizable.class.isAssignableFrom(cl)
!Proxy.isProxyClass(cl)
!cl.isInterface()
!Modifier.STATIC | Modifier.TRANSIENT
cl.getDeclaredField("serialPersistentFields")



反序列化流程图
基于ObjectInputStream流，进行对象恢复

Java反序列化简介

MANT EUM E

KCon

远程传输任意字节码 还原任意对象

反序列化过程中 存在恶意方法

环境中存在薄弱可利用库

json转换Obj

xml转换Obj

自实现对象转换

攻击点

薄弱攻击点

魔术方法

void readObject (ObjectInputStream.class)

void readObjectNoData()

Object writeReplace()

void validateObject()

Object readResolve()

void writeObject (ObjectOutputStream.class)

void readExternal (ObjectInput.class)

Java反序列化简介

MANT EUM E

```java
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    OutputStream output = this.getOutputStream();
    if (this.cachedContent ≠ null) {
        output.write(this.cachedContent);
```

```java
public OutputStream getOutputStream() throws IOException {
    if (this.dfos == null) {
        File outputFile = this.getTempFile();
        this.dfos = new DeferredFileOutputStream(this.sizeThreshold, outputFile);
    }

    return this.dfos;
}

}
```

```java
private transient DeferredFileOutputStream dfos;
private transient File tempFile;
```

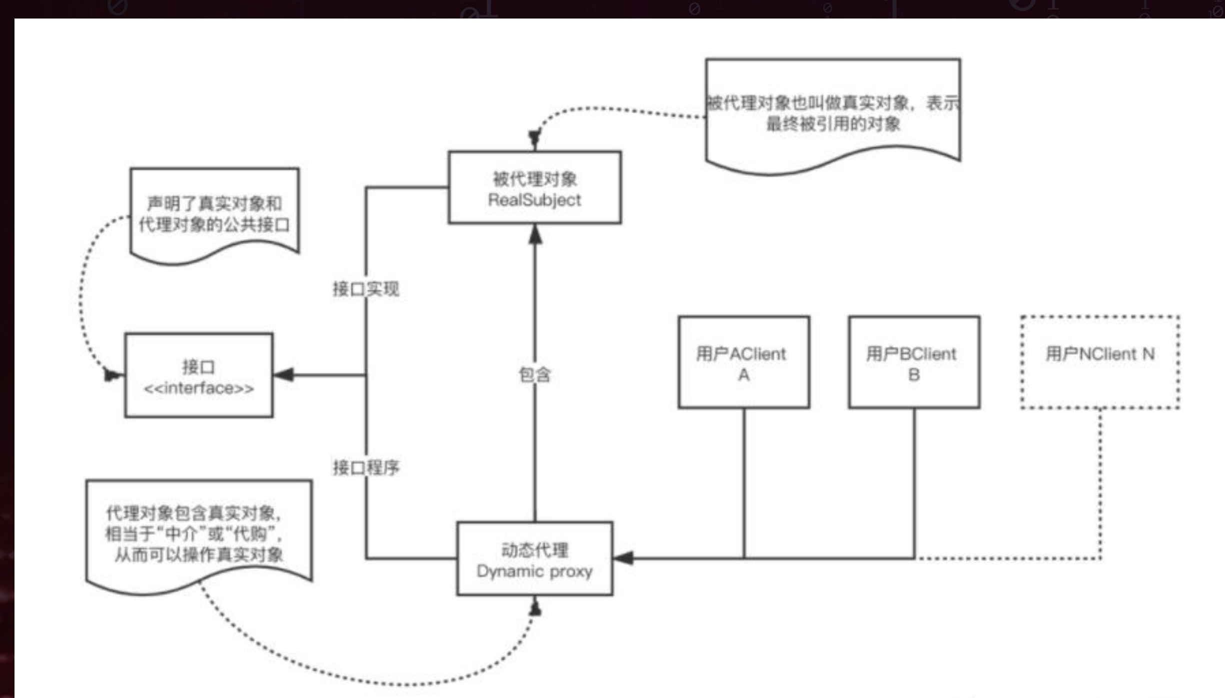
getOutputStream获取输出流并通过write或copy写入

其中有个知识点：
transient修饰的变量无法序列化和反序列化

Java代理模式应用

动态代理方式：
将接口方法 “ 代理 ” 给 InvocationHandler 接口
通过 Proxy 类创建代理对象

在反序列化漏洞中代表的jdk7u21就是采用了动态代理的方式，利用hash碰撞调用equals方法，进一步通过AnnotationInvocationHandler进行反射调用




```
public static boolean isProxyClass(Class<?> cl)
```

```
InvocationHandler getInvocationHandler(Object proxy)
```

```
Class<?> getProxyClass(ClassLoader loader,
                       Class<?>... interfaces)
```

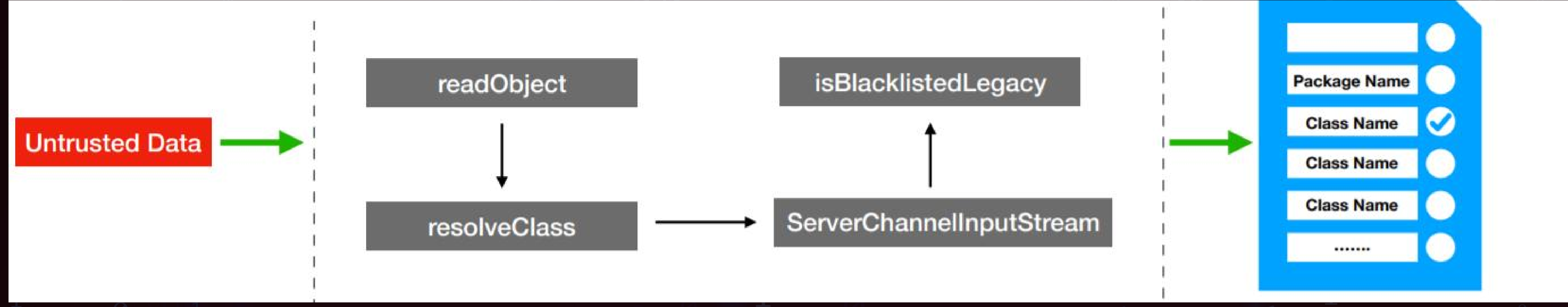
```
Object newProxyInstance(ClassLoader loader,
                        Class<?>[] interfaces,
                        InvocationHandler h)
```

KCon
PART ONE
02
常见防御思路与思考
MANDAMUS MEDIOCREM REREHENDUNT

常见防御思路与思考

MANT EUM E

继承ObjectInputStream并重写resolveClass方法，添加过滤方法
在可序列化存在可利用的类中，添加readResolve方法，过滤变量某些属性
高版本中利用JEP290机制继承ObjectInputFilter类，添加过滤方法




```
766 #
767 # Serialization process-wide filter
768 #
769 # A filter, if configured, is used by java.io.ObjectInputStream during
770 # deserialization to check the contents of the stream.
771 # A filter is configured as a sequence of patterns, each pattern is either
772 # matched against the name of a class in the stream or defines a limit.
773 # Patterns are separated by ";" (semicolon).
774 # Whitespace is significant and is considered part of the pattern.
775 #
776 # If the pattern ends with ".**" it matches any class in the package and all subpackages.
777 # If the pattern ends with ".*" it matches any class in the package.
778 # If the pattern ends with "*", it matches any class with the pattern as a prefix.
779 # If the pattern is equal to the class name, it matches.
780 # Otherwise, the status is UNDECIDED.
781 #
782 jdk.serialFilter=pattern;pattern
783 jdk.serialFilter=!sun.rmi.server.**;!org.codehaus.groovy.runtime.**
```

jre/lib/security/java.security

```
jdk.serialFilter=pattern;pattern
jdk.serialFilter=!sun.rmi.server.**;
jdk.serialFilter=!org.codehaus.groovy.runtime.**;
jdk.serialFilter=org.apache.commons.beanutils.BeanComparator
jdk.serialFilter=!org.codehaus.groovy.runtime.MethodClosure
```

常见绕过方式

基于继承ObjectInputStream类设置过滤方法
- 寻找新的调用链，绕过现有黑名单机制
- 寻找二次反序列化点，在低版本或未设置jep290的情况下，绕过过滤方法
- 寻找其他可替代流去反序列化对象，绕过过滤方法

序列化类中存在readResolve方法，过滤变量某些属性
- 寻找其他可操作的参数，在反序列化时，触发某些敏感操作

KCon
2024

继承ObjectInputFilter类设置过滤方法
- 低版本下，过滤方法失效
- 高版本下，寻找可利用替换流反序列化对象，绕过过滤方法

KCon
PART ONE
03
以点带面深入分析
MANDAMUS MEDIOCREM REREHENDUNT

安全与性能-Coherence

Oracle coherence是行业先进的内存中数据网格解决方案，通过快速访问常用数据，让企业能够以可预测的方式扩展任务关键型应用。通过实现自定义序列化机制，对序列化与反序列化数据进行了更细致的定制化处理，继承Serializable类，实现新的ExternalizableLite接口类并增加对原始DataInput和DataOutput流的操作方法，实现了高效的处理应用程序数据。

同时为了方便处理实现不同接口的类，官方还提供了对应接口助手类ExternalizableHelper，根据实现的不同接口调用不同方法进行反序列化处理

```
public static int getStreamFormat(Object o) {
    return o == null ? 0 : (o instanceof String ? 6 : (o instanceof Number ? (o instanceof Integer ? 1 : (o instanceof Long ? 2 : (o instanceof Double ? 3 : (o instanceof BigInteger ? 4 : (o instanceof BigDecimal ? 5 : (o instanceof Float ? 14 : (o instanceof Short ? 15 : (o instanceof Byte ? 16 : 11)))))))) : (o instanceof byte[] ? 8 : (o instanceof ReadBuffer ? 7 : (o instanceof XmlBean ? 12 : (o instanceof ExternalizableHelper.IntDecoratedObject ? 13 : (o instanceof ExternalizableLite ? 10 : (o instanceof Boolean ? 17 : (o instanceof Serializable ? 11 : (o instanceof Optional ? 22 : (o instanceof OptionalInt ? 23 : (o instanceof OptionalLong ? 24 : (o instanceof OptionalDouble ? 25 : (o instanceof XmlSerializable ? 9 : 255))))))))))))));
}
```

获取类属性并对其赋值

以点带面深入分析

MANT EUM E

深入思考实现

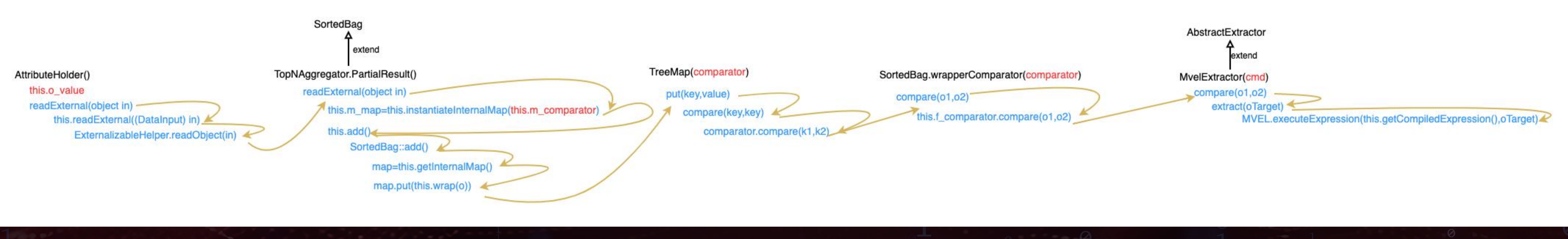
coherence在反序列化中实现新的处理方式：
通过直接调用DataInput反序列化readUTF读取类名，loadClass直接加载类的方式，代替了传统ObjectStreamClass.getName()读取类名的方式，进而绕过了在ObjectInputStream侧设置的ObjectInputFilter以及resolveClass的防御措施
再循环调用readExternal方法，进一步反序列化对应参数

```java
public static ExternalizableLite readExternalizableLite(DataInput in, ClassLoader loader) throws IOException {
  ExternalizableLite value;
  if (in instanceof PofInputStream) {
    value = (ExternalizableLite)((PofInputStream)in).readObject();
  } else {
    String sClass = readUTF(in);
    try {
      value = loadClass(sClass, loader, (ClassLoader)null).newInstance();
    } catch (InstantiationException e) {
      throw new IOException("Unable to instantiate an instance of class '" + sClass + "'; this is most likely due to a missing public " + "
    } catch (Exception e) {
      throw new IOException("Class initialization failed: " + e + "\n" + getStackTrace(e) + "\nClass: " + sClass + "\nClassLoader: " + loade
    }
    value.readExternal(in);
  }
  return value;
}
```

调用方式：寻找可以将ObjectInputStream流替换为DataInput流的魔术方法

CVE-2020-14756

- 利用条件需要继承ExternalizableLite类并使用了接口助手类ExternalizableHelper进行反序列化操作
- 利用AttributeHolader作为调用方式，将流进行转换
- 寻找TopNAggregator.PartialResult进行转接，调用Map的put处理，触发compare键值操作
- 最后直接利用之前compare->extract的调用链，完成对黑名单机制的绕过



以点带面深入分析

MANT EUM E

```java
public static ExternalizableLite readExternalizableLite(DataInput in, ClassLoader loader) throws IOException {
  ExternalizableLite value;
  if (in instanceof PofInputStream) {
    value = (ExternalizableLite)((PofInputStream)in).readObject();
  } else {
    WrapperDataInputStream wrapperDataInputStream1;
    String sClass = readUTF(in);
    WrapperDataInputStream inWrapper = (in instanceof WrapperDataInputStream) ? (WrapperDataInputStream)in : null;
    try {
      Class<?> clz = loadClass(sClass, loader, (inWrapper == null) ? null : inWrapper
          .getClassLoader());
      if (in instanceof ObjectInputStream) {
        ObjectInputStream ois = (ObjectInputStream)in;
        if (!checkObjectInputFilter(clz, ois))
          throw new InvalidClassException("Deserialization of class " + sClass + " was rejected");
      }
      value = (ExternalizableLite)clz.newInstance();
    } catch (InstantiationException e) {
      throw new IOException("Unable to instantiate an instance of class '" + sClass + "'; this is most likely due to a mi

          getStackTrace(e) + "\nClass: " + sClass + "\nClassLoader: " + loader + "\nContextClassLoader: " +

          getContextClassLoader());
```

```java
protected static boolean checkObjectInputFilter(Class<?> clz, ObjectInputStream ois) {
  try {
    Object filter = (HANDLE_GET_FILTER == null) ? null : HANDLE_GET_FILTER.invoke(ois);
    if (filter == null)
      return true;
    DynamicFilterInfo dynamic = s_tloHandler.get();
    dynamic.setClass(clz);
    Object oFilterInfo = dynamic.getFilterInfo();
    Enum status = HANDLE_CHECKINPUT.invoke(filter, oFilterInfo);
    dynamic.setClass(null);
    return (status != null && !status.name().equals("REJECTED"));
  } catch (ClassNotFoundException|IllegalAccessException|java.lang.reflect.InvocationTargetException e) {
    err("Unable to invoke checkInput on objectInputFilter ");
    err(e);
  } catch (Throwable throwable) {}
  return false;
}
```

```java
try {
  Class<?> clzFilter = null;
  Class<? extends Enum> clzFilterStatus = null;
  String sFilterMethod = null;
  Method methodGet = null;
  if ((clzFilter = getClass("java.io.ObjectInputFilter")) != null) {
    clzFilterInfo = Class.forName("java.io.ObjectInputFilter$FilterInfo");
    clzFilterStatus = (Class)Class.forName("java.io.ObjectInputFilter$Status");
    sFilterMethod = "getObjectInputFilter";
  } else if ((clzFilter = getClass("sun.misc.ObjectInputFilter")) != null) {
    clzFilterInfo = Class.forName("sun.misc.ObjectInputFilter$FilterInfo");
    clzFilterStatus = (Class)Class.forName("sun.misc.ObjectInputFilter$Status");
    sFilterMethod = "getInternalObjectInputFilter";
  }
  if (sFilterMethod != null) {
    Class<ObjectInputStream> clzObjectInputStream = ObjectInputStream.class;
    methodGet = clzObjectInputStream.getDeclaredMethod(sFilterMethod, new Class[0]);
    methodGet.setAccessible(true);
    MethodType mtCheckInput = MethodType.methodType(clzFilterStatus, clzFilterInfo);
    MethodHandles.Lookup lookup = MethodHandles.lookup();
    handleGetFilter = lookup.unreflect(methodGet);
    handleCheckInput = lookup.findVirtual(clzFilter, "checkInput", mtCheckInput);
  }
} catch (ClassNotFoundException|NoSuchMethodException|SecurityException|IllegalAccessException e) {
  CacheFactory.log("ObjectInputFilter will not be honored due to: " + e
      .getMessage() + '\n' + Base.printStackTrace(e), 3);
}
s_clzFilterInfo = clzFilterInfo;
HANDLE_GET_FILTER = handleGetFilter;
HANDLE_CHECKINPUT = handleCheckInput;
s_astats = new Stats[6451];
s_mapSerializerByClassLoader = Collections.synchronizedMap(new WeakHashMap<>());
}
```

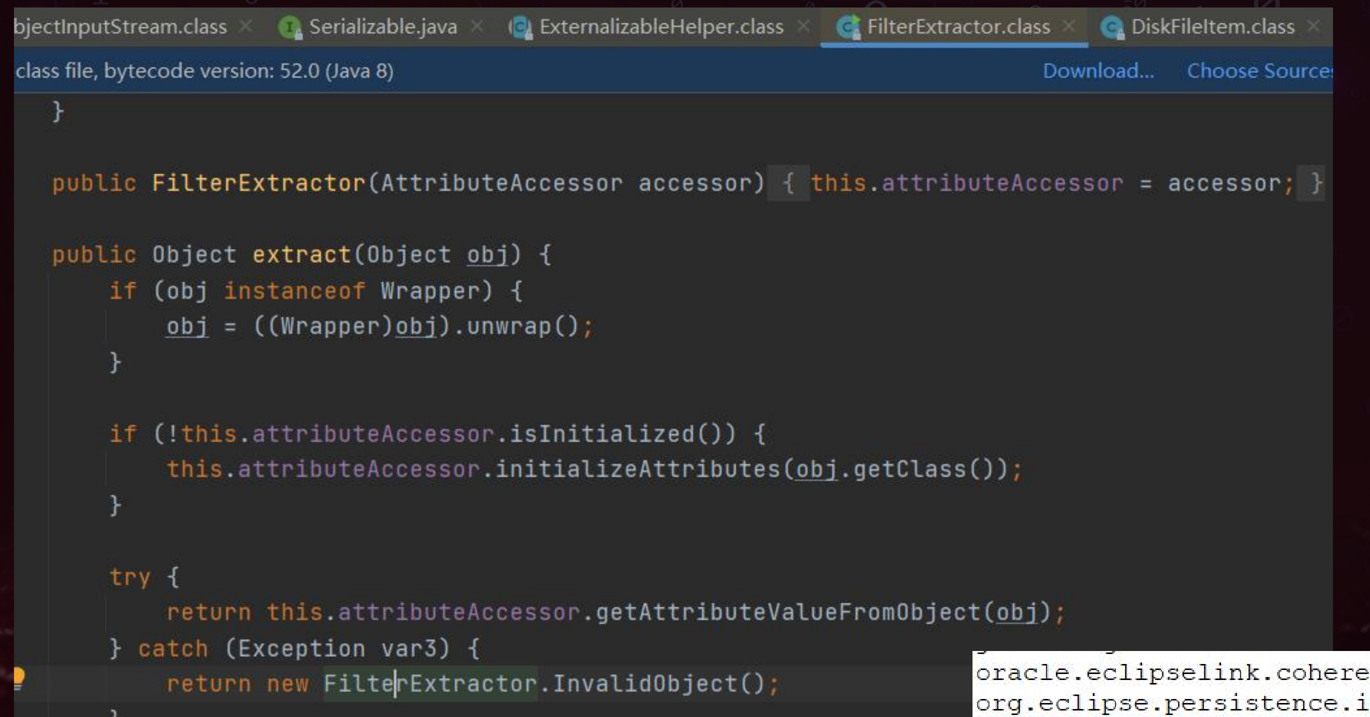
- ObjectInputStream流进行了二次处理，进一步通过黑名单过滤的恶意类。
- checkObjectInputFilter方法使用了JEP290特性
- 低版本下，此修复无效

以点带面深入分析

MANT EUM E

站在巨人的肩膀上思考问题，扩散思维

- 寻找其他存在的loadClass
- 寻找可控点存在的恶意行为
- 寻找extract方法中存在的可控点
- 寻找除loadClass加载类的其他方式



Object (java.lang)
AttributeAccessor (org.eclipse.persistence.mappings)
MultitenantPrimaryKeyAccessor (org.eclipse.persisten
OrmAttributeAccessor (org.eclipse.persistence.interna
JAXBArrayAttributeAccessor (org.eclipse.persistence.i
InstanceVariableAttributeAccessor (org.eclipse.persist
ValuesAccessor (org.eclipse.persistence.internal.dynan
IsSetNullPolicyIsSetParametersAttributeAccessor in Ob
IsSetNullPolicyIsSetParameterTypesAttributeAccessor
CustomAccessorAttributeAccessor (org.eclipse.persist
NullPolicyAttributeAccessor in ObjectPersistenceRunti
MapValueAttributeAccessor (org.eclipse.persistence.i
StoredFunctionResultAccessor in ObjectPersistenceRu
StoredProcedureArgumentsAccessor in ObjectPersiste
MethodAttributeAccessor (org.eclipse.persistence.int
SDOFragmentMappingAttributeAccessor (org.eclipse.

- 正常通过反序列化操作恢复变量，只能操
- 作图中的类，再通过筛选可序列化和
- 除黑名单外的类，发现并无可利用的点。

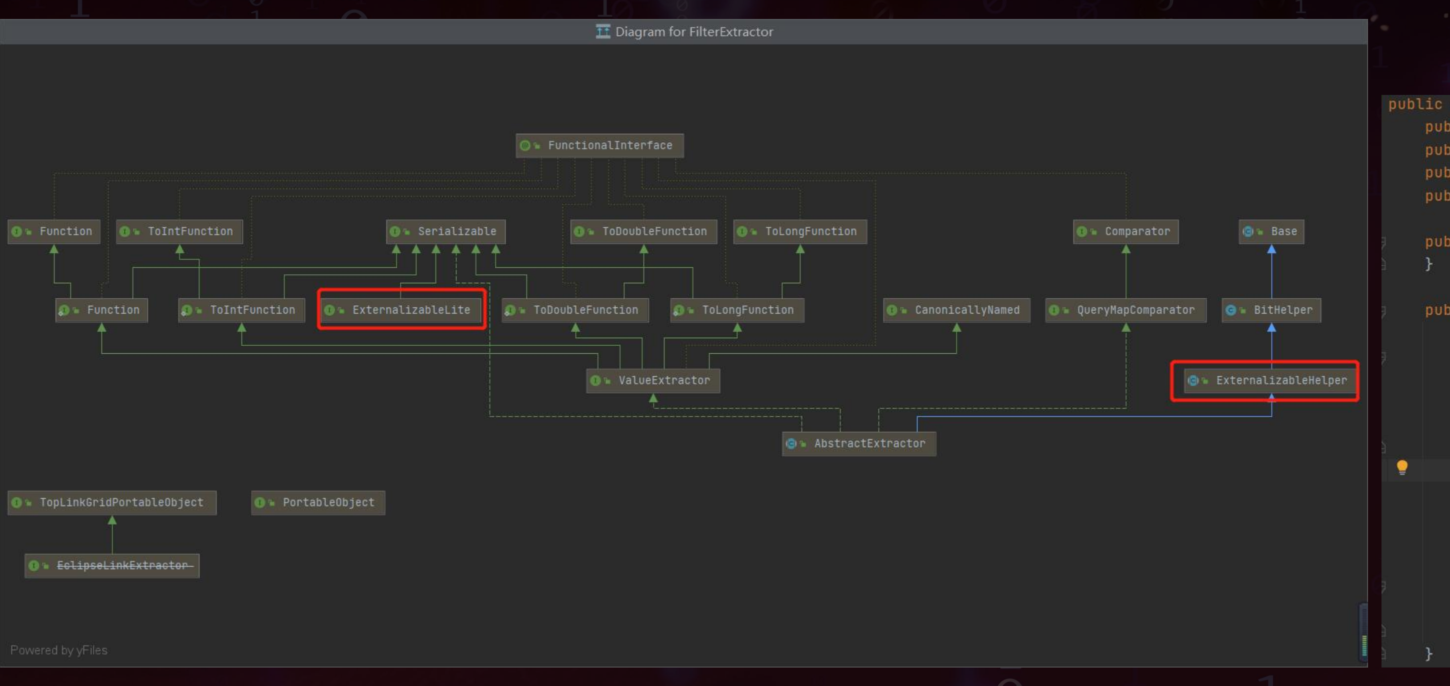
```
oracle.eclipselink.coherence.integrated.internal.cache.LockVersionExtractor
org.eclipse.persistence.internal.descriptors.MethodAttributeAccessor
org.eclipse.persistence.internal.descriptors.InstanceVariableAttributeAccessor
```

黑名单

以点带面深入分析

MANT EUM E

整体继承图，发现存在接口助手类并继承了ExternalizableLite类，观察对于DataInput流的自定义处理

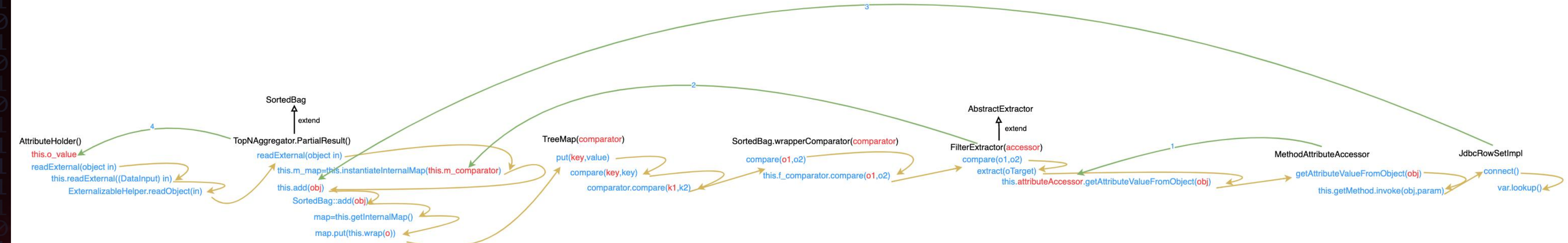



```java
public class SerializationHelper {
    public static final int OTHER = -1;
    public static final int INSTANCE_ACCESSOR = 0;
    public static final int METHOD_ACCESSOR = 1;
    public static final int ACCESSOR_ARRAY = 2;

    public SerializationHelper() {
    }

    public static AttributeAccessor readAttributeAccessor(DataInput in) throws IOException {
        int id = ExternalizableHelper.readInt(in);
        if (id == 0) {
            InstanceVariableAttributeAccessorExtended accessor = new InstanceVariableAttributeA
            accessor.setAttributeName((String)ExternalizableHelper.readObject(in));
            return accessor;
        } else if (id == 1) {
            MethodAttributeAccessor accessor = new MethodAttributeAccessor();
            accessor.setAttributeName((String)ExternalizableHelper.readObject(in));
            accessor.setGetMethodName((String)ExternalizableHelper.readObject(in));
            accessor.setSetMethodName((String)ExternalizableHelper.readObject(in));
            return accessor;
        } else {
            return null;
        }
    }
```

```java
public void readExternal(DataInput in) throws IOException {
    this.attributeAccessor = SerializationHelper.readAttributeAccessor(in);
}
```

CVE-2021-2394

- 利用AttributeHoladerr作为调用方式，将流进行转换
- 转换后的流执行自定义readExternal方法，通过SerializationHelper恢复变量，绕过黑名单
- 寻找TopNAggregator.PartialResult进行转接，调用Map的put处理，触发compare键值操作
- 利用之前compare->extract的调用链，调用任意代码执行



以点带面深入分析

MANT EUM E

高版本与ObjectInputFilter类设置黑名单双重防护

- 寻找新的利用链，难度较大
- 寻找新的转换流方法，转换其他新的流
- 寻找可替代ObjectInputStream的流，完成反序列化操作

```java
public static <T> T fromBinary(Binary bin) { return fromBinary(bin, ensureSerializer((ClassLoader)null)); }
```

ExternalizableHelper.fromBinary

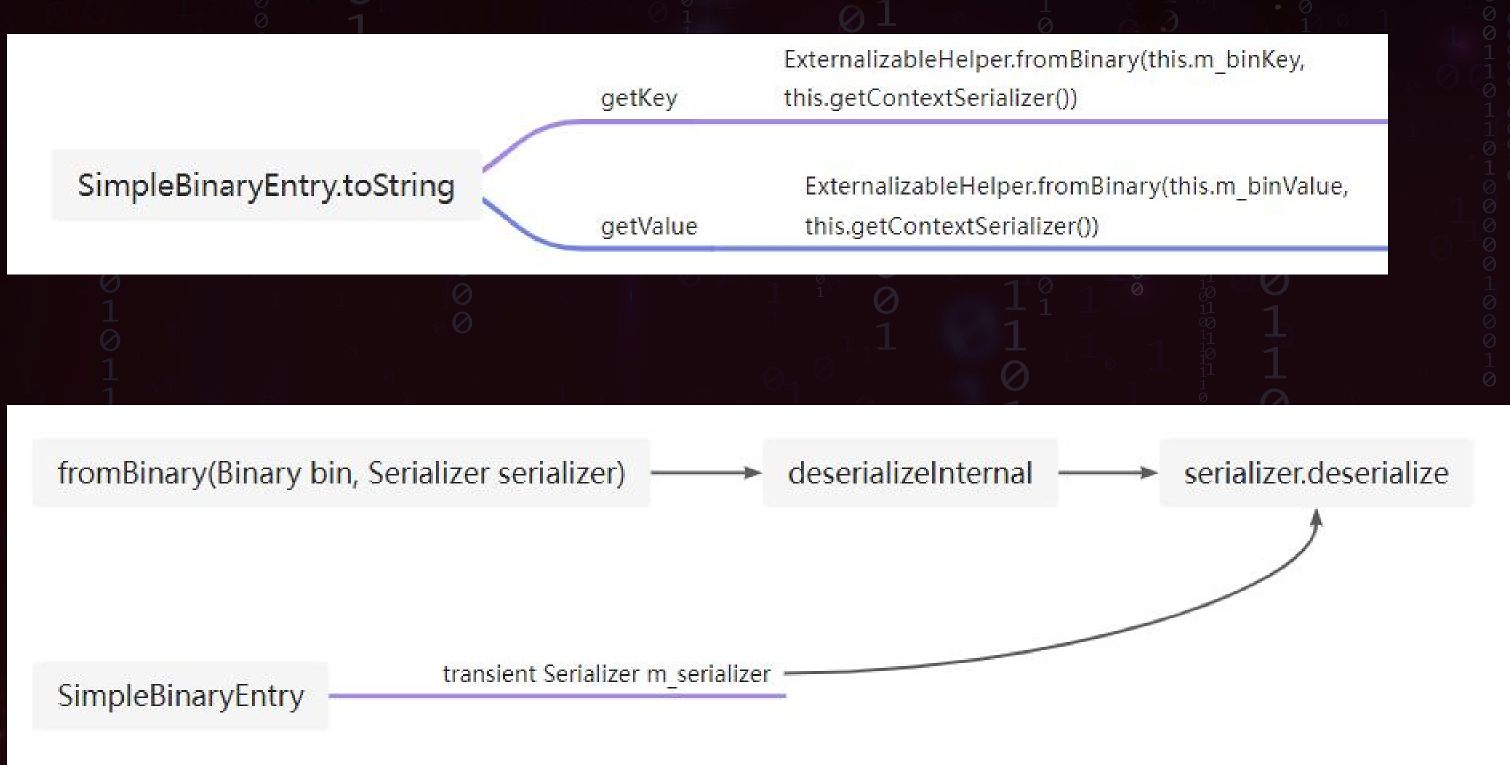
```java
private static <T> T deserializeInternal(Serializer serializer, ReadBuffer buf, Remote.Function<ReadBuffer.BufferInput, ReadBuffer.BufferIn
  long nMask;
  int cb;
  ReadBuffer.BufferInput in = buf.getBufferInput();
  int nType = in.readUnsignedByte();
  switch (nType) {
    case 13:
      readInt(in);
      nType = in.readUnsignedByte();
      break;
    case 18:
    case 19:
      nMask = (nType == 18) ? in.readByte() : in.readPackedLong();
      if ((nMask & 0x1L) == 0L)
        throw new EOFException("Decorated value is missing a value");
      cb = in.readPackedInt();
      in = buf.getReadBuffer(in.getOffset(), cb).getBufferInput();
      nType = in.readUnsignedByte();
      break;
  }
  if (supplierBufferIn != null)
    in = (ReadBuffer.BufferInput)supplierBufferIn.apply(in);
  Object o = (nType == 21) ? serializer.deserialize(in, clazz) : readObjectInternal((DataInput)in, nType, ((ClassLoaderAware)serializer)
      .getContextClassLoader());
  return realize(o, serializer);
}
```

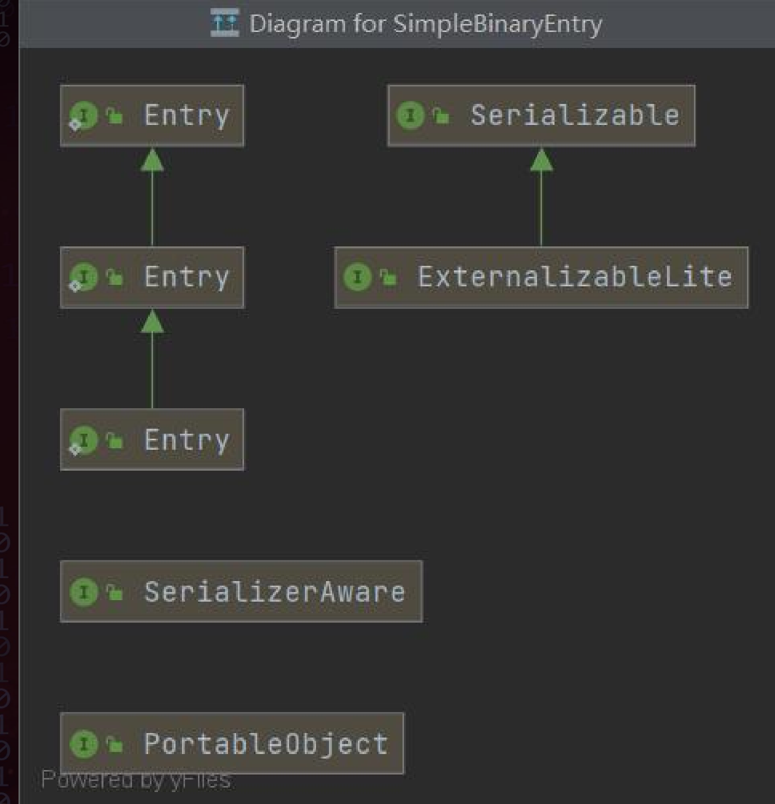
- 将Binary类转换为ReadBuffer，进一步从buffer中读取BufferInput缓冲区输入流进行反序列化操作。绕过了之前对ObjectInputStream流的黑名单过滤操作，可以接着调用loadClass方法还原变量

以点带面深入分析

MANT EUM E

全局搜索调用fromBinary方法的类，首先分析简单调用





以点带面深入分析

MANT EUM E

绕过transient修饰变量无法序列化

value instanceof SerializerAware

readExternalizableLite → setContextSerializer → ensureSerializer(ClassLoader) → ensureClassLoader(ClassLoader) → getContextClassLoader → Thread.currentThread().getContextClassLoade

loader==null

Base.class.getClassLoader() → ClassLoader.getSystemClassLoader()

s_mapSerializerByClassLoader.get(loader)

DefaultSerializer.setContextClassLoader(loader)

s_mapSerializerByClassLoader.put(loader,serializer)

关键节点：

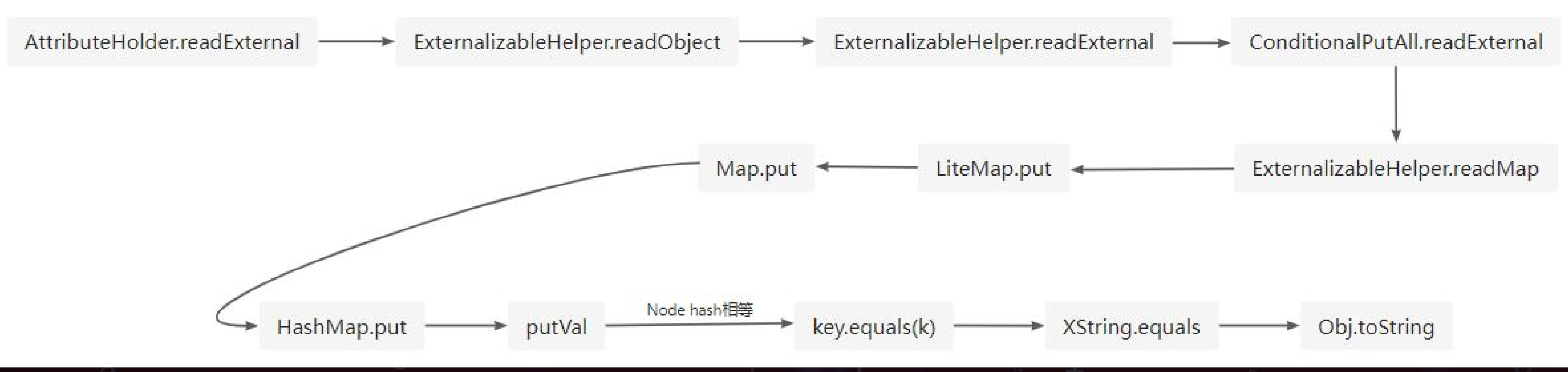
- 需要调用obj.toString方法
- 需要继承ExternalizableLite类
- 同时满足上述两点
- 需要依据readExternal方法进行拓展

突破点：

- 根据readExternal方法拓展，联想到Map等类型操作
- 根据equals，hashcode等方法进一步拓展，找到任意类调用toString
- source AttributeHolader转换流存在上述操作

以点带面深入分析

MANT EUM E




```java
public static int readMap(DataInput in, Map<Object, Object> map, ClassLoader loader) throws IOException {
  int cEntries;
  if (in instanceof PofInputStream) {
    PofInputStream inPof = (PofInputStream)in;
    inPof.getPofReader().readMap(inPof.nextIndex(), map);
    cEntries = map.size();
  } else {
    cEntries = in.readInt();
    for (int i = 0; i < cEntries; i++) {
      Object oKey = readObject(in, loader);
      Object oVal = readObject(in, loader);
      map.put(oKey, oVal);
    }
  }
  return cEntries;
}
```

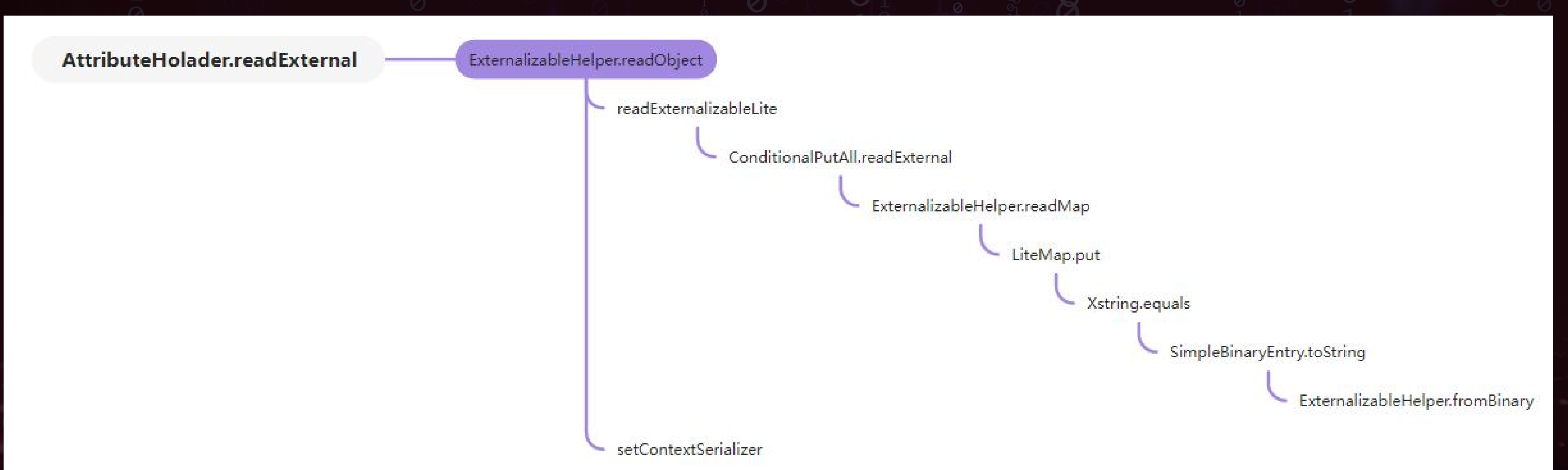
```java
public static HashMap makeMap ( Object v1, Object v2 ) throws Exception, ClassNotFoundException, NoSuch
        IllegalAccessException, InvocationTargetException {
    HashMap s = new HashMap();
    Reflections.setFieldValue(s, "size", 2);
    Class nodeC;
    try {
        nodeC = Class.forName("java.util.HashMap$Node");
    }
    catch ( ClassNotFoundException e ) {
        nodeC = Class.forName("java.util.HashMap$Entry");
    }
    Constructor nodeCons = nodeC.getDeclaredConstructor(int.class, Object.class, Object.class, nodeC);
    Reflections.setAccessible(nodeCons);

    Object tbl = Array.newInstance(nodeC, 2);
    Array.set(tbl, 0, nodeCons.newInstance(0, v1, v1, null));
    Array.set(tbl, 1, nodeCons.newInstance(0, v2, v2, null));
    Reflections.setFieldValue(s, "table", tbl);
    return s;
}
```

以点带面深入分析

MANT EUM E

CVE-2021-2394

- 利用AttributeHolader作为调用方式，将流进行转换
- 转换后的流执行满足条件的readExternal方法，调用Map相关操作
- 寻找ConditionalPutAll类作为调用满足条件的readExternal方法
- 利用HashMap对键的比较，调用Xstring.equals->SimpleBinaryEntry.toString
- 最后通过ExternalizableHelper.fromBinary处理新流，绕过基于ObjectInputStream流黑名单



以点带面深入分析

MANT EUM E

将之前的利用的类加入黑名单后如何绕过：

- 寻找可利用的魔术方法，重新实例化类
- 寻找Coherence在提高性能时，在序列化与反序列化操作存在的不同点
- 寻找其他类的可利用方法，调用到黑名单类，无需在反序列化时进行实例化操作

```java
public static XmlSerializable readXmlSerializable(DataInput in, ClassLoader loader) throws IOException {
  XmlSerializable value;
  if (in instanceof PofInputStream) {
    value = (XmlSerializable)((PofInputStream)in).readObject();
  } else {
    String sClass = readUTF(in);
    try {
      value = loadClass(sClass, loader, (ClassLoader)null).newInstance();
    } catch (Exception e) {
      throw new IOException("Class initialization failed: " + e + "\n" +

          getStackTrace(e) + "\nClass: " + sClass + "\nClassLoader: " + loader +

          getContextClassLoader());
    }
    String sXml = readUTF(in);
    XmlDocument xmlDocument = (new SimpleParser(false)).parseXml(sXml);
    value.fromXml((XmlElement)xmlDocument);
  }
  return value;
}
```

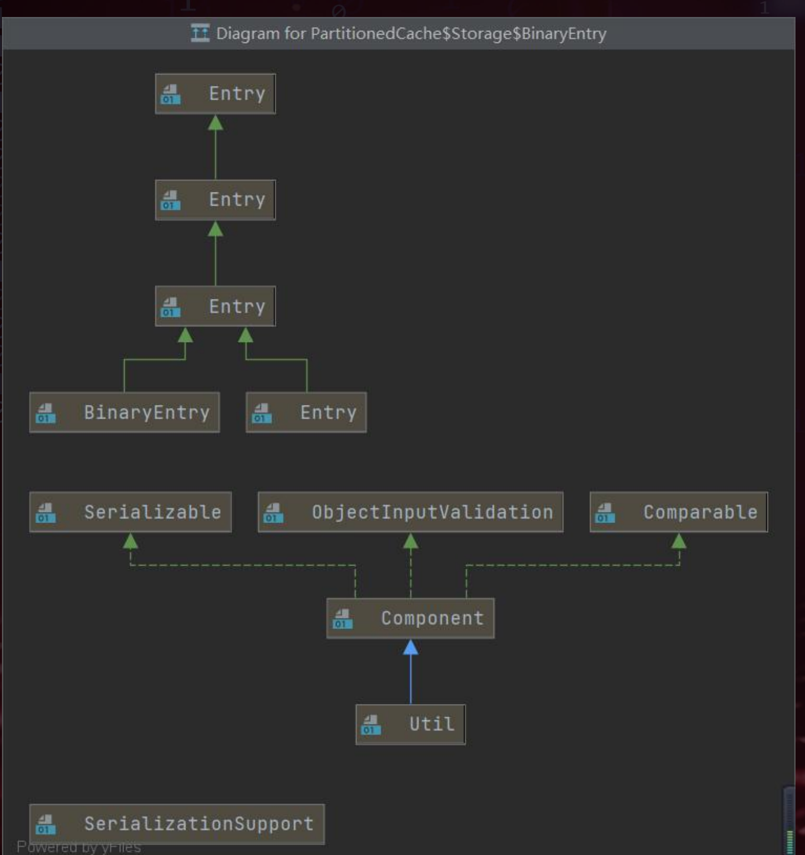
```java
public static Object replace(Object o) throws ObjectStreamException {
  if (Lambdas.isLambda(o))
    o = Lambdas.ensureRemotable((Serializable)o);
  if (o instanceof SerializationSupport)
    o = ((SerializationSupport)o).writeReplace();
  return o;
}

public static <T> T realize(Object o, Serializer serializer) throws ObjectStreamException {
  if (o instanceof SerializerAware)
    ((SerializerAware)o).setContextSerializer(serializer);
  if (o instanceof SerializationSupport) {
    o = ((SerializationSupport)o).readResolve();
    if (o instanceof SerializerAware)
      ((SerializerAware)o).setContextSerializer(serializer);
  }
  return (T)o;
}
```

以点带面深入分析

MANT EUM E

寻找满足以上条件的类：PartitionedCache$Storage$BinaryEntry

```java
public Object writeReplace() throws ObjectStreamException {
    return new SimpleBinaryEntry(this.getBinaryKey(), this.getBinaryValue());
}
```



- 继承序列化与SerializationSupport接口
- 满足在序列化时调用writeReplace方法的条件
- 寻找在反序列化时会触发序列化的操作

```java
public static void writeObject(DataOutput out, Object o) throws IOException {
    if (out instanceof PofOutputStream) {
        ((PofOutputStream)out).writeObject(o);
    } else {
        o = replace(o);
```

```java
private static WriteBuffer serializeInternal(Serializer serializer, Object o, boolean fBinary)
    boolean fDeco = false;
    int nDeco = 0;
    int cb = 1;
    boolean fUser = false;
    ExternalizableHelper.Stats stats = null;
    if (o instanceof ExternalizableHelper.IntDecoratedObject) {
        ExternalizableHelper.IntDecoratedObject ido = (ExternalizableHelper.IntDecoratedObject
        fDeco = true;
        nDeco = ido.getDecoration();
        cb += 5;
        o = ido.getValue();
    }

    o = replace(o);
```

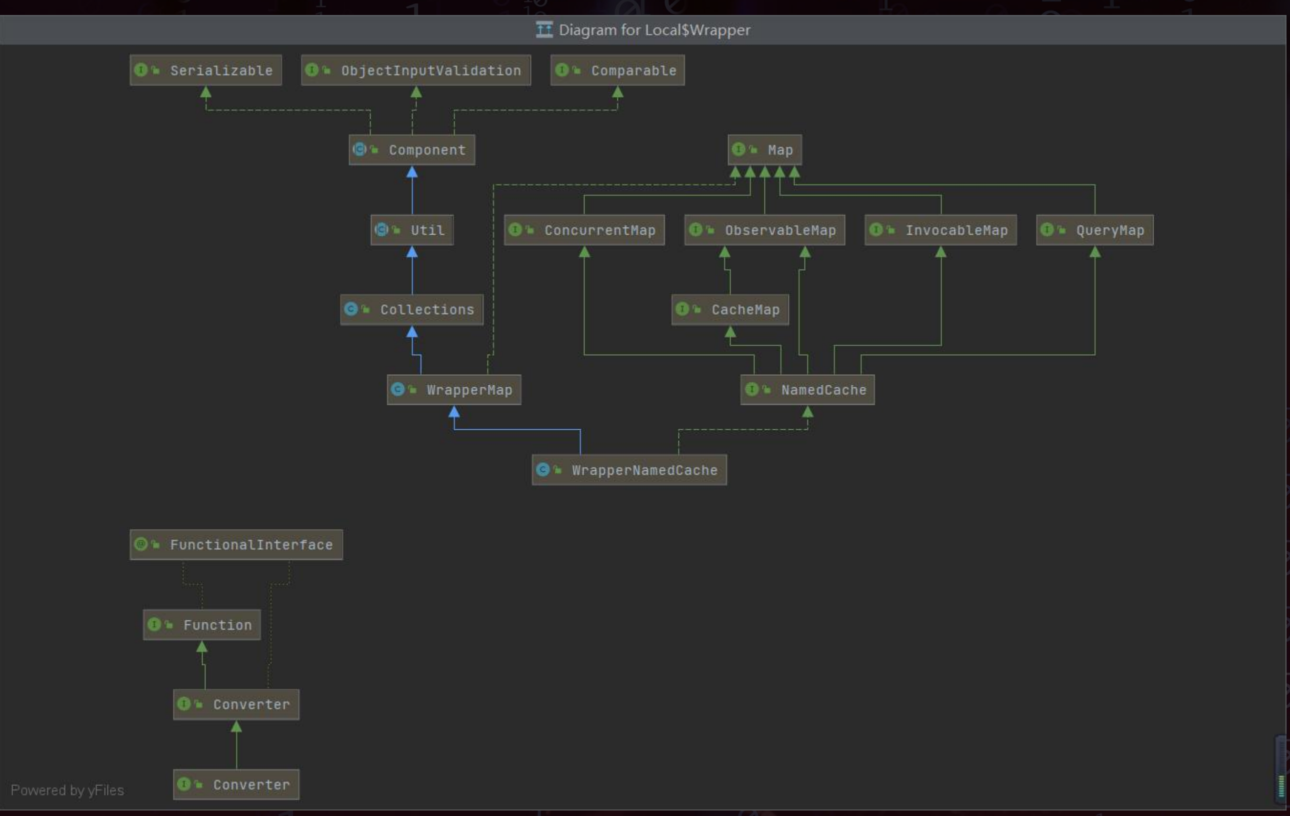
以点带面深入分析

MANT EUM E

向上链接Local$Wrapper满足条件

```java
public static Binary toBinary(Object o, Serializer serializer) {
    try {
        return serializeInternal(serializer, o,  fBinary: true).toBinary();
    } catch (IOException var3) {
        throw new WrapperException(var3);
    }
}
```

```java
public Object convert(Object o) { return this.ensureClone(o); }

public Object ensureClone(Object o) {
    boolean fImmutable = false;
    if (o instanceof Number) {
        fImmutable = ((((((o instanceof Byte ? true : o instanceof Short) ? true : o instance
    } else {
        fImmutable = ((o instanceof String ? true : o instanceof Binary) ? true : o instanceo
    }

    if (fImmutable ? true : o == null) {
        return o;
    } else {
        if (o instanceof Cloneable) {
            try {
                return ClassHelper.invoke(o,  sName: "clone", ClassHelper.VOID);
            } catch (Throwable var6) {
            }
        }

        Serializer serializer;
        try {
            serializer = ((Local)this.get_Module()).getNamedCache().getCacheService().getSeri
        } catch (RuntimeException var5) {
            serializer = ExternalizableHelper.ensureSerializer(o.getClass().getClassLoader())
        }

        return ExternalizableHelper.fromBinary(ExternalizableHelper.toBinary(o, serializer),
}
```



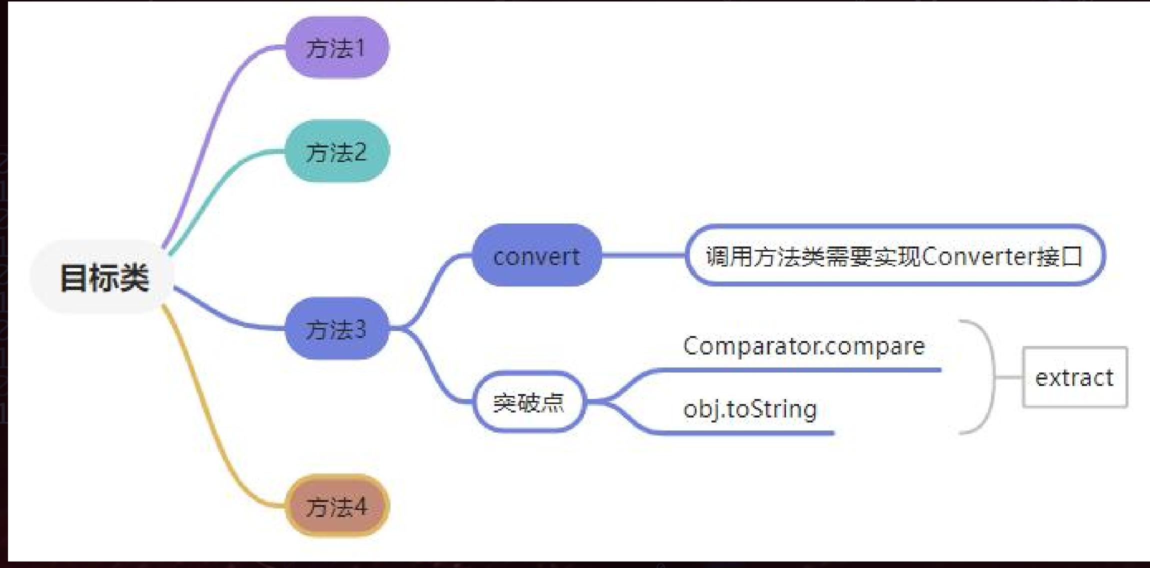
根据接口类进一步挖掘相关调用实现类

以点带面深入分析

MANT EUM E

整理目前得到的信息以及接下来需要寻找的方向：

- 目标类方法内部需要调用到convert方法
- 调用到convert方法的类需要实现Converter接口
- 目标类需要符合可序列化要求
- 简化调用链，最好可以利用到之前的突破点



- 根据整理信息找到ConverterCollections.ConverterComparator满足以上条件

```java
public static class ConverterComparator<F, T> implements Comparator<T>, Serializable {
  private final Converter<T, F> m_conv;

  private final Comparator<? super F> m_comparator;

  public ConverterComparator(Comparator<? super F> comparator, Converter<T, F> conv) {
    this.m_comparator = comparator;
    this.m_conv = conv;
  }

  public int compare(T o1, T o2) {
    Converter<T, F> conv = this.m_conv;
    return this.m_comparator.compare((F)conv.convert(o1), (F)conv.convert(o2));
  }
}
```

这里调用PriorityQueue类作为起始类，调用compare方法

至此调用链串通，由于在序列化时会调用writeReplace方法，需要利用javassist重写方法或直接覆盖类

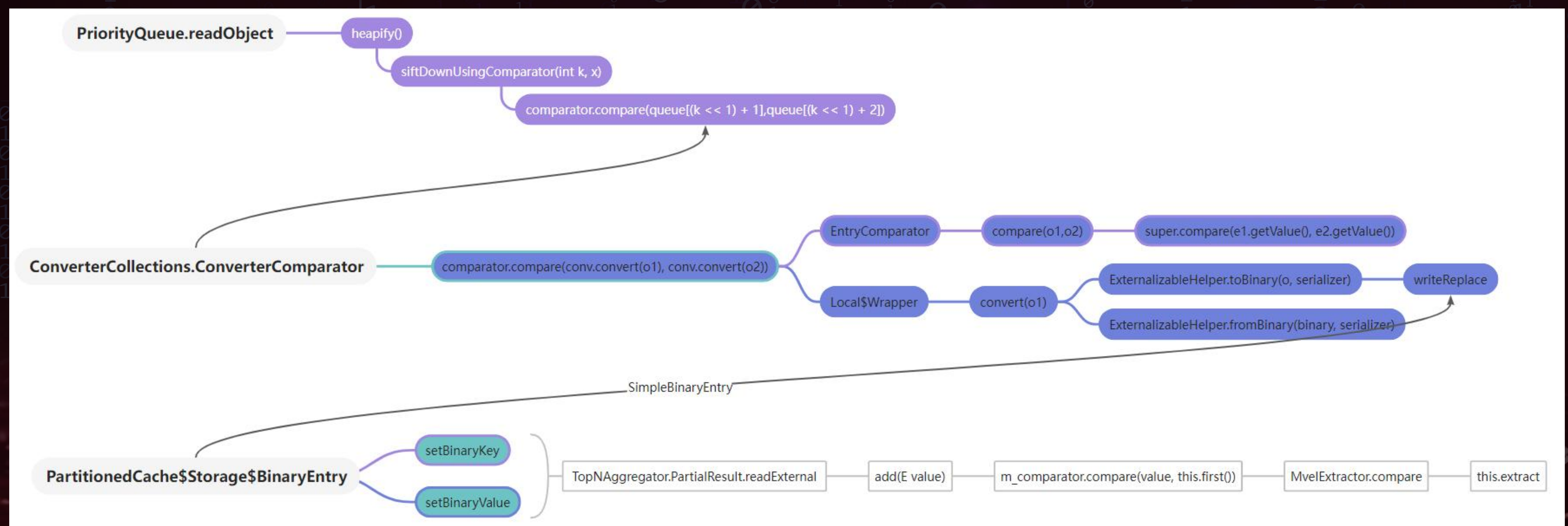
以点带面深入分析

MANT EUM E

KCon

CVE-2021-35617

- 利用PartitionedCache$Storage$BinaryEntry替换黑名单类SimpleBinaryEntry
- Local$Wrapper类承载序列化后的Binary数据
- 调用ConverterCollections.ConverterComparator串联Converter.convert与compare方法
- 利用之前compare->extract的调用链，调用任意代码执行



以点带面深入分析

MANT EUM E

同样在换汤不换药的修复方式下，思考如何在之前的利用的类加入黑名单后如何绕过：

- 通过魔术方法重新生成黑名单中的可利用类
- 已知Coherence在反序列化中有新的处理方式，是否在对原生数据处理上有所不同可利用
- 寻找除黑名单之外的其他利用类

```java
public static Object readSerializable(DataInput in, ClassLoader loader) throws IOException {
  Object o;
  if (in instanceof PofInputStream) {
    o = ((PofInputStream)in).readObject();
  } else {
    ObjectInput streamObj = getObjectInput(in, loader);
    try {
      o = streamObj.readObject();
      if (o instanceof SerializerAware)
        ((SerializerAware)o).setContextSerializer(ensureSerializer(loader));
    } catch (IOException e) {
      throw e;
    } catch (Exception e) {
      throw new IOException("readObject failed: " + e + "\n" +

          getStackTrace(e) + "\nClassLoader: " + loader);
    }
  }
  return o;
}
```

```java
public static class DefaultObjectStreamFactory implements ObjectStreamFactory {
  public ObjectInput getObjectInput(DataInput in, ClassLoader loader, boolean fForceNew) thr
    if (!fForceNew && in instanceof ObjectInput && (!ExternalizableHelper.FORCE_RESOLVING_ST
      return (ObjectInput)in;
    InputStream stream = ExternalizableHelper.getInputStream(in);
    loader = Base.ensureClassLoader((loader == null && in instanceof WrapperDataInputStream)
        .getClassLoader() : loader);
    return (ObjectInput)new ResolvingObjectInputStream(stream, loader);
  }
```

```java
factory = AccessController.<ObjectStreamFactory>doPrivileged(new PrivilegedAction<ObjectStreamFactory>() {
      public ObjectStreamFactory run() {
        try {
          if (!XmlHelper.isInstanceConfigEmpty(xmlFactory))
            return (ObjectStreamFactory)XmlHelper.createInstance(xmlFactory, ExternalizableHelper.class
                .getClassLoader(), null);
        } catch (Exception e) {
          Base.err("Unable to instantiate an ObjectStreamFactory \"" + xmlFactory + "\":");
          Base.err(e);
        }
        return new ExternalizableHelper.DefaultObjectStreamFactory();
      }
    });
```

通过分析，如果类继承了原生的Serializable接口，那么依旧会经过ObjectFilter的过滤，无法绕过黑名单

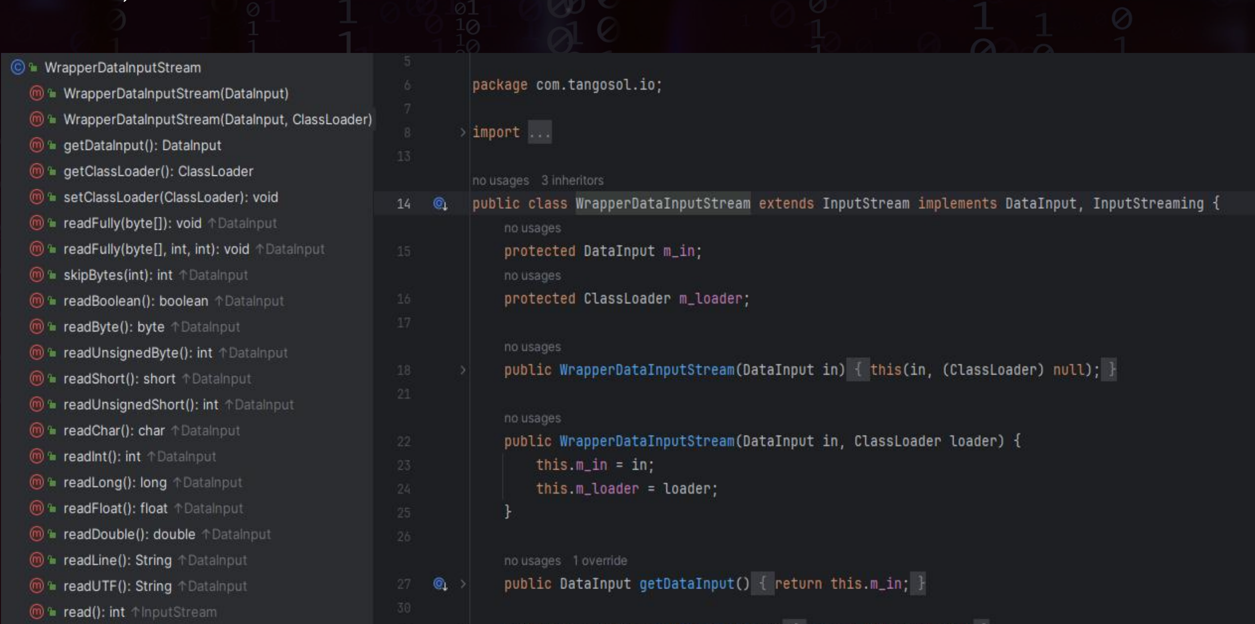
```java
public class ResolvingObjectInputStream extends ObjectInputStream implements Resolving {
  private ClassLoader m_loader;

  public ResolvingObjectInputStream(InputStream stream, ClassLoader loader) throws IOException {
    super(stream);
    this.m_loader = loader;
  }

  protected Class resolveClass(ObjectStreamClass descriptor) throws IOException, ClassNotFoundException {
    ClassLoader loader = this.m_loader;
    if (loader != null)
      try {
        return Class.forName(descriptor.getName(), false, loader);
      } catch (ClassNotFoundException classNotFoundException) {}
    return super.resolveClass(descriptor);
  }
}
```

以点带面深入分析

MANT EUM E

如果在原生数据的处理上无差别的情况下，经过CVE-2020-1475补丁的修复，是否还有利用可能

```java
public static ExternalizableLite readExternalizableLite(DataInput in, ClassLoader loader) throws IOException {
  ExternalizableLite value;
  if (in instanceof PofInputStream) {
    value = (ExternalizableLite)((PofInputStream)in).readObject();
  } else {
    WrapperDataInputStream wrapperDataInputStream1;
    String sClass = readUTF(in);
    WrapperDataInputStream inWrapper = (in instanceof WrapperDataInputStream) ? (WrapperDataInputStream)in : null;
    try {
      Class<?> clz = loadClass(sClass, loader, (inWrapper == null) ? null : inWrapper
          .getClassLoader());
      if (in instanceof ObjectInputStream) {
        ObjectInputStream ois = (ObjectInputStream)in;
        if (!checkObjectInputFilter(clz, ois))
          throw new InvalidClassException("Deserialization of class " + sClass + " was rejected");
      }
      value = (ExternalizableLite)clz.newInstance();
    } catch (InstantiationException e) {
      throw new IOException("Unable to instantiate an instance of class '" + sClass + "'; this is most likely due to a missing public no-ar

          getStackTrace(e) + "\nClass: " + sClass + "\nClassLoader: " + loader + "\nContextClassLoader: " +

          getContextClassLoader());
    } catch (Exception e) {
      throw new IOException("Class initialization failed: " + e + "\n" +

          getStackTrace(e) + "\nClass: " + sClass + "\nClassLoader: " + loader + "\nContextClassLoader: " +

          getContextClassLoader(), e);
    }
    if (loader != null)
      if (inWrapper == null) {
        wrapperDataInputStream1 = new WrapperDataInputStream(in, loader);
      } else if (loader != inWrapper.getClassLoader()) {
        inWrapper.setClassLoader(loader);
      }
    value.readExternal((DataInput)wrapperDataInputStream1);
    if (value instanceof SerializerAware)
      ((SerializerAware)value).setContextSerializer(ensureSerializer(loader));
  }
  return value;
}
```



思考

- 已知修复方式也是基于ObjectFilter进行过滤，是否存在差异化
- 对于不同的数据流，如何处理
- WrapperDataInputStream是否也存在黑名单的过滤

分析

- 在传入数据流不是基于ObjectInputStream时，不会进入黑名单过滤
- WrapperDataInputStream是Coherence新的处理数据流类，不存在黑名单过滤
- 寻找是否存在数据流转换，不基于ObjectInputStream实现了对象的还原

以点带面深入分析

MANT EUM E

经过前面的分析，对于转换流马上可以联想到CVE-2021-35617的处理方式，但是入口类被加入黑名单的情况下，如何再次利用转换

```java
public class PartitionedCache$Storage$EntryToBinaryEntryConverter extends Util implements Converter {
  public PartitionedCache$Storage$EntryToBinaryEntryConverter() {
    super(null, null, true);
  }

  public PartitionedCache$Storage$EntryToBinaryEntryConverter(Object sName, Object compParent, boolean fInit) {
    super((String)sName, (Component)compParent, false);
    if (fInit)
      super.__init();
  }
```

PartitionedCache$Storage.class

```java
public PartitionedCache$Storage$BinaryEntry instantiateBinaryEntry(Binary binKey, Binary binValue, boolean fReadOnly) {
  entry = new PartitionedCache$Storage$BinaryEntry();
  entry.setBinaryKey(binKey);
  entry.setBinaryValue(binValue);
  if (fReadOnly)
    entry.ensureReadOnly();
  _linkChild((Component)entry);
  return entry;
}
```

```java
  protected void __initPrivate() {
    super.__initPrivate();
  }

  public Object convert(Object o) {
    entry = (Map.Entry)o;
    return ((PartitionedCache$Storage)get_Parent()).instantiateBinaryEntry((Binary)entry.getKey(), (Binary)entry.getValue(), true);
  }
```

寻找满足以上条件的类：PartitionedCache$Storage$EntryToBinaryEntryConverter

- 继承了Converter类
- convert方法中实现了黑名单中的类

以点带面深入分析

MANT EUM E

已知在PartitionedCache$Storage$BinaryEntry的writeReplace方法中会还原SimpleBinaryEntry，目前调用链可以回溯到反序列化中调用用convert方法使PartitionedCache$Storage$EntryToBinaryEntryConverter生成PartitionedCache$Storage$BinaryEntry，接下来如何再次通过序列化重新生成SimpleBinaryEntry，需要进一步挖掘sink

```java
@Deprecated
public static class ConverterEntry extends ConverterCollections.ConverterEntry {
  public ConverterEntry(Map.Entry entry, Converter conKeyUp, Converter conValUp, Conv
    super(entry, conKeyUp, conValUp, conValDown);
  }
}
```

```java
public static class ConverterEntry<FK, TK, FV, TV> extends AbstractConverterEntry<FK, TK, FV, T
  protected final Converter<FK, TK> m_convKeyUp;

  protected final Converter<FV, TV> m_convValUp;

  protected final Converter<TV, FV> m_convValDown;

  public ConverterEntry(Map.Entry<FK, FV> entry, Converter<FK, TK> convKeyUp, Converter<FV, TV
    super(entry);
    this.m_convKeyUp = convKeyUp;
    this.m_convValUp = convValUp;
    this.m_convValDown = convValDown;
  }

  protected Converter<FK, TK> getConverterKeyUp() {
    return this.m_convKeyUp;
  }

  protected Converter<FV, TV> getConverterValueUp() {
    return this.m_convValUp;
  }

  protected Converter<TV, FV> getConverterValueDown() {
    return this.m_convValDown;
  }
}
```

```java
protected static abstract class AbstractConverterEntry<FK, TK, FV, TV> implements Map.Entry<TK, TV>, Serializable {
  protected final Map.Entry<FK, FV> m_entry;

  protected transient TK m_oKeyUp;

  protected transient TV m_oValueUp;

  protected AbstractConverterEntry(Map.Entry<FK, FV> entry) {
    this.m_entry = entry;
  }

  protected abstract Converter<FK, TK> getConverterKeyUp();

  protected abstract Converter<FV, TV> getConverterValueUp();

  protected abstract Converter<TV, FV> getConverterValueDown();

  public TK getKey() {
    TK oKeyUp = this.m_oKeyUp;
    if (oKeyUp == null)
      this
        .m_oKeyUp = oKeyUp = (TK)getConverterKeyUp().convert(getEntry().getKey());
    return oKeyUp;
  }

  public TV getValue() {
    TV oValueUp = this.m_oValueUp;
    if (oValueUp == null)
      this
        .m_oValueUp = oValueUp = (TV)getConverterValueUp().convert(getEntry().getValue());
    return oValueUp;
  }

  public TV setValue(TV value) {
    this.m_oValueUp = null;
    return (TV)getConverterValueUp().convert(
        getEntry().setValue((FV)getConverterValueDown().convert(value)));
  }
```

以点带面深入分析

MANT EUM E

CVE-2022-21420

- 利用PartitionedCache$Storage$EntryToBinaryEntryConverter的convert方法实现黑名单中的PartitionedCache$Storage$BinaryEntry类
- 调用ConverterCollections.ConverterEntry，通过嵌套的方式，将上述生成的类再利用Local$Wrapper类承载序列化后的Binary数据
- 利用Binary数据绕过ObjectFilter机制，再使用toString->getKey->extract的调用链，调用任意代码执行

KCon
PART ONE
04
代理模式下的利用
MANDAMUS MEDIOCREM REREHENDUNT

代理模式下的利用

MANT EUM E

Java代理模式，引入代理类控制其他对象的访问，主要用于控制对复杂对象的访问和操作，或在访问对象时添加额外的功能。代表有jdk的AnnotationInvocationHandler，著名的jdk7u21与jdk8u20的反序列化就诞生于此。以防促攻，挖掘补丁以及修复手段，从而发现新的利用方式

```java
private synchronized void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException {

    synchronized(BeanContext.globalHierarchyLock) {
        ois.defaultReadObject();

        initialize();

        bcsPreDeserializationHook(ois);

        if (serializable > 0 && this.equals(getBeanContextPeer()))
            readChildren(ois);

        deserialize(ois, bcmListeners = new ArrayList( initialCapacity: 1));
    }
}
```

```java
            var8 = var9.memberValues.get(var6);
        } else {
            try {
                var8 = var5.invoke(var1);
```

在jdk7u21时未对反序化的字段进行数据校验，所以在代理到equals方法时，可以反射执行恶意方法

在jdk8u20时通过抛出异常结束反序列化，绕过是因为找到一个类在反序化时捕捉了异常未抛出，导致反序列化继续进行，绕过了异常抛出的修复方式

```java
public final void readChildren(ObjectInputStream ois) throws IOException, ClassNotFoundException
{
    int count = serializable;

    while (count-- > 0) {
        Object                      child = null;
        BeanContextSupport.BCSChild bscc  = null;

        try {
            child = ois.readObject();
            bscc  = (BeanContextSupport.BCSChild)ois.readObject();
        } catch (IOException ioe) {
            continue;
        } catch (ClassNotFoundException cnfe) {
            continue;
        }
```

代理模式下的利用

MANT EUM E

Jdk8u20的修复方式，在获取对应class的方法时增加了限制。站在巨人的肩膀上，分析可以挖掘的点：

- 继承并实现Serializable与InvocationHandler接口
- 代理类实现功能时，存在对传入方法进行自定义的处理
- 没有对代理类的type或method进行限制

```java
class AnnotationProxy implements Annotation, InvocationHandler, Serializable {

    private static final long serialVersionUID = 6907601010599429454L;
    private static final Log log = LoggerFactory.make();

    private final Class<? extends Annotation> annotationType;
    private final Map<String, Object> values;
    private final int hashCode;

    AnnotationProxy(AnnotationDescriptor<?> descriptor) {
        this.annotationType = descriptor.type();
        values = Collections.unmodifiableMap( getAnnotationValues( descriptor ) );
        this.hashCode = calculateHashCode();
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        if ( values.containsKey( method.getName() ) ) {
            return values.get( method.getName() );
        }
        return method.invoke( this, args );
    }
```

```java
public boolean equals(Object obj) {
    if ( this == obj ) {
        return true;
    }
    if ( obj == null ) {
        return false;
    }
    if ( !annotationType.isInstance( obj ) ) {
        return false;
    }

    Annotation other = annotationType.cast( obj );

    //compare annotation member values
    for ( Entry<String, Object> member : values.entrySet() ) {

        Object value = member.getValue();
        Object otherValue = getAnnotationMemberValue( other, member.getKey() );

        if ( !areEqual( value, otherValue ) ) {
            return false;
        }
    }
```

```java
private Object getAnnotationMemberValue(Annotation annotation, String name) {
    try {
        return run( GetDeclaredMethod.action( annotation.annotationType(), name ) ).invoke( annotation );
    }
```

代理模式下的利用

MANT EUM E

CVE-2022-21306

- 利用AnnotationProxy生成代理类，携带调用方法名，通过equals方法进入invoke调用getAnnotationMemberValue
- 通过getAnnotationMemberValue转换对应method属性，通过二次代理AnnotationProxy，对应调用方法返回调用目标方法正常返回方法属性
- 将上述返回的方法通过ProxyIndirectionHandler进行链接，将返回的方法属性赋予需要需要后续调用的类，完成任意无参方法调用
- 寻找满足将AnnotationProxy与ProxyIndirectionHandler串联的类

BadAttributeValueExpException.toString
ExtractorEventTransformer.equals
extractorOld
Proxy.newProxyInstance("".class.getClassLoader(),new Class[]{ ValueExtractor.class},AnnotationProxy)
annotationType
Annotation
values
hashMap
key:getConnection
value:aaa
invoke
method.invoke(this, args)
equals
getAnnotationMemberValue
annotation name=getConnection
extractorNew
Proxy.newProxyInstance(Xxx_I2_2_1_4_0.class.getClassLoader(), new Class[]{Annotation.class,RowSetInternal.class,ValueExtractor.class},CompositeInvocationHandlerImpl)
classToInvocationHandler
Annotation
AnnotationProxy
annotationType
Annotation
values
key:annotationType
value:RowSetInternal.class
method:annotationType()
RowSetInternal
ProxyIndirectionHandler
ValueHolder
JdbcRowSetImpl
invoke
getValue
get(method.getDeclaringClass)
invoke
return:RowSetInternal.class
name:getConnection

TONGDAO

KCon
2024

KCon 2024

THANKS

汇报人：2rpang

时间：2024.08.24