When a malware is more complex than the paper.

sebdraven.medium.com/when-a-malware-is-more-complex-than-the-paper-5822fc7ff257

August 29, 2018

Seb drave

Sebdraven
Aug 28, 2018

Fireye has published a paper of the backdoor Felixroot after using two vulnerabilities CVE-2017–0199 and CVE-2017–11882. The RTF document drops an executable.

And the Analyst explains:

The dropped executable (MD5: 78734CD268E5C9AB4184E1BBE21A6EB9) contains the compressed FELIXROOT dropper component in the Portable Executable (PE) binary overlay section. When it is executed, it creates two files: an LNK file that points to %system32%\rundll32.exe, and the FELIXROOT loader component. The LNK file is moved to the startup directory. Figure 5 shows the command in the LNK file to execute the loader component of FELIXROOT.

Microsoft Office Vulnerabilities Used to Distribute FELIXROOT Backdoor in Recent Campaign "…

Campaign Details In September 2017, FireEye identified the FELIXROOT backdoor as a payload in a campaign targeting…

www.fireeye.com

But it’s no so easy. The dropper copies two PE files after using RC4 and a decompression function custom in memory.

The two PE file are an installer and the backdoor Felixroot.

The installer puts on the disk the backdoor and after decrypting strings, it creates the persistance (a lnk in startup folder) and execute run32dll with Felixroot and uses some technics anti forensic to change the timestamps of the backdoor.

Now all technical details !

Encryption and decompression

load the overlay in memory

After a look with Pestudio, the overlay is 53% of the dropper and the entropy is 7,994. it’s too high for a compression.

Overlay of the dropper

to dump the overlay, two lines of python are enough. The overlay starts at 0xD800 in the file.

```
overlay =
open(‘573ea78afb50100f896185164da3b8519e2e0f609a34a7c70460eca5b4ae640d’,’rb’).read()
[0xD800:]

open(‘overlay.dump’,’wb’).write(overlay)
```

So launch the debugger to understand how this overlay is used by the dropper.

The dropper starts to read itself.

Read itself

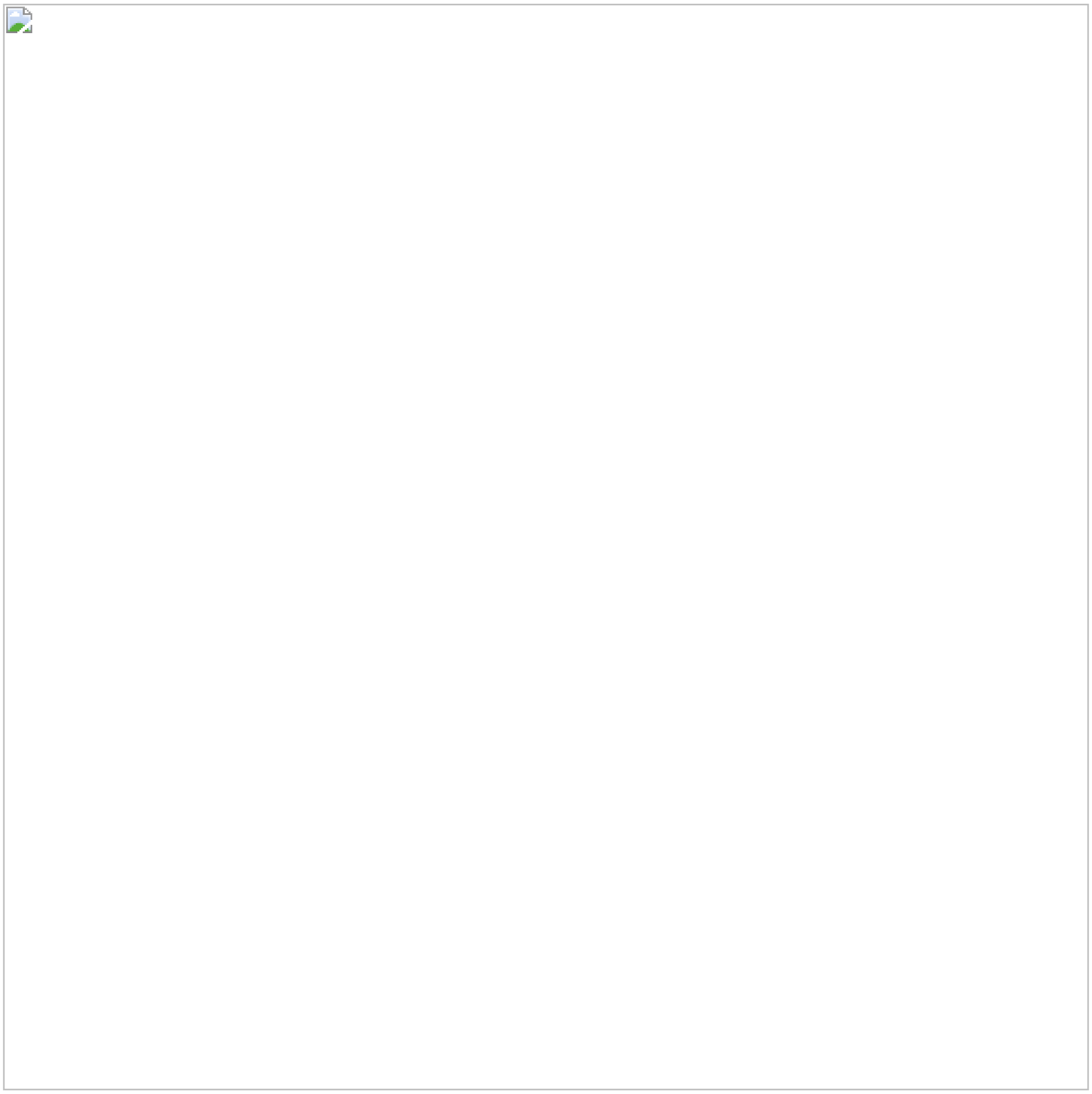
The file handler is in EAX as value 288.

If we check in IDA, this part is badly interpreted by IDA. It’s patched at runtime.

erro in IDA

So the best way, it’s to set a breakpoint at the CreateFile and ReadFile.

So it reads and stores the result in [ebp + C].(here 194408)



And the dropper seeks to D800 to set EAX at the start of the overlay.

RC4 and Custom decompression

Overlay in memory

if we check in IDA where we are.

sub_406681

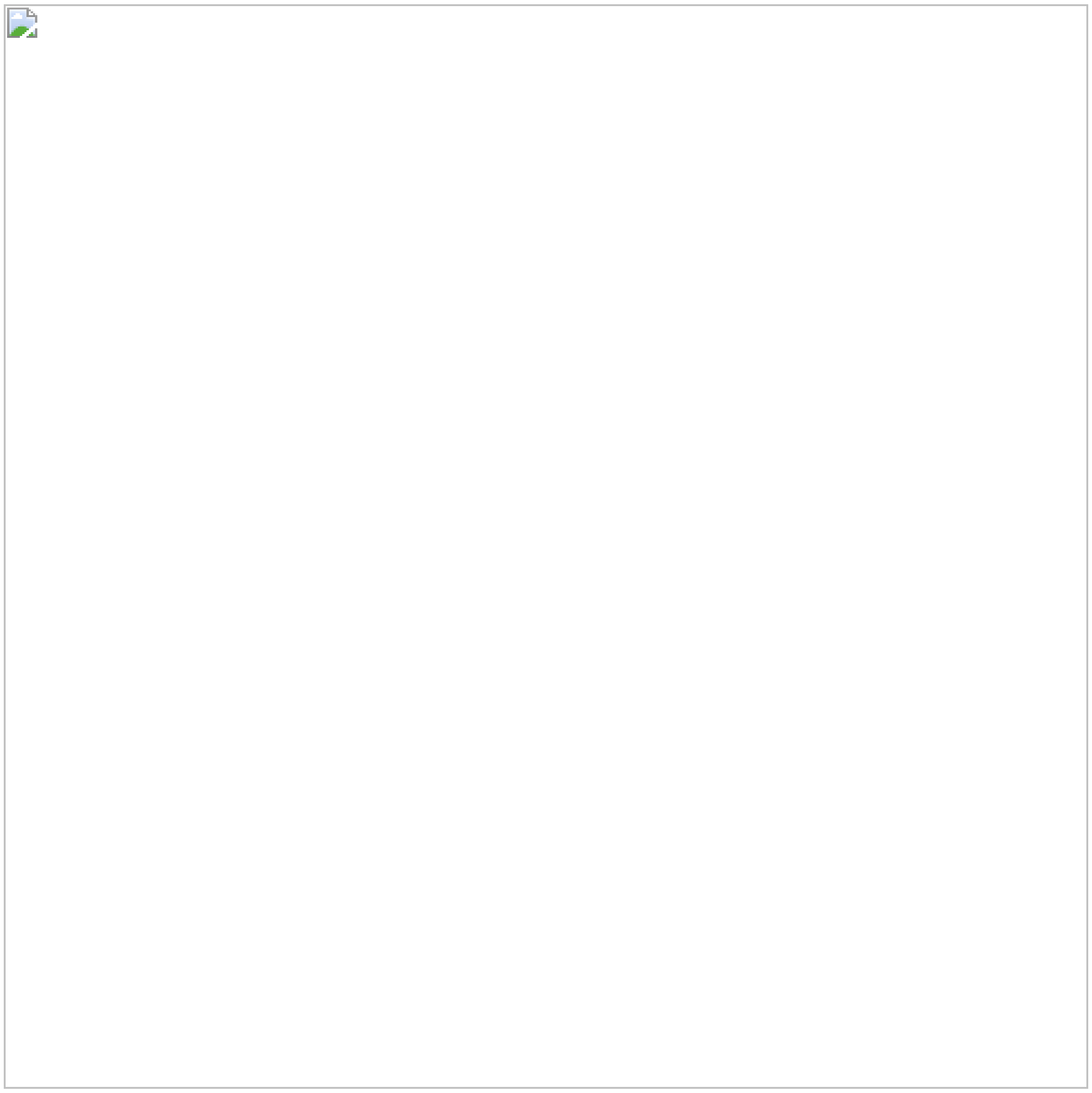
If we check the the graph overview,this function is in a huge function with many jmp.

It seems this function is like a packer.

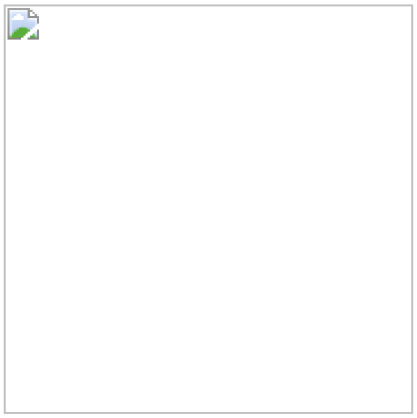
In the second step, it reads 40 bytes (from C08 to C30)

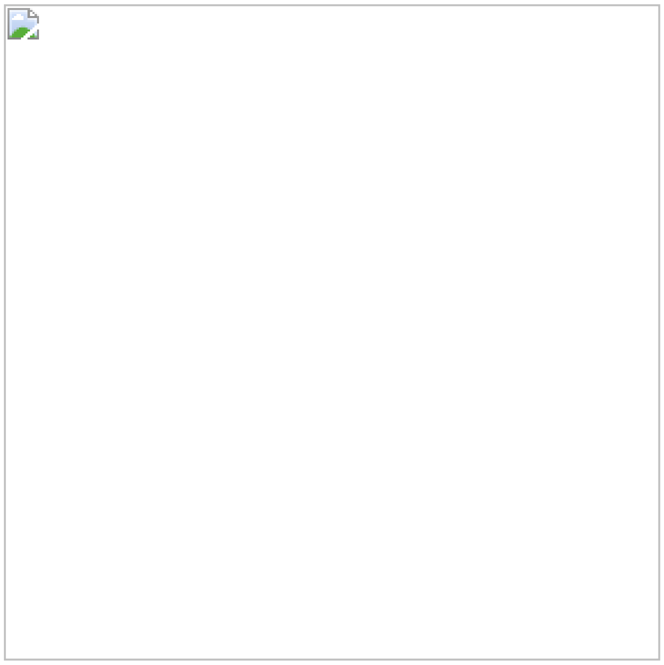
The loop is made by the the jmp to go to the start of the function if the 40bits are not read.

After reading the 40 bits, we have a loop of 256 steps, to store 01 to 256 on the stack.


init of RC4

And it manipulates the 40 bytes and stores the result on the stack in a loop of 256 steps.

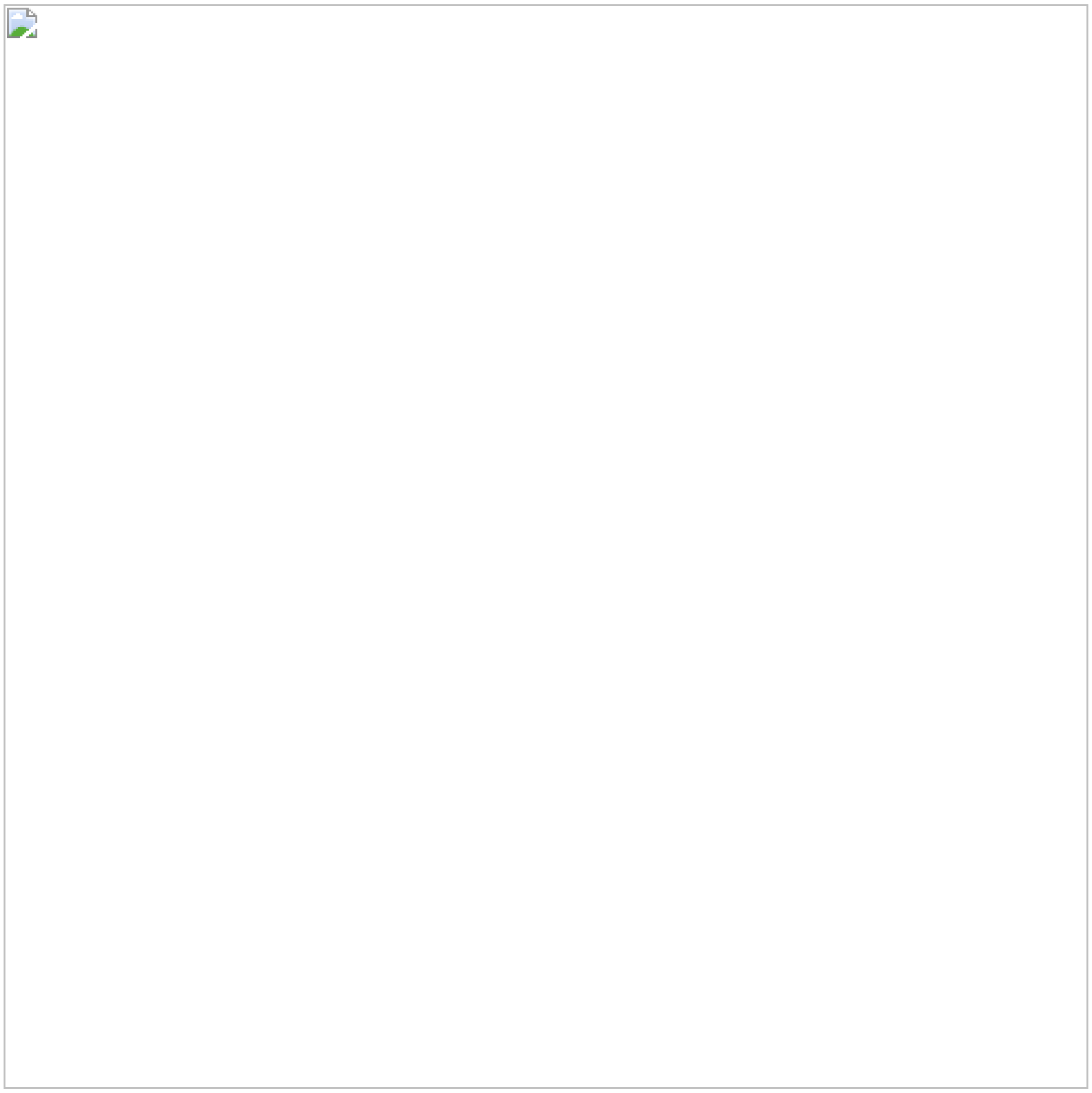



Init of RC4

And a third function with aritmethic operations works on the overlay with the results of the two lasted functions

[EBP+C] store F9D8. Remember !? It’s the size of file.

The result of this function is stored at the same place of the overlay in C30.



Ok, everybody has recognized the three steps of RC4.

if we do a comparaison using python langage, the first functions is:

```
self.state = list(range(256))
```

The second function is:

```
def init(self, key):        for i in range(256):            self.x = (ord(key[i %
len(key)]) + self.state[i] + self.x) & 0xFF            self.state[i], self.state[self.x]
= self.state[self.x], self.state[i]        self.x = 0
```

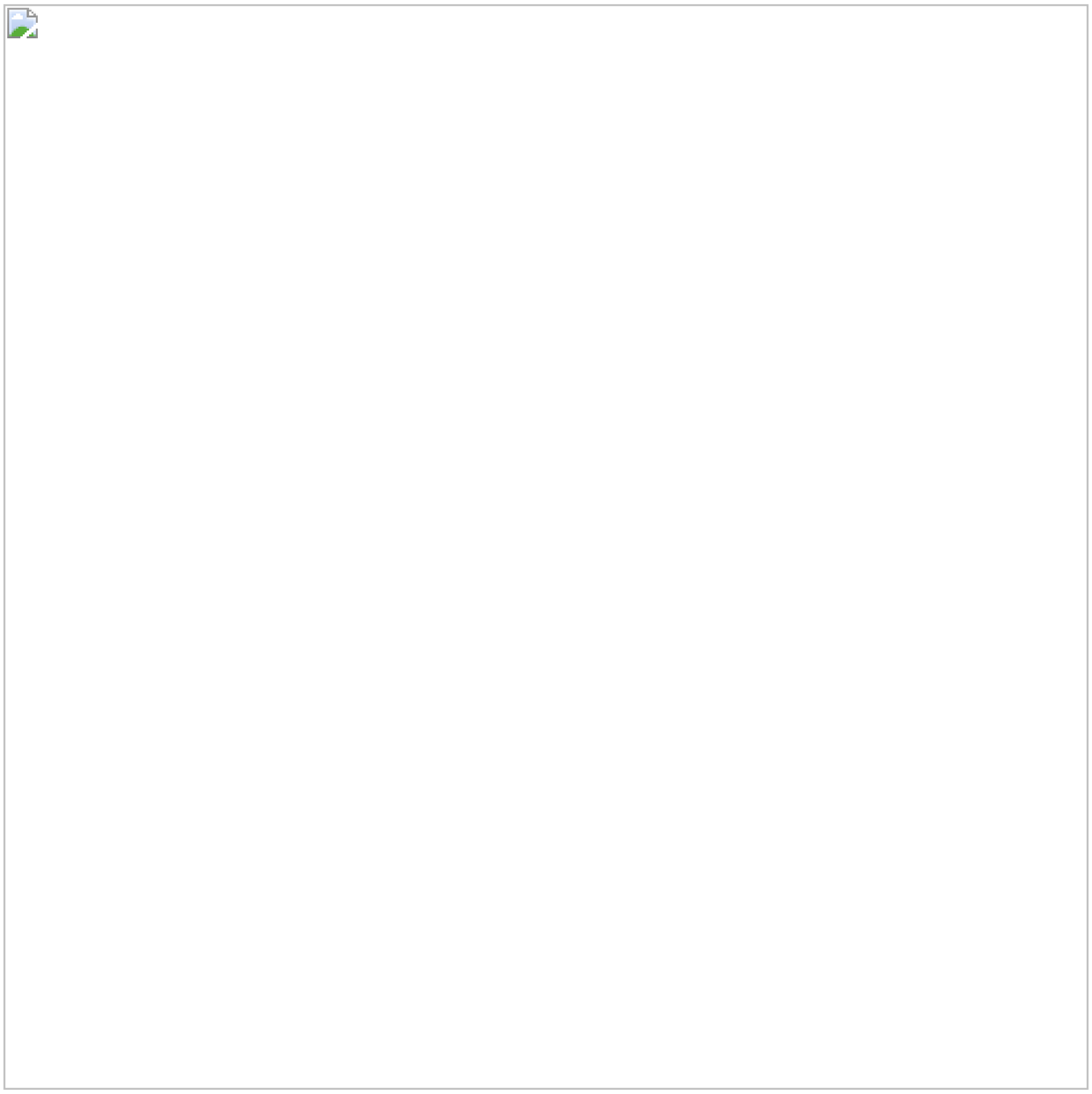
And the third function is:

```
def decrypt(self, input):        output = [None]*len(input)        for i in
range(len(input)):            self.x = (self.x + 1) & 0xFF            self.y =
(self.state[self.x] + self.y) & 0xFF            self.state[self.x], self.state[self.y] =
self.state[self.y], self.state[self.x]            output[i] = chr((ord(input[i]) ^
self.state[(self.state[self.x] + self.state[self.y]) & 0xFF]))        return
''.join(output)
```

So here, the key of the RC4 is the first 40 bytes of the overlay.

1B 73 B4 17 5E 5F 14 59 AF F7 BA AF DA 75 AB F5 19 4D 32 50 36 01 46 30 09 AB 9C 09 4D B2 74 01 9E C0 C0 9E FD B9 ED E5

The result seems to be a PE file but not totally.

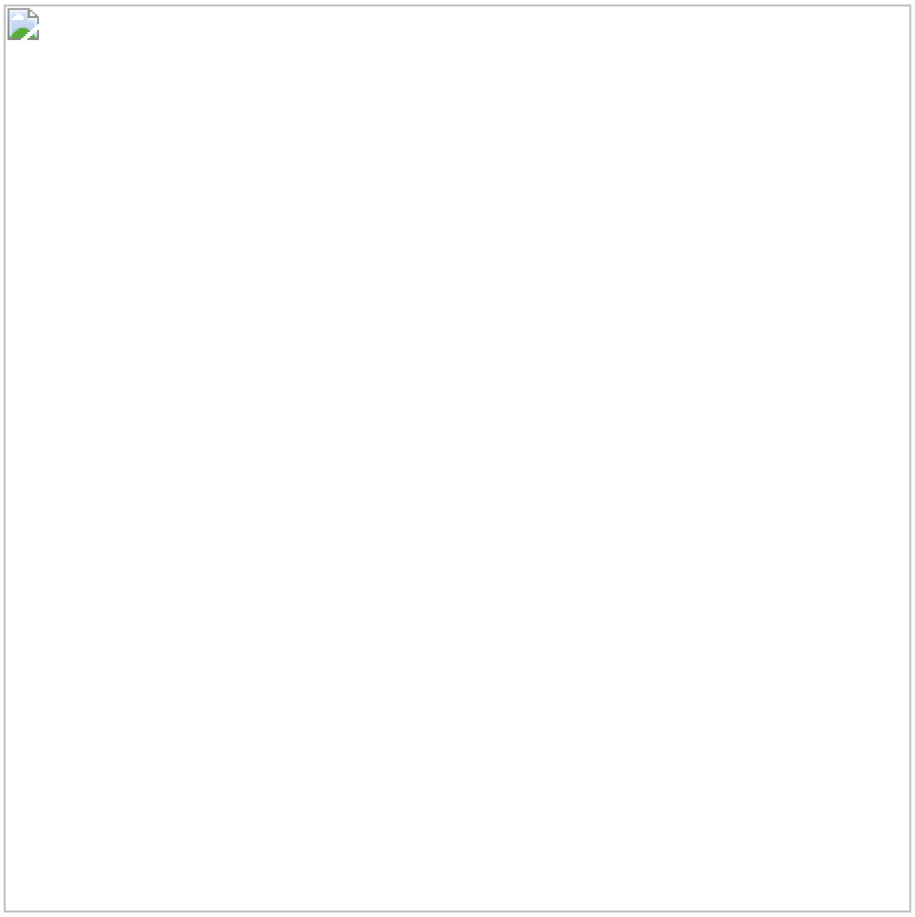


After that, the dropper increases the stack from 12F000 to 12C000 before launching the decompression function (seemly custom after many searches, but if it's not that, write a comment at the end of this post !)

the dropper puts the three bytes [AB,CD,EF] in the stack.

The algorithm used in the first AB,C and secondly D, EF in this function sub_004055FC.

AB,C



D,EF

If C or D == 0 then the dropper writes AB or EF.

If C != 0 then the dropper writes 0 and the number of 0 depends on AB

if D!= 0 then the dropper write 0 and the number of 0 depends on EF

We develop many examples to better understand:

4D 00 5A -> 4D 5A

90 00 00 -> 90 00

03 00 00 -> 03 00

51 10 04 -> 00 00 04

The result is stored in a buffer. The address of the buffer is stored by EAX.

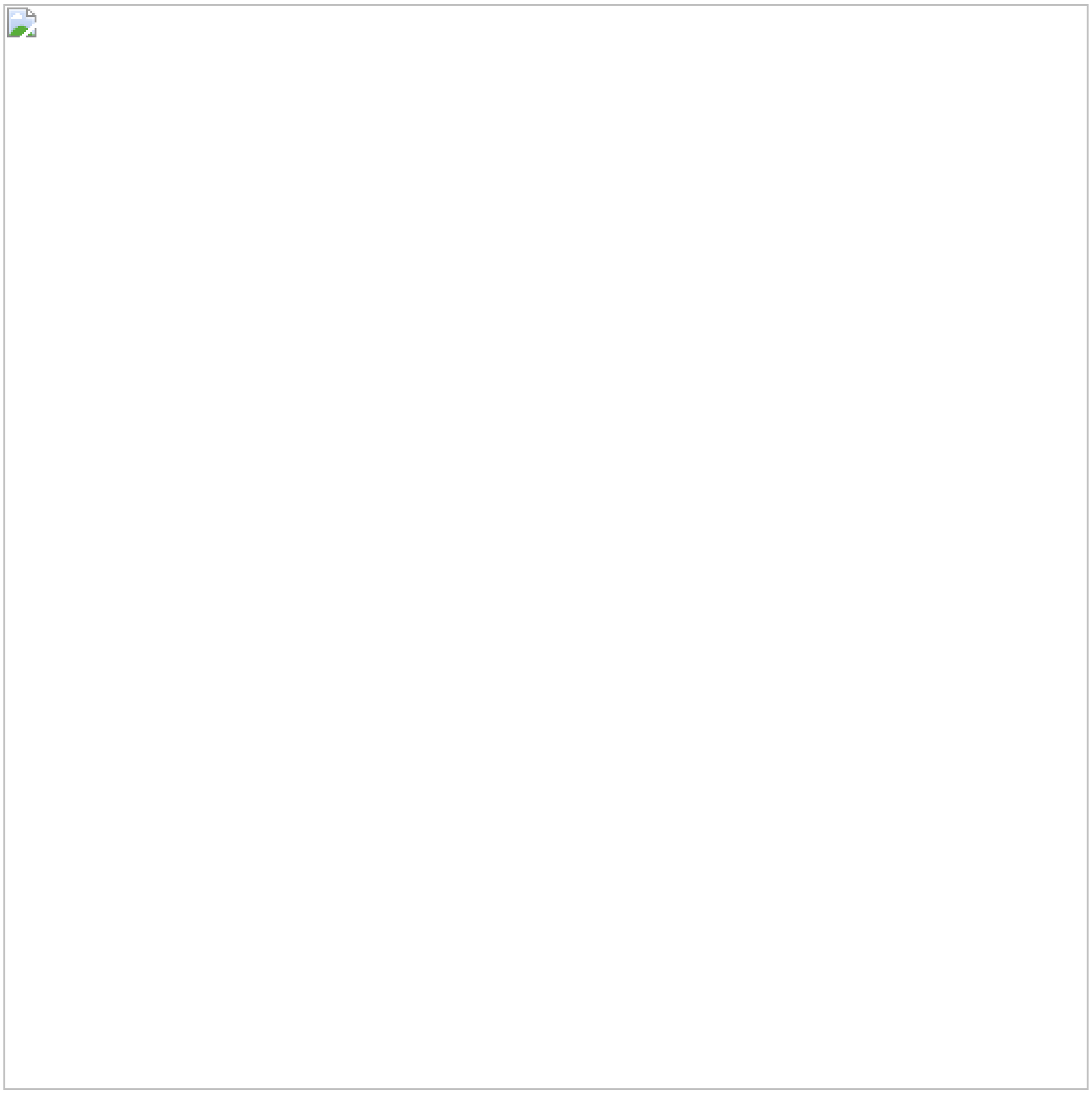
The dropper decodes two PE Files: another dropper which we name drop and the backdoor Felixroot.

Felixroot is copyied to 378C38.

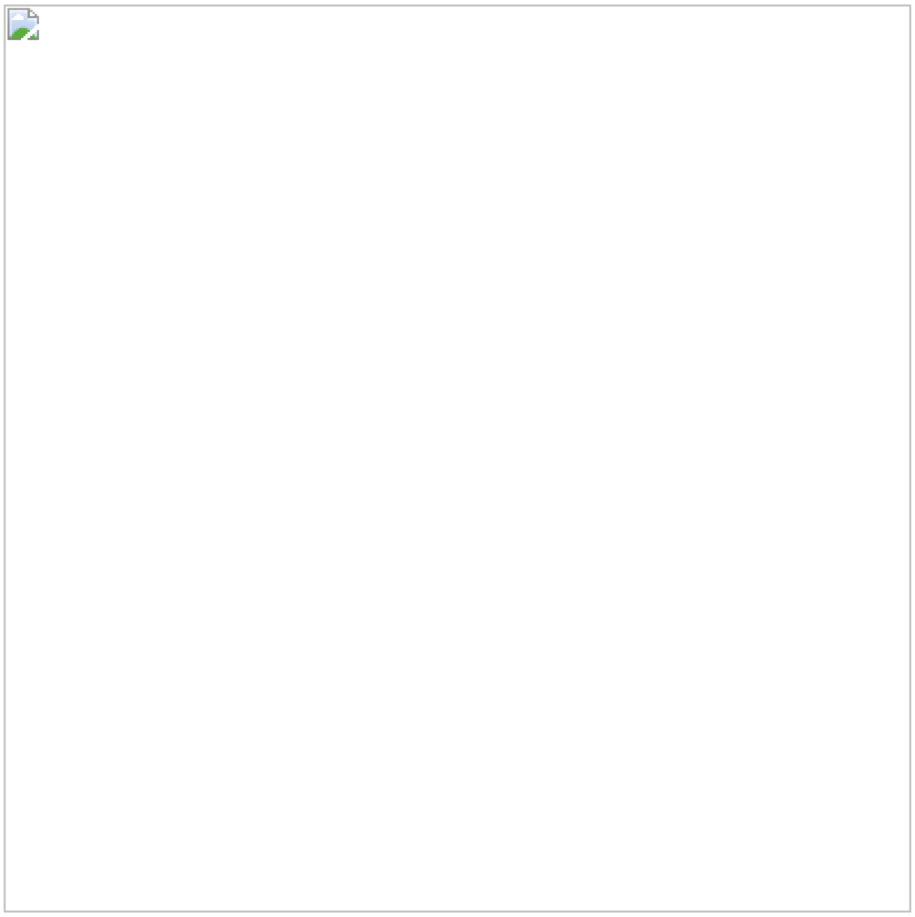
drop is copied after a Virtualloc at 20000 and drop is executed in 00021964.

Installation

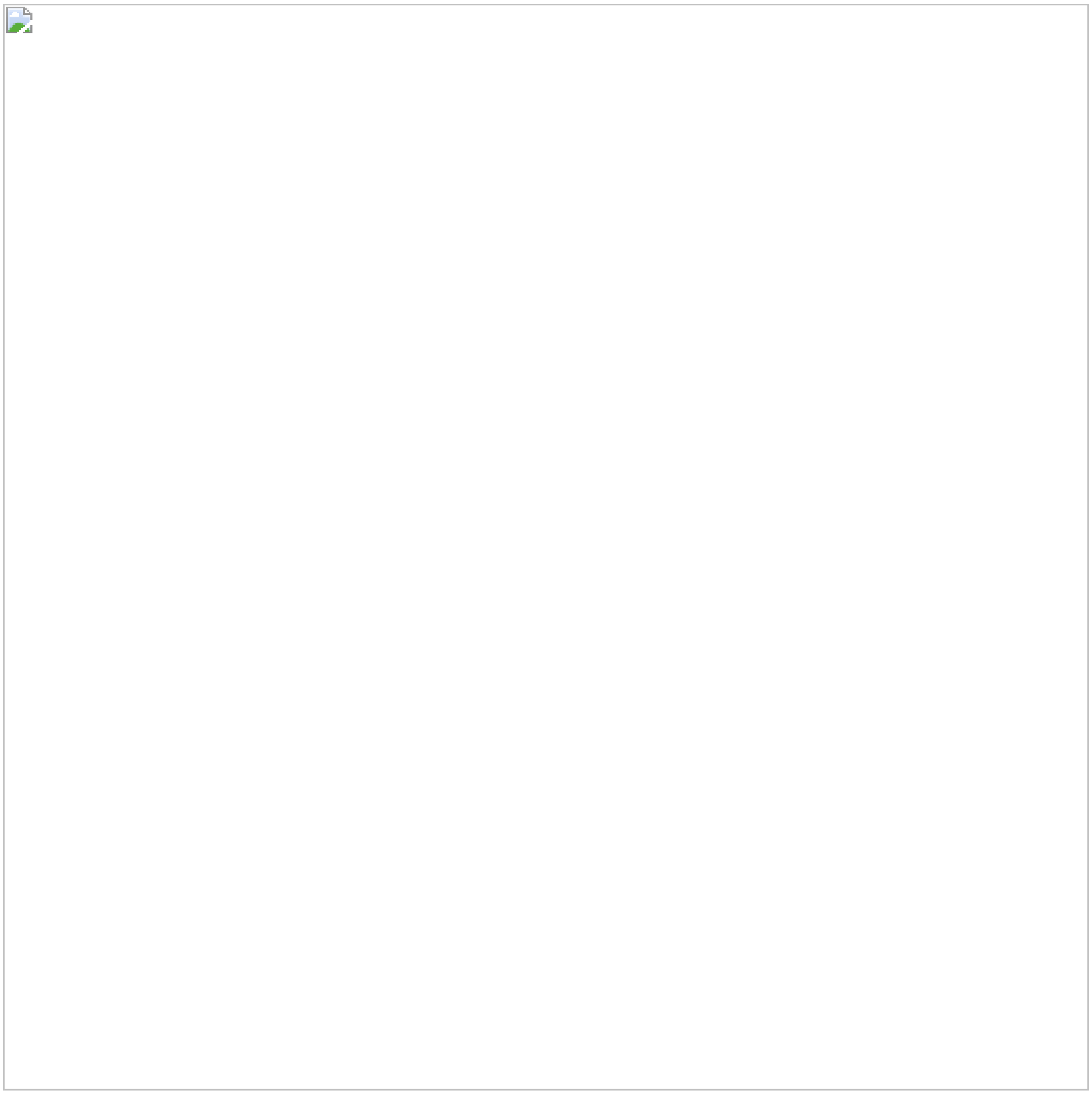
drop verifies if it’s executed after the dropper by checking a mutex.



If it’s ok, drop checks if it’s executed on a 64bits systems to correctly set the parameters of decoding strings



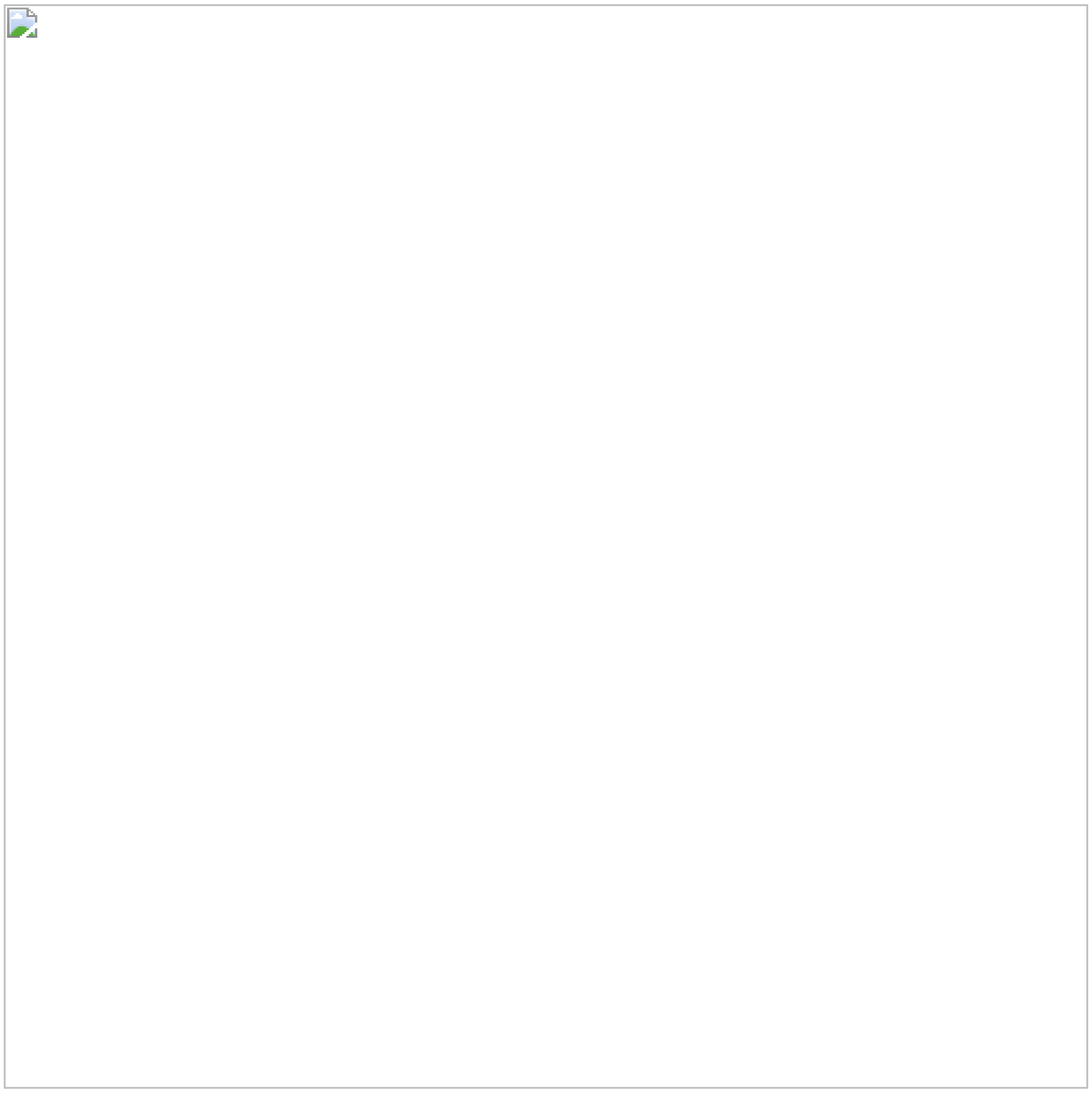
Drop decodes the first string depending on if it’s 64bits or not.



offset unk_ is the key of the decode functions.

The function is a XOR with a and with FF.

The data to decrypt is in drop ressources.



The result is the stored the result in a buffer.

The result of the first decoding function is \\System32.

rundll32.exe is the result of the second decoding.

drop drops the backdoor after decoding strings to set in which folder the backdoor is dropped.

in first it decodes the pattern of the path L"%lS\\%lS\\%lS.dbf"

It chooses the special folder: %APPDATA%/Roaming/Microsoft/

In 32bits, this folder is not used so it's very easy to hide a malware.

And the path of file is: "C:\\Users\\IEUser\\AppData\\Roaming\\Microsoft\\{22B4CEF1–633C-4F94–824E-0C207AC4F2DF}.dbf"

The name of file changes at each execution.

drop writes the backoor in the disk from 378C38in sub 210E5.

drop decodes a string to have the path: c:\\windows\\system32\\msvcrt.dll in sub_00021845, drop catches the creation date of c:\\windows\\system32\\msvcrt.dll drop changes the attributes of the timestamps (creation date of the file, last modified…) of the backdoor switching the value with L”c:\\windows\\system32\\msvcrt.dll” in sub_000218AF

the persistance is created. The first step is the decoding of the name of file in 00021A1A. the result is .lnk

After ther create the persistance of the backdoor in sub_000211D7. It’s a shortcut installed in startup folder and create a copy in roaming folder.

in sub_216D1, drop installs the shortcut in the Startup Folder and copies the shorcut and execute with the function ShellExecute.

```
0012DC84 0012DCA0 L”C:\\Windows\\system32\\cmd.exe”
0012DC88 0012E6C8 L”/c move \”C:\\Users\\IEUser\\AppData\\Roaming\\ .lnk\”
\”C:\\Users\\IEUser\\AppData\\Roaming\\Microsoft\\Windows\\Start
Menu\\Programs\\Startup\\ .lnk\””
```

Shortcut to restart

The lnk runs run32dll.exe with the .dbf and ordinal 1 of the export of the dll.

Few words about Threat Intel

It’s strange that FireEye hasn’t published a full paper with the analysis of the dropper and the backdoor.

The dropper uses many very interesting techniques:

- rc4 encryption
- decompression function custom
- run32dll to bypass applocker and AV
- change timestamps of the installed files

- decrypting strings to install the backdoor
- decrypting strings for the persistance settings

The function sub_406681 is very interesting and it’s very difficult to make a yara rules on it because there are many jump to have enough binaries to make a rule. the rc4 encryption and a decompression function are in this function.

Thanks

thank to the *zone de confort* to try to understand this f… decompression function and thank to @FliegenEinhorn to find the sample !