Aria-Body Loader? Is that you?

medium.com/insomniacs/aria-body-loader-is-that-you-53bdd630f8a1

BlueMonkey

September 29, 2021

BlueMonkey

Sep 29, 2021

·

9 min read

Hello! This is my first time writing a blog on a loader which I had gotten hold of. So, I am a new analyst in the Malware Analysis field and I am trying to do a research into cobalt strike. Recently, I ran a modified public YARA rule for cobalt strike on Virus Total and from the scan, I found two samples which I find interesting.

> 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f567d274a184bb67af1eb19c
>
> c5696e660f3cfa9232756418e40ad18729cfe32fb284bba2314dd523ba527258

These two samples have a relative large size (17.05 MB) as compared to other files and their size is exactly the same. Additionally, their upload timing is quite close to each other, around 3 minutes apart. I started to analyze the files and from my finding, it doesn't look anything like a cobalt strike loader. Turning to my friend who have much more experience in malware analysis, I told him my findings and he told me that from what I had described, it sounds a little bit like Aria-Body instead. So I did some read up on Aria-Body and here are what I had found….

In 2020, Check Point Research release a write up (http://research[.]checkpoint[.]com/2020/nikon-apt-cyber-espionage-reloaded) describing how Naikon APT group is using Aria-body. In the report, they summarized the loader to have these capabilities:

1. Establish persistence via the Startup folder or the Run registry key [some variants].

2. Inject itself to another process such as rundll32.exe and dllhost.exe [some variants].

3. Decrypt two blobs: Import Table and the loader configuration.

4. Utilize a DGA algorithm if required

5. Contact the embedded / calculated C&C address in order to retrieve the next stage payload.

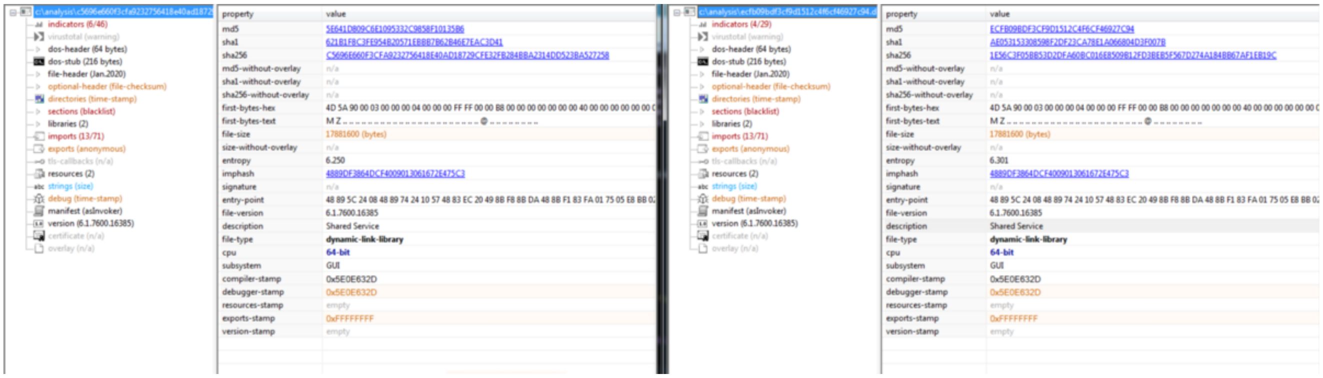
6. Decrypt the received payload DLL (Aria-Body backdoor).

7. Load and execute an exported function of the DLL — calculated using djb2 hashing algorithm.

Take note on these points as I will be mentioning some of them in this post.

Analysis of the samples

As I had mentioned in the intro, the first thing that I noticed about the two sample is that both of them have exactly the same size. When I loaded them into PE studio, I noticed that they share the same compiler, debugger and exports timestamp.



c:\analysis\c5696e660f3cfa9232756418e40ad1872

- indicators (6/46)
- virustotal (warning)
- dos-header (64 bytes)
- dos-stub (216 bytes)
- file-header (Jan.2020)
- optional-header (file-checksum)
- directories (time-stamp)
- sections (blacklist)
- libraries (2)
- imports (13/71)
- exports (anonymous)
- tls-callbacks (n/a)
- resources (2)
- strings (size)
- debug (time-stamp)
- manifest (asInvoker)
- version (6.1.7600.16385)
- certificate (n/a)
- overlay (n/a)

property	value
md5	5E641D809C6E1095332C9858F10135B6
sha1	621B1F8C3FE954B20571EBBB7B62B46E7EAC3D41
sha256	C5696E660F3CFA9232756418E40AD18729CFE32FB284BBA2314DD523BA527258
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 0
first-bytes-text	M Z .... @ ....
file-size	17881600 (bytes)
size-without-overlay	n/a
entropy	6.250
imphash	4889DF3864DCF4009013061672E475C3
signature	n/a
entry-point	48 89 5C 24 08 48 89 74 24 10 57 48 83 EC 20 49 8B F8 8B DA 48 8B F1 83 FA 01 75 05 E8 BB 02
file-version	6.1.7600.16385
description	Shared Service
file-type	**dynamic-link-library**
cpu	**64-bit**
subsystem	GUI
compiler-stamp	0x5E0E632D
debugger-stamp	0x5E0E632D
resources-stamp	empty
exports-stamp	0xFFFFFFFF
version-stamp	empty

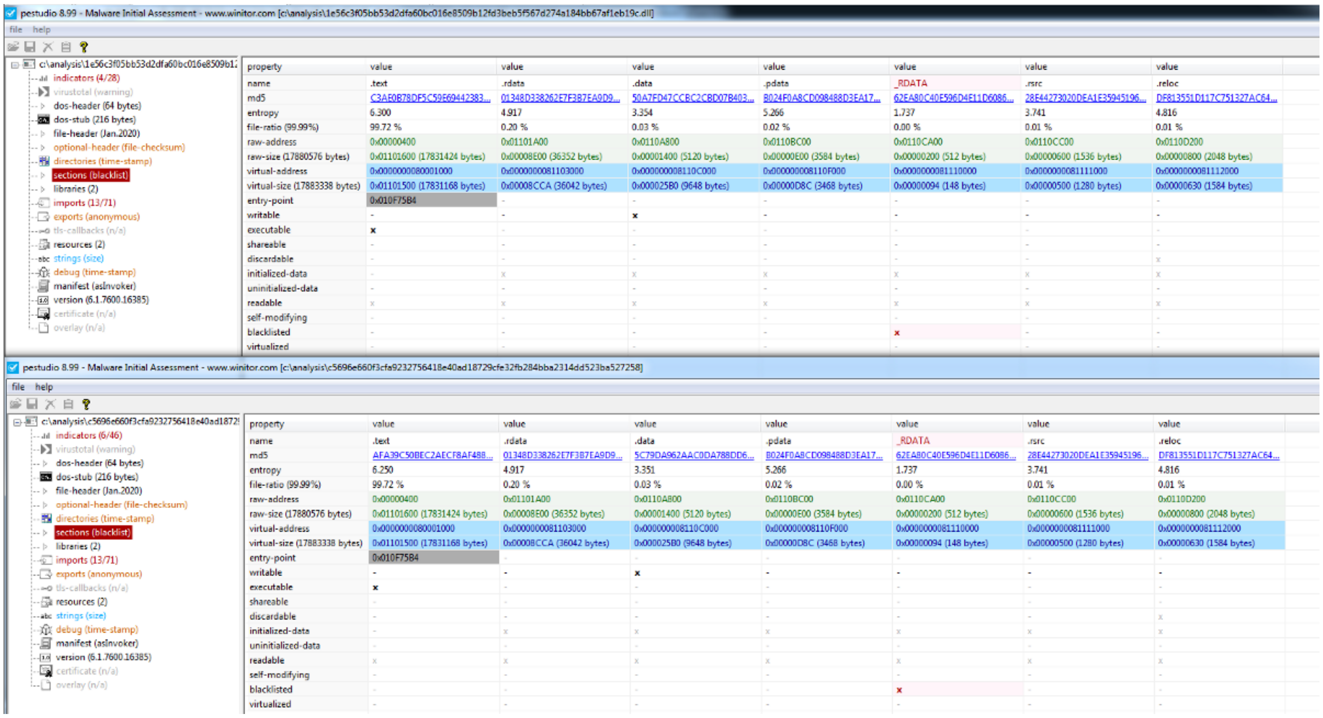
c:\analysis\ecfb09bdf3cf9d1512c4f6cf46927c94.d

- indicators (4/29)
- virustotal (warning)
- dos-header (64 bytes)
- dos-stub (216 bytes)
- file-header (Jan.2020)
- optional-header (file-checksum)
- directories (time-stamp)
- sections (blacklist)
- libraries (2)
- imports (13/71)
- exports (anonymous)
- tls-callbacks (n/a)
- resources (2)
- strings (size)
- debug (time-stamp)
- manifest (asInvoker)
- version (6.1.7600.16385)
- certificate (n/a)
- overlay (n/a)

property	value
md5	ECFB09BDF3CF9D1512C4F6CF46927C94
sha1	AE05315330859BF2DF23CA78E1A066804D3F007B
sha256	1E56C3F05BB53D2DFA60BC016E8509B12FD3BEB5F567D274A184BB67AF1EB19C
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 0
first-bytes-text	M Z .... @ ....
file-size	17881600 (bytes)
size-without-overlay	n/a
entropy	6.301
imphash	4889DF3864DCF4009013061672E475C3
signature	n/a
entry-point	48 89 5C 24 08 48 89 74 24 10 57 48 83 EC 20 49 8B F8 8B DA 48 8B F1 83 FA 01 75 05 E8 BB 02
file-version	6.1.7600.16385
description	Shared Service
file-type	**dynamic-link-library**
cpu	**64-bit**
subsystem	GUI
compiler-stamp	0x5E0E632D
debugger-stamp	0x5E0E632D
resources-stamp	empty
exports-stamp	0xFFFFFFFF
version-stamp	empty

Similar Timestamp

Now looking at the section's metadata, we can see that all but two of them have the same hash. The two sections with the different hash is the *.text* and the *.data* section. Although they have a different hash, we can see that the raw-size, virtual-size, raw-address and virtual-address are the same.

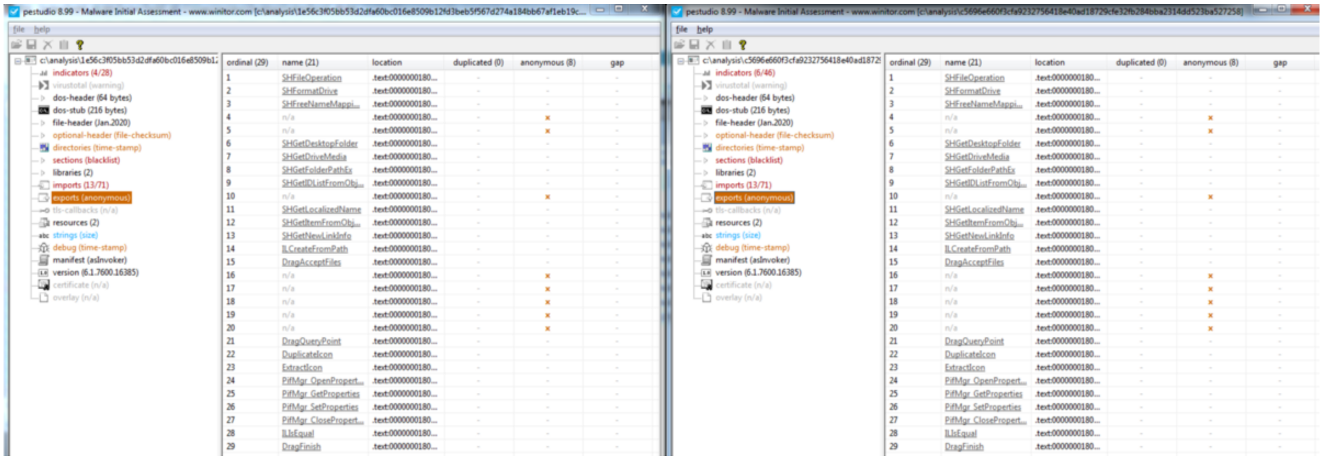


property	value	value	value	value	value	value	value
name	.text	.rdata	.data	.pdata	_RDATA	.rsrc	.reloc
md5	C3AE0B78DF5C59E69442383...	01348D338262E7F3B7EA9D9...	50A7FD47CCBC2CBD07B403...	B024F0A8CD098488D3EA17...	62EA80C40E596D4E11D6086...	28E44273020DEA1E35945196...	DF813551D117C751327AC64...
entropy	6.300	4.917	3.354	5.266	1.737	3.741	4.816
file-ratio (99.99%)	99.72 %	0.20 %	0.03 %	0.02 %	0.00 %	0.01 %	0.01 %
raw-address	0x00000400	0x01101A00	0x0110A800	0x0110BC00	0x0110CA00	0x0110CC00	0x0110D200
raw-size (17880576 bytes)	0x01101600 (17831424 bytes)	0x00008E00 (36352 bytes)	0x00001400 (5120 bytes)	0x00000E00 (3584 bytes)	0x00000200 (512 bytes)	0x00000600 (1536 bytes)	0x00000800 (2048 bytes)
virtual-address	0x0000000080001000	0x0000000081103000	0x000000008110C000	0x000000008110F000	0x0000000081110000	0x0000000081111000	0x0000000081112000
virtual-size (17883338 bytes)	0x01101500 (17831168 bytes)	0x00008CCA (36042 bytes)	0x000025B0 (9648 bytes)	0x00000D8C (3468 bytes)	0x00000094 (148 bytes)	0x00000500 (1280 bytes)	0x00000630 (1584 bytes)
entry-point	0x010F75B4	-	-	-	-	-	-
writable	-	-	x	-	-	-	-
executable	x	-	-	-	-	-	-
shareable	-	-	-	-	-	-	-
discardable	-	-	-	-	-	-	x
initialized-data	-	x	x	x	x	x	x
uninitialized-data	-	-	-	-	-	-	-
readable	x	x	x	x	x	x	x
self-modifying	-	-	-	-	-	-	-
blacklisted	-	-	-	-	x	-	-
virtualized	-	-	-	-	-	-	-

property	value	value	value	value	value	value	value
name	.text	.rdata	.data	.pdata	_RDATA	.rsrc	.reloc
md5	AFA39C50BEC2AECF8AF488...	01348D338262E7F3B7EA9D9...	5C79DA962AAC0DA788DD6...	B024F0A8CD098488D3EA17...	62EA80C40E596D4E11D6086...	28E44273020DEA1E35945196...	DF813551D117C751327AC64...
entropy	6.250	4.917	3.351	5.266	1.737	3.741	4.816
file-ratio (99.99%)	99.72 %	0.20 %	0.03 %	0.02 %	0.00 %	0.01 %	0.01 %
raw-address	0x00000400	0x01101A00	0x0110A800	0x0110BC00	0x0110CA00	0x0110CC00	0x0110D200
raw-size (17880576 bytes)	0x01101600 (17831424 bytes)	0x00008E00 (36352 bytes)	0x00001400 (5120 bytes)	0x00000E00 (3584 bytes)	0x00000200 (512 bytes)	0x00000600 (1536 bytes)	0x00000800 (2048 bytes)
virtual-address	0x0000000080001000	0x0000000081103000	0x000000008110C000	0x000000008110F000	0x0000000081110000	0x0000000081111000	0x0000000081112000
virtual-size (17883338 bytes)	0x01101500 (17831168 bytes)	0x00008CCA (36042 bytes)	0x000025B0 (9648 bytes)	0x00000D8C (3468 bytes)	0x00000094 (148 bytes)	0x00000500 (1280 bytes)	0x00000630 (1584 bytes)
entry-point	0x010F75B4	-	-	-	-	-	-
writable	-	-	x	-	-	-	-
executable	x	-	-	-	-	-	-
shareable	-	-	-	-	-	-	-
discardable	-	-	-	-	-	-	x
initialized-data	-	x	x	x	x	x	x
uninitialized-data	-	-	-	-	-	-	-
readable	x	x	x	x	x	x	x
self-modifying	-	-	-	-	-	-	-
blacklisted	-	-	-	-	x	-	-
virtualized	-	-	-	-	-	-	-

Similar Section

Looking into the import and export section, this two samples also have a same import and export.



ordinal (29)	name (21)	location	duplicated (0)	anonymous (8)	gap
1	SHFileOperation	.text:0000000180...	-	-	-
2	SHFormatDrive	.text:0000000180...	-	-	-
3	SHFreeNameMappi...	.text:0000000180...	-	-	-
4	n/a	.text:0000000180...	-	x	-
5	n/a	.text:0000000180...	-	x	-
6	SHGetDesktopFolder	.text:0000000180...	-	-	-
7	SHGetDriveMedia	.text:0000000180...	-	-	-
8	SHGetFolderPathEx	.text:0000000180...	-	-	-
9	SHGetIDListFromObj...	.text:0000000180...	-	-	-
10	n/a	.text:0000000180...	-	x	-
11	SHGetLocalizedName	.text:0000000180...	-	-	-
12	SHGetItemFromObj...	.text:0000000180...	-	-	-
13	SHGetNewLinkInfo	.text:0000000180...	-	-	-
14	ILCreateFromPath	.text:0000000180...	-	-	-
15	DragAcceptFiles	.text:0000000180...	-	-	-
16	n/a	.text:0000000180...	-	x	-
17	n/a	.text:0000000180...	-	x	-
18	n/a	.text:0000000180...	-	x	-
19	n/a	.text:0000000180...	-	x	-
20	n/a	.text:0000000180...	-	x	-
21	DragQueryPoint	.text:0000000180...	-	-	-
22	DuplicateIcon	.text:0000000180...	-	-	-
23	ExtractIcon	.text:0000000180...	-	-	-
24	PifMgr_OpenPropert...	.text:0000000180...	-	-	-
25	PifMgr_GetProperties	.text:0000000180...	-	-	-
26	PifMgr_SetProperties	.text:0000000180...	-	-	-
27	PifMgr_ClosePropert...	.text:0000000180...	-	-	-
28	ILIsEqual	.text:0000000180...	-	-	-
29	DragFinish	.text:0000000180...	-	-	-

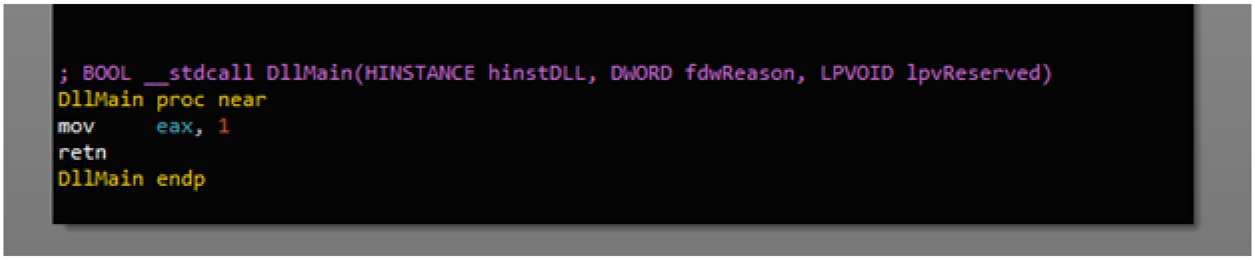
ordinal (29)	name (21)	location	duplicated (0)	anonymous (8)	gap
1	SHFileOperation	.text:0000000180...	-	-	-
2	SHFormatDrive	.text:0000000180...	-	-	-
3	SHFreeNameMappi...	.text:0000000180...	-	-	-
4	n/a	.text:0000000180...	-	x	-
5	n/a	.text:0000000180...	-	x	-
6	SHGetDesktopFolder	.text:0000000180...	-	-	-
7	SHGetDriveMedia	.text:0000000180...	-	-	-
8	SHGetFolderPathEx	.text:0000000180...	-	-	-
9	SHGetIDListFromObj...	.text:0000000180...	-	-	-
10	n/a	.text:0000000180...	-	x	-
11	SHGetLocalizedName	.text:0000000180...	-	-	-
12	SHGetItemFromObj...	.text:0000000180...	-	-	-
13	SHGetNewLinkInfo	.text:0000000180...	-	-	-
14	ILCreateFromPath	.text:0000000180...	-	-	-
15	DragAcceptFiles	.text:0000000180...	-	-	-
16	n/a	.text:0000000180...	-	x	-
17	n/a	.text:0000000180...	-	x	-
18	n/a	.text:0000000180...	-	x	-
19	n/a	.text:0000000180...	-	x	-
20	n/a	.text:0000000180...	-	x	-
21	DragQueryPoint	.text:0000000180...	-	-	-
22	DuplicateIcon	.text:0000000180...	-	-	-
23	ExtractIcon	.text:0000000180...	-	-	-
24	PifMgr_OpenPropert...	.text:0000000180...	-	-	-
25	PifMgr_GetProperties	.text:0000000180...	-	-	-
26	PifMgr_SetProperties	.text:0000000180...	-	-	-
27	PifMgr_ClosePropert...	.text:0000000180...	-	-	-
28	ILIsEqual	.text:0000000180...	-	-	-
29	DragFinish	.text:0000000180...	-	-	-

Similar Export

Looking at the static properties analysis, these two files seems like twins. They could have the same origins or they might be built using a builder. But this is just my speculation at this point as there are not enough information to support the claim.

Looking for the "action"

Now that I had done the analysis on the files properties and confirmed that it is a 64 bit DLL, it's time to throw the sample 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f567d274a184bb67af1eb19c into IDA for analysis. After IDA has finished loading, the first thing that it displayed is this:

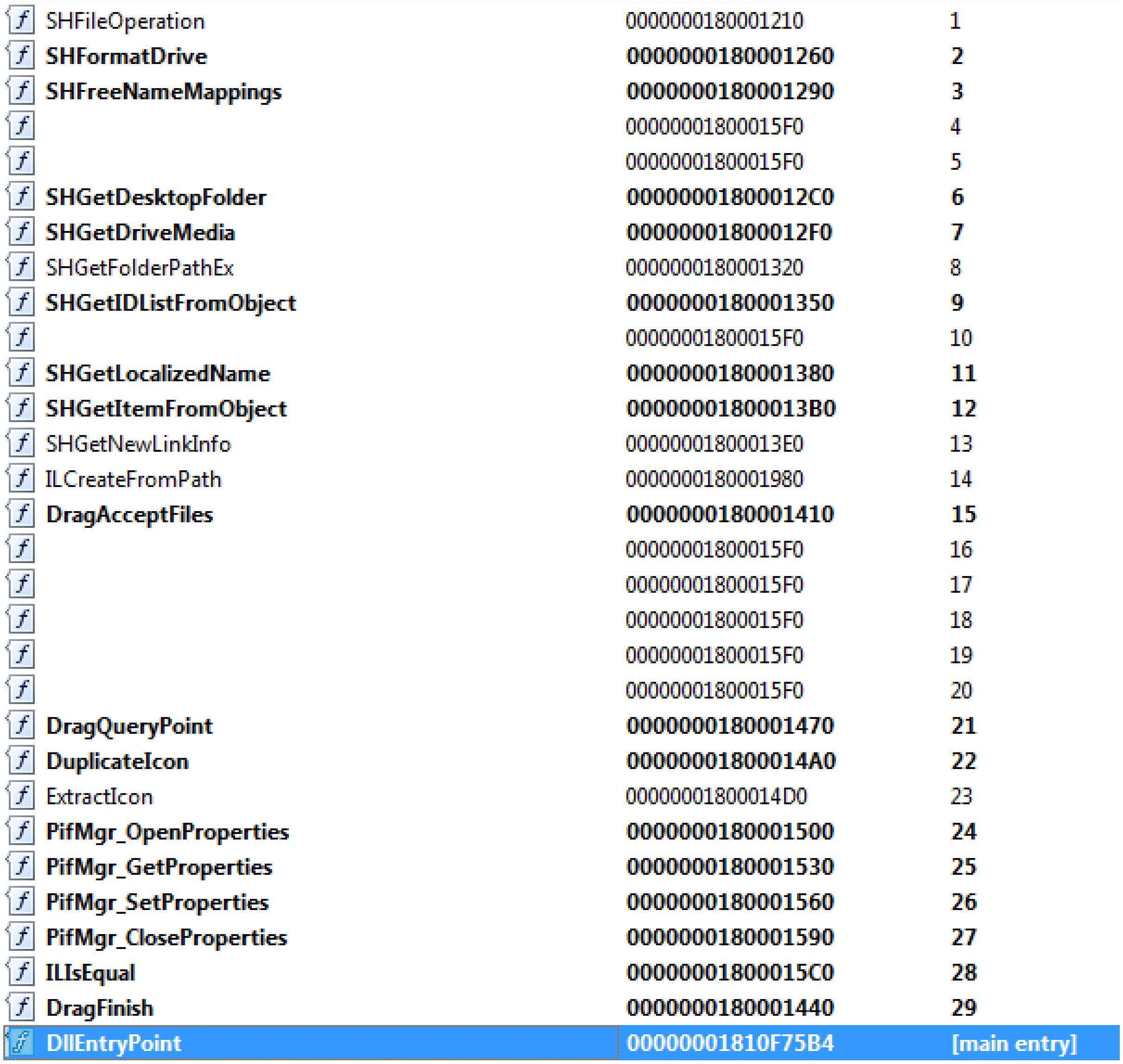


```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
DllMain proc near
mov     eax, 1
retn
DllMain endp
```

DllMain

That’s all ._. not helpful at all. Seems like I have to find the “action” through other means~~

Since this is a DLL, we can try to look for the “action” by checking the export functions.



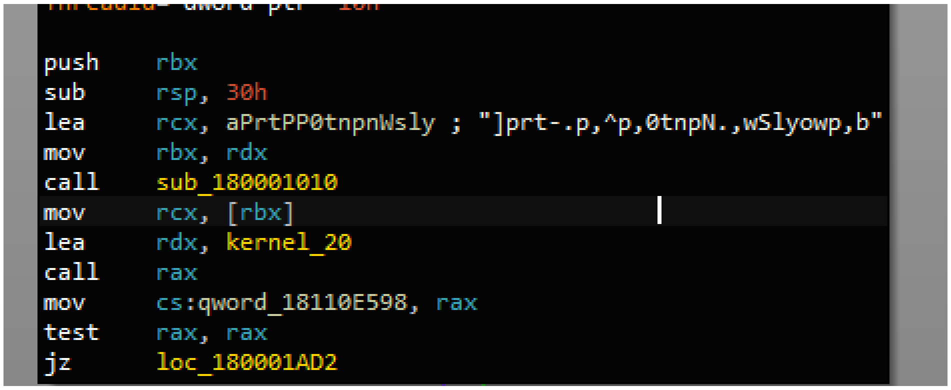
SHFileOperation	0000000180001210	1
SHFormatDrive	0000000180001260	2
SHFreeNameMappings	0000000180001290	3
	00000001800015F0	4
	00000001800015F0	5
SHGetDesktopFolder	00000001800012C0	6
SHGetDriveMedia	00000001800012F0	7
SHGetFolderPathEx	0000000180001320	8
SHGetIDListFromObject	0000000180001350	9
	00000001800015F0	10
SHGetLocalizedName	0000000180001380	11
SHGetItemFromObject	00000001800013B0	12
SHGetNewLinkInfo	00000001800013E0	13
ILCreateFromPath	0000000180001980	14
DragAcceptFiles	0000000180001410	15
	00000001800015F0	16
	00000001800015F0	17
	00000001800015F0	18
	00000001800015F0	19
	00000001800015F0	20
DragQueryPoint	0000000180001470	21
DuplicateIcon	00000001800014A0	22
ExtractIcon	00000001800014D0	23
PifMgr_OpenProperties	0000000180001500	24
PifMgr_GetProperties	0000000180001530	25
PifMgr_SetProperties	0000000180001560	26
PifMgr_CloseProperties	0000000180001590	27
ILIsEqual	00000001800015C0	28
DragFinish	0000000180001440	29
DllEntryPoint	00000001810F75B4	[main entry]

Export Functions

From this list of 30 export functions, two of them *DllEntryPoint* and *ILCreateFromPath* caught my attention. After looking through the two functions, I had determined that the *ILCreateFromPath* function contains the “actions” that we are interested in.

Obfuscation

While scrolling through the *ILCreateFromPath* function, I noticed a pattern:



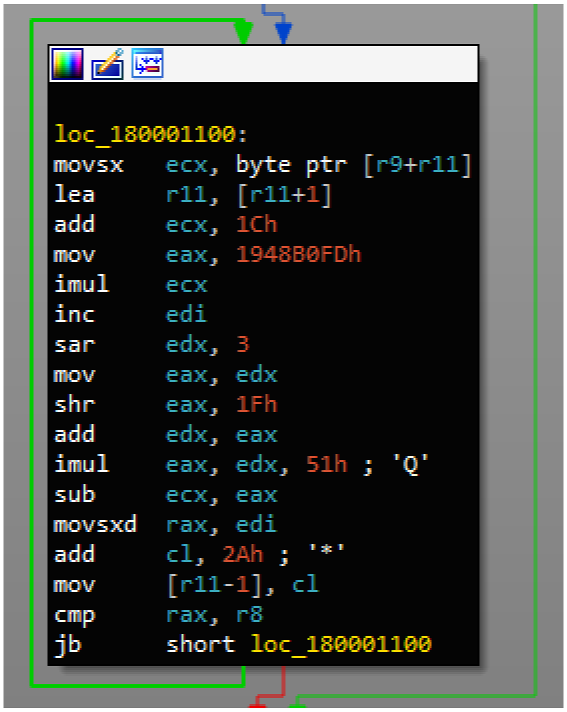
```
push    rbx
sub     rsp, 30h
lea     rcx, aPrtPP0tnpnWsly ; "]prt-.p,^p,0tnpN.,wSlyowp,b"
mov     rbx, rdx
call    sub_180001010
mov     rcx, [rbx]
lea     rdx, kernel_20
call    rax
mov     cs:qword_18110E598, rax
test    rax, rax
jz      loc_180001AD2
```

Encoded String

Noticed that the value “*]prt-.p,^p,0tnpN.,wSlyowp,b*” in the variable *aPrtPP0tnpnWsly* was lea into *rcx* followed by calling the sub function *sub_180001010* then followed by a *call rax*. From my analysis, the function sub_18001010 consist of two parts.

1 — Decode the string

First it will decode a string that is passed in as argument which in this case is the value in the variable *aPrtPP0tnpnWsly*.

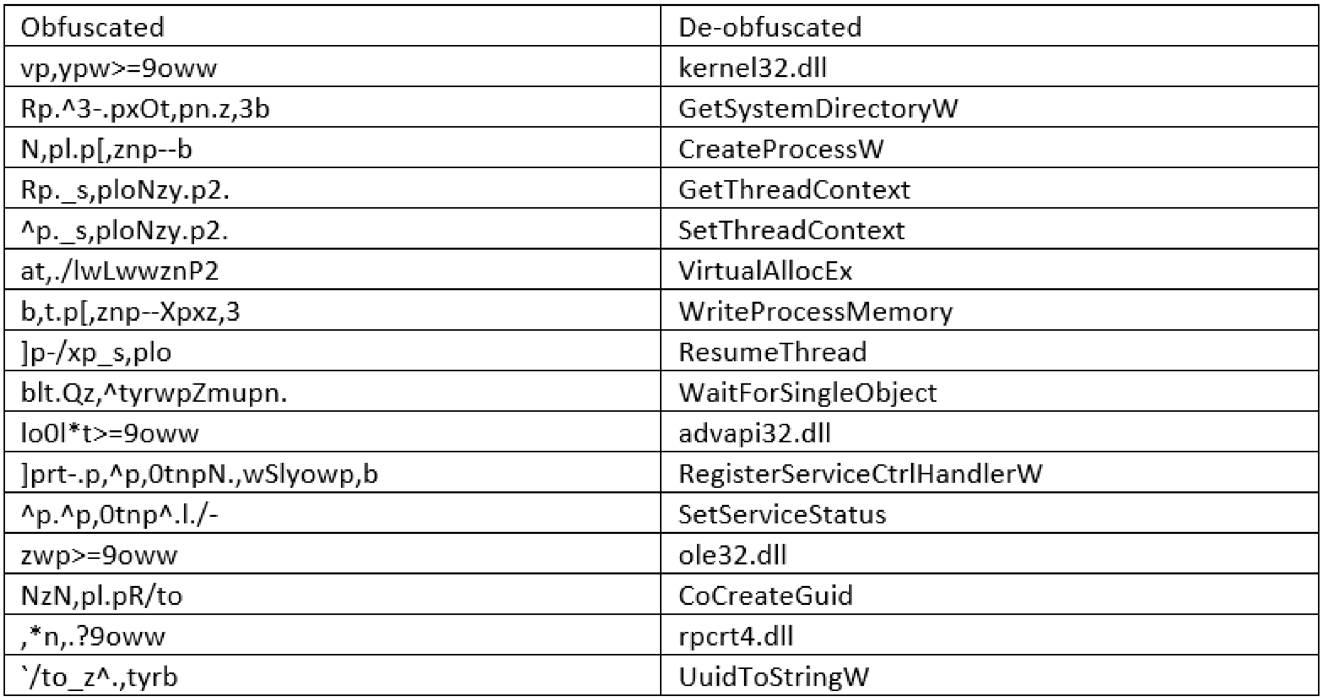


```
loc_180001100:
movsx   ecx, byte ptr [r9+r11]
lea     r11, [r11+1]
add     ecx, 1Ch
mov     eax, 1948B0FDh
imul    ecx
inc     edi
sar     edx, 3
mov     eax, edx
shr     eax, 1Fh
add     edx, eax
imul    eax, edx, 51h ; 'Q'
sub     ecx, eax
movsxd  rax, edi
add     cl, 2Ah ; '*'
mov     [r11-1], cl
cmp     rax, r8
jb      short loc_180001100
```

Decode Section

This function decodes the characters by applying the concept of Substitution cipher where it takes the ASCII value of each character, add 28 follow by mod 81 and finally add 42. This is the formula for the substitution cipher that I had just describe: *plain_text = (cipher_text + 28) % 81 + 42*. Thus, the value of variable *aPrtPP0tnpnWsly* decodes into *RegisterServiceCtrlHandlerW* which is a Win32 API.

Using this formula, I wrote a simple python script to do the decryption and ran it on all the strings found in the same data section as *aPrtPP0tnpnWsly* and here are the decrypted strings:

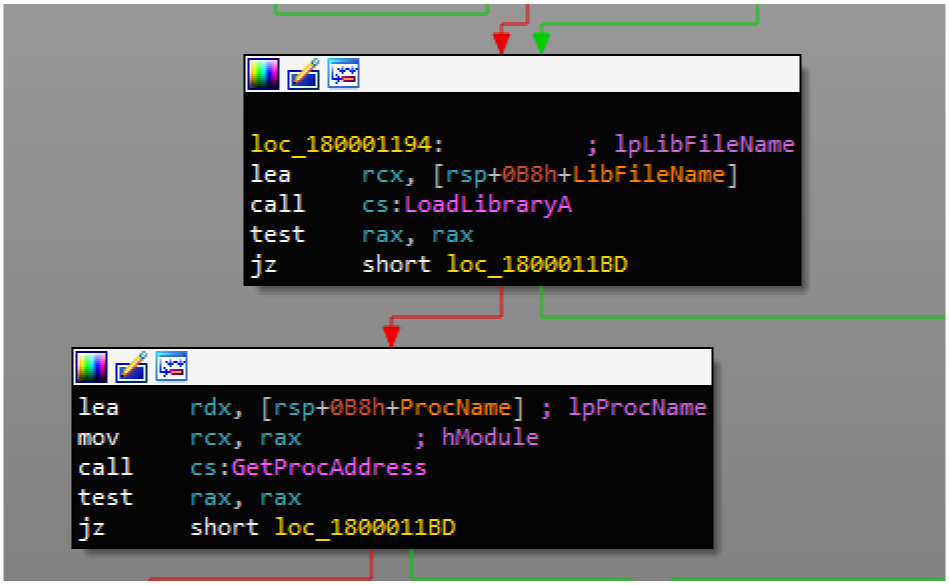


Obfuscated	De-obfuscated
vp,ypw>=9oww	kernel32.dll
Rp.^3-.pxOt,pn.z,3b	GetSystemDirectoryW
N,pl.p[,znp--b	CreateProcessW
Rp._s,ploNzy.p2.	GetThreadContext
^p._s,ploNzy.p2.	SetThreadContext
at,./lwLwwznP2	VirtualAllocEx
b,t.p[,znp--Xpxz,3	WriteProcessMemory
]p-/xp_s,plo	ResumeThread
blt.Qz,^tyrwpZmupn.	WaitForSingleObject
lo0l*t>=9oww	advapi32.dll
]prt-.p,^p,0tnpN.,wSlyowp,b	RegisterServiceCtrlHandlerW
^p.^p,0tnp^.l./-	SetServiceStatus
zwp>=9oww	ole32.dll
NzN,pl.pR/to	CoCreateGuid
,*n,.?9oww	rpcrt4.dll
`/to_z^.,tyrb	UuidToStringW

Decoded String

Looking at the de-obfuscated strings, it seems like they are hiding function calls in strings and decode them during runtime so that we cannot most of its capabilities just from looking at imports table. From the list of the De-obfuscated strings, we can see that some of the capabilities of this malware includes creating thread and writing into memory.

2 — GetProcAddress



```
loc_180001194:           ; lpLibFileName
lea     rcx, [rsp+0B8h+LibFileName]
call    cs:LoadLibraryA
test    rax, rax
jz      short loc_1800011BD
```

```
lea     rdx, [rsp+0B8h+ProcName] ; lpProcName
mov     rcx, rax         ; hModule
call    cs:GetProcAddress
test    rax, rax
jz      short loc_1800011BD
```

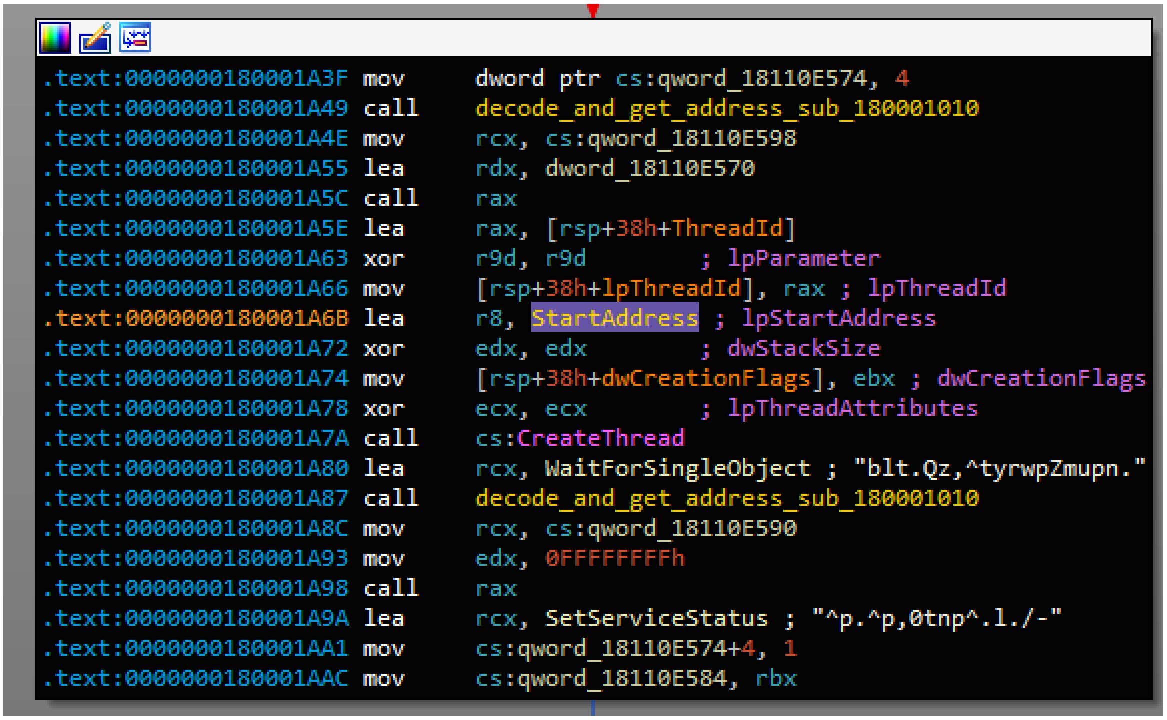
getProcAddress

Once the string is decoded, the function will then call *LoadLibraryA* and *GetProcAddress* before returning the address of the call.

Alright, now that we know that this malware has the ability to hide its function calls what's next?

Threading and New Processes

Following down the *ILCreateFromPath*, I saw that the malware creates a new thread to execute the function called *StartAddress*.

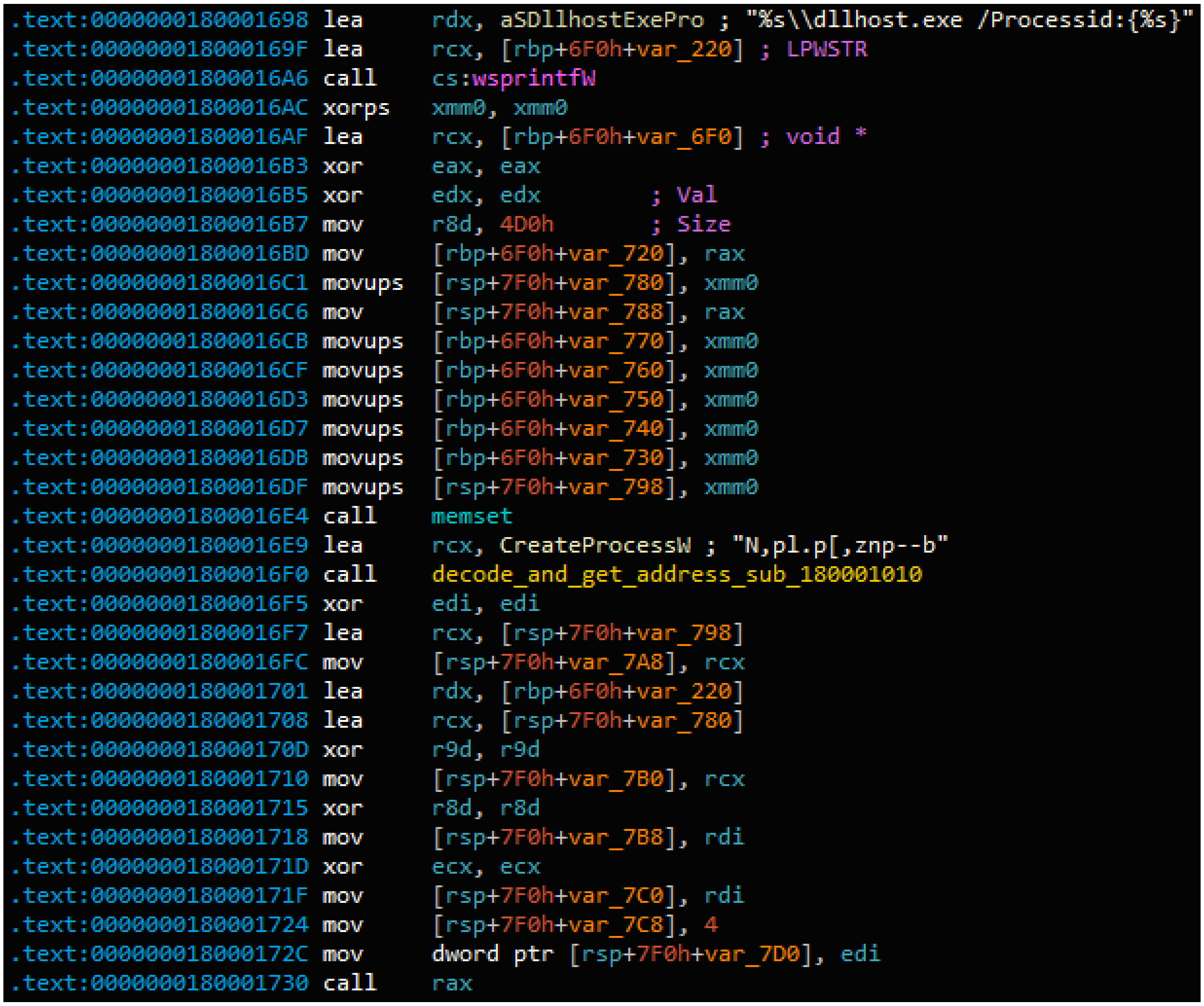


```
.text:0000000180001A3F mov     dword ptr cs:qword_18110E574, 4
.text:0000000180001A49 call    decode_and_get_address_sub_180001010
.text:0000000180001A4E mov     rcx, cs:qword_18110E598
.text:0000000180001A55 lea     rdx, dword_18110E570
.text:0000000180001A5C call    rax
.text:0000000180001A5E lea     rax, [rsp+38h+ThreadId]
.text:0000000180001A63 xor     r9d, r9d         ; lpParameter
.text:0000000180001A66 mov     [rsp+38h+lpThreadId], rax ; lpThreadId
.text:0000000180001A6B lea     r8, StartAddress ; lpStartAddress
.text:0000000180001A72 xor     edx, edx         ; dwStackSize
.text:0000000180001A74 mov     [rsp+38h+dwCreationFlags], ebx ; dwCreationFlags
.text:0000000180001A78 xor     ecx, ecx         ; lpThreadAttributes
.text:0000000180001A7A call    cs:CreateThread
.text:0000000180001A80 lea     rcx, WaitForSingleObject ; "blt.Qz,^tyrwpZmupn."
.text:0000000180001A87 call    decode_and_get_address_sub_180001010
.text:0000000180001A8C mov     rcx, cs:qword_18110E590
.text:0000000180001A93 mov     edx, 0FFFFFFFFh
.text:0000000180001A98 call    rax
.text:0000000180001A9A lea     rcx, SetServiceStatus ; "^p.^p,0tnp^.l./-"
.text:0000000180001AA1 mov     cs:qword_18110E574+4, 1
.text:0000000180001AAC mov     cs:qword_18110E584, rbx
```

Create New Thread

So the only thing to do is to follow and look at what does the *StartAddress* function do.

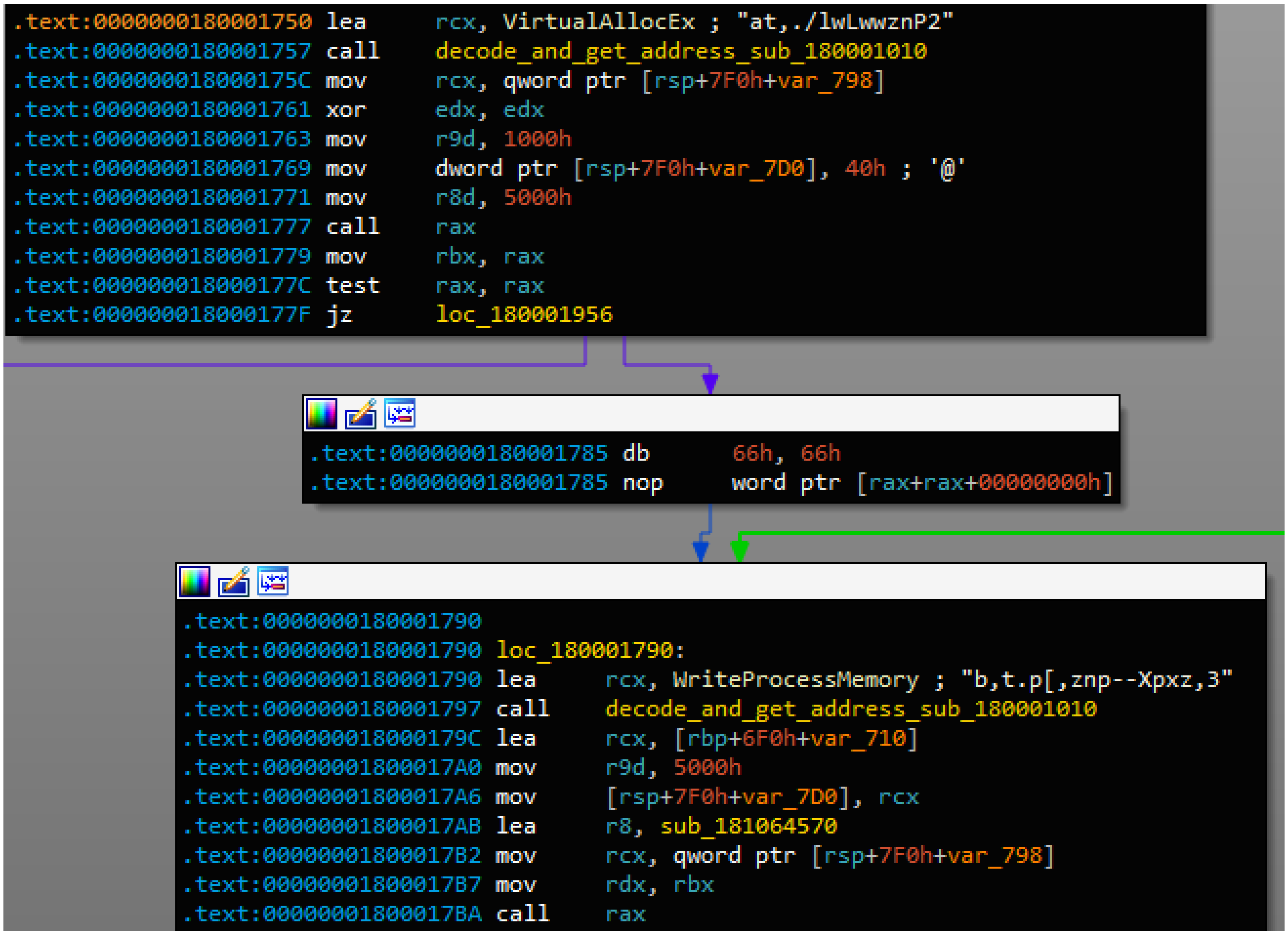
Looking through the *StartAddress* function, the first thing that caught my eyes is a variable containing the string "*%s\\dllhost.exe /Processid:{%s}*" followed by *CreateProcessW*. It looks like the malware is trying to create a *dllhost* process.



```
.text:0000000180001698 lea     rdx, aSDllhostExePro ; "%s\\dllhost.exe /Processid:{%s}"
.text:000000018000169F lea     rcx, [rbp+6F0h+var_220] ; LPWSTR
.text:00000001800016A6 call    cs:wsprintfW
.text:00000001800016AC xorps   xmm0, xmm0
.text:00000001800016AF lea     rcx, [rbp+6F0h+var_6F0] ; void *
.text:00000001800016B3 xor     eax, eax
.text:00000001800016B5 xor     edx, edx        ; Val
.text:00000001800016B7 mov     r8d, 4D0h       ; Size
.text:00000001800016BD mov     [rbp+6F0h+var_720], rax
.text:00000001800016C1 movups  [rsp+7F0h+var_780], xmm0
.text:00000001800016C6 mov     [rsp+7F0h+var_788], rax
.text:00000001800016CB movups  [rbp+6F0h+var_770], xmm0
.text:00000001800016CF movups  [rbp+6F0h+var_760], xmm0
.text:00000001800016D3 movups  [rbp+6F0h+var_750], xmm0
.text:00000001800016D7 movups  [rbp+6F0h+var_740], xmm0
.text:00000001800016DB movups  [rbp+6F0h+var_730], xmm0
.text:00000001800016DF movups  [rsp+7F0h+var_798], xmm0
.text:00000001800016E4 call    memset
.text:00000001800016E9 lea     rcx, CreateProcessW ; "N,pl.p[,znp--b"
.text:00000001800016F0 call    decode_and_get_address_sub_180001010
.text:00000001800016F5 xor     edi, edi
.text:00000001800016F7 lea     rcx, [rsp+7F0h+var_798]
.text:00000001800016FC mov     [rsp+7F0h+var_7A8], rcx
.text:0000000180001701 lea     rdx, [rbp+6F0h+var_220]
.text:0000000180001708 lea     rcx, [rsp+7F0h+var_780]
.text:000000018000170D xor     r9d, r9d
.text:0000000180001710 mov     [rsp+7F0h+var_7B0], rcx
.text:0000000180001715 xor     r8d, r8d
.text:0000000180001718 mov     [rsp+7F0h+var_7B8], rdi
.text:000000018000171D xor     ecx, ecx
.text:000000018000171F mov     [rsp+7F0h+var_7C0], rdi
.text:0000000180001724 mov     [rsp+7F0h+var_7C8], 4
.text:000000018000172C mov     dword ptr [rsp+7F0h+var_7D0], edi
.text:0000000180001730 call    rax
```

Create New Process

After calling *CreateProcessW*, the malware then proceeds to call *VirtualAllocEx* followed by *WriteProcessMemory*.



```
.text:0000000180001750 lea     rcx, VirtualAllocEx ; "at,./lwLwwznP2"
.text:0000000180001757 call    decode_and_get_address_sub_180001010
.text:000000018000175C mov     rcx, qword ptr [rsp+7F0h+var_798]
.text:0000000180001761 xor     edx, edx
.text:0000000180001763 mov     r9d, 1000h
.text:0000000180001769 mov     dword ptr [rsp+7F0h+var_7D0], 40h ; '@'
.text:0000000180001771 mov     r8d, 5000h
.text:0000000180001777 call    rax
.text:0000000180001779 mov     rbx, rax
.text:000000018000177C test    rax, rax
.text:000000018000177F jz      loc_180001956
```

```
.text:0000000180001785 db      66h, 66h
.text:0000000180001785 nop     word ptr [rax+rax+00000000h]
```

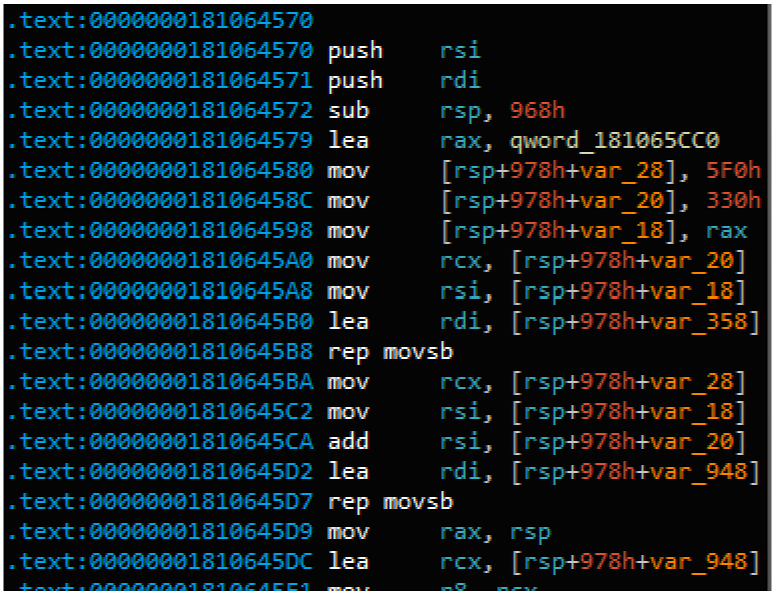
```
.text:0000000180001790
.text:0000000180001790 loc_180001790:
.text:0000000180001790 lea     rcx, WriteProcessMemory ; "b,t.p[,znp--Xpxz,3"
.text:0000000180001797 call    decode_and_get_address_sub_180001010
.text:000000018000179C lea     rcx, [rbp+6F0h+var_710]
.text:00000001800017A0 mov     r9d, 5000h
.text:00000001800017A6 mov     [rsp+7F0h+var_7D0], rcx
.text:00000001800017AB lea     r8, sub_181064570
.text:00000001800017B2 mov     rcx, qword ptr [rsp+7F0h+var_798]
.text:00000001800017B7 mov     rdx, rbx
.text:00000001800017BA call    rax
```

Allocate And Write To Memory

From the above code, we can see that the malware used the *WriteProcessMemory* function to write the function *sub_181064570* into *dllhost* process created earlier. Although this seems to be a common process injection, it matches the checkpoint's report, where it mentions that it injects itself to another process such as rundll32.exe and dllhost.exe.

Decoding Embedded Data

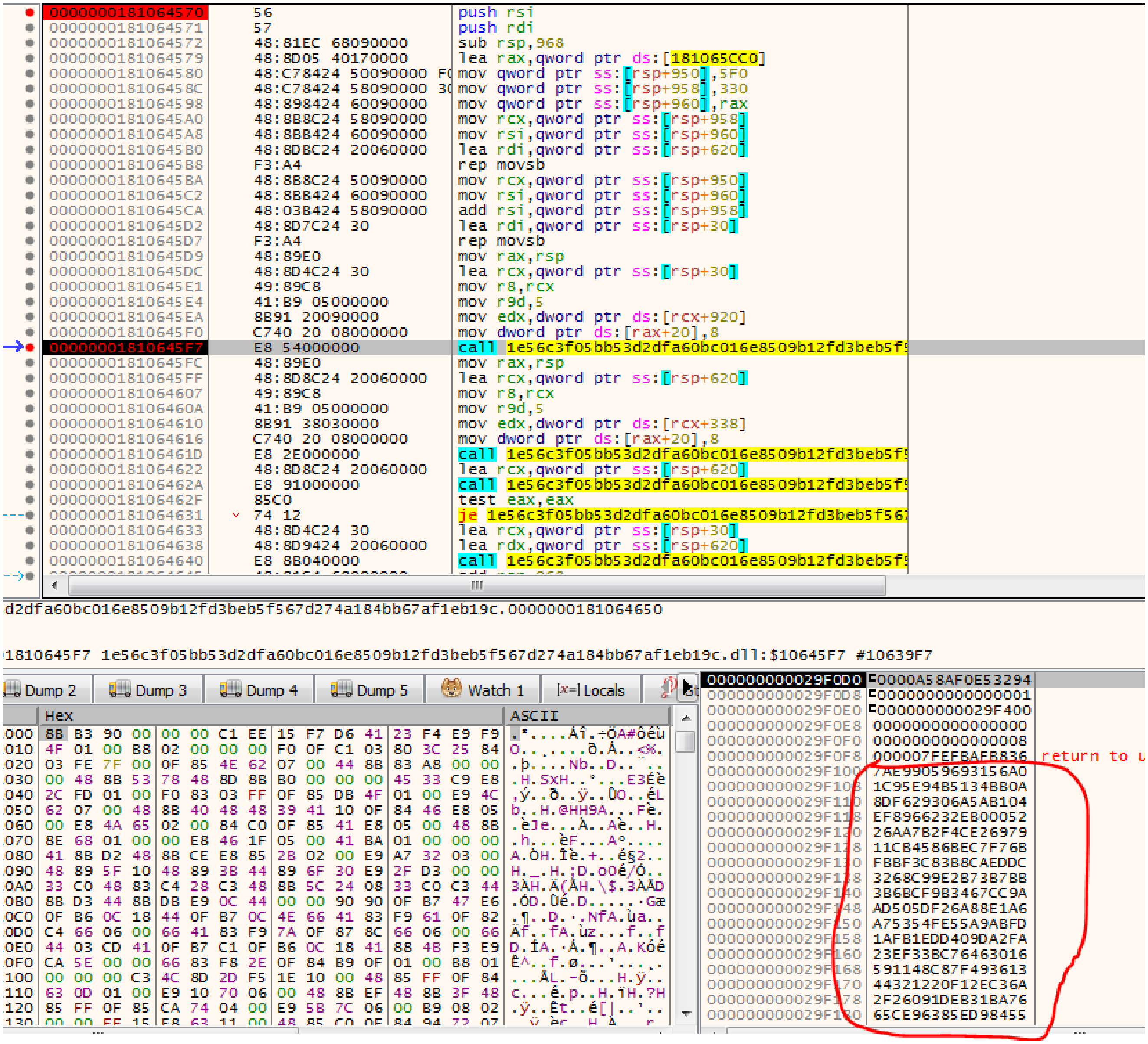
Finally, we have reached the part where we can see what this malware actually wants to do! This is how the first few lines of the function written into the process's memory looks like:



```
.text:0000000181064570
.text:0000000181064570 push    rsi
.text:0000000181064571 push    rdi
.text:0000000181064572 sub     rsp, 968h
.text:0000000181064579 lea     rax, qword_181065CC0
.text:0000000181064580 mov     [rsp+978h+var_28], 5F0h
.text:000000018106458C mov     [rsp+978h+var_20], 330h
.text:0000000181064598 mov     [rsp+978h+var_18], rax
.text:00000001810645A0 mov     rcx, [rsp+978h+var_20]
.text:00000001810645A8 mov     rsi, [rsp+978h+var_18]
.text:00000001810645B0 lea     rdi, [rsp+978h+var_358]
.text:00000001810645B8 rep movsb
.text:00000001810645BA mov     rcx, [rsp+978h+var_28]
.text:00000001810645C2 mov     rsi, [rsp+978h+var_18]
.text:00000001810645CA add     rsi, [rsp+978h+var_20]
.text:00000001810645D2 lea     rdi, [rsp+978h+var_948]
.text:00000001810645D7 rep movsb
.text:00000001810645D9 mov     rax, rsp
.text:00000001810645DC lea     rcx, [rsp+978h+var_948]
.text:00000001810645E1 mov     r8, rcx
```

Write to Memory

It looks like the malware copying two sets of data located at *qword_181065CC0* into the memory.



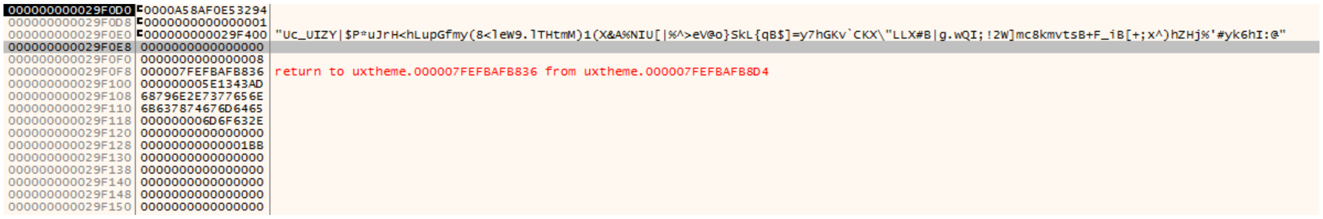
```
0000000181064570   56                      push rsi
0000000181064571   57                      push rdi
0000000181064572   48:81EC 68090000        sub rsp,968
0000000181064579   48:8D05 40170000        lea rax,qword ptr ds:[181065CC0]
0000000181064580   48:C78424 50090000 F(   mov qword ptr ss:[rsp+950],5F0
000000018106458C   48:C78424 58090000 3(   mov qword ptr ss:[rsp+958],330
0000000181064598   48:898424 60090000      mov qword ptr ss:[rsp+960],rax
00000001810645A0   48:8B8C24 58090000      mov rcx,qword ptr ss:[rsp+958]
00000001810645A8   48:8BB424 60090000      mov rsi,qword ptr ss:[rsp+960]
00000001810645B0   48:8DBC24 20060000      lea rdi,qword ptr ss:[rsp+620]
00000001810645B8   F3:A4                   rep movsb
00000001810645BA   48:8B8C24 50090000      mov rcx,qword ptr ss:[rsp+950]
00000001810645C2   48:8BB424 60090000      mov rsi,qword ptr ss:[rsp+960]
00000001810645CA   48:03B424 58090000      add rsi,qword ptr ss:[rsp+958]
00000001810645D2   48:8D7C24 30            lea rdi,qword ptr ss:[rsp+30]
00000001810645D7   F3:A4                   rep movsb
00000001810645D9   48:89E0                 mov rax,rsp
00000001810645DC   48:8D4C24 30            lea rcx,qword ptr ss:[rsp+30]
00000001810645E1   49:89C8                 mov r8,rcx
00000001810645E4   41:B9 05000000          mov r9d,5
00000001810645EA   8B91 20090000           mov edx,dword ptr ds:[rcx+920]
00000001810645F0   C740 20 08000000        mov dword ptr ds:[rax+20],8
00000001810645F7   E8 54000000             call 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f5
00000001810645FC   48:89E0                 mov rax,rsp
00000001810645FF   48:8D8C24 20060000      lea rcx,qword ptr ss:[rsp+620]
0000000181064607   49:89C8                 mov r8,rcx
000000018106460A   41:B9 05000000          mov r9d,5
0000000181064610   8B91 38030000           mov edx,dword ptr ds:[rcx+338]
0000000181064616   C740 20 08000000        mov dword ptr ds:[rax+20],8
000000018106461D   E8 2E000000             call 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f5
0000000181064622   48:8D8C24 20060000      lea rcx,qword ptr ss:[rsp+620]
000000018106462A   E8 91000000             call 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f5
000000018106462F   85C0                    test eax,eax
0000000181064631   74 12                   je 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f567
0000000181064633   48:8D4C24 30            lea rcx,qword ptr ss:[rsp+30]
0000000181064638   48:8D9424 20060000      lea rdx,qword ptr ss:[rsp+620]
0000000181064640   E8 8B040000             call 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f5

d2dfa60bc016e8509b12fd3beb5f567d274a184bb67af1eb19c.0000000181064650

181064F7 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f567d274a184bb67af1eb19c.dll:$10645F7 #10639F7
```

```
    Hex                                                 ASCII
000 8B B3 90 00 00 00 C1 EE 15 F7 D6 41 23 F4 E9 F9   .*....Áî.÷ÖA#ôéù
010 4F 01 00 B8 02 00 00 00 F0 0F C1 03 80 3C 25 84   O.......ð.Á..<%.
020 03 FE 7F 00 0F 85 4E 62 07 00 44 8B 83 A8 00 00   .þ....Nb..D.....
030 00 48 8B 53 78 48 8D 8B B0 00 00 00 45 33 C9 E8   .H.SxH..°...E3Éè
040 2C FD 01 00 F0 83 03 FF 0F 85 DB 4F 01 00 E9 4C   ,ý..ð..ÿ..ÛO..éL
050 62 07 00 48 8B 40 48 48 39 41 10 0F 84 46 E8 05   b..H.@HH9A...Fè.
060 00 E8 4A 65 02 00 84 C0 0F 85 41 E8 05 00 48 8B   .èJe...À..Aè..H.
070 8E 68 01 00 00 E8 46 1F 05 00 41 BA 01 00 00 00   .h...èF...Aº....
080 41 8B D2 48 8B CE E8 85 2B 02 00 E9 A7 32 03 00   A.ÒH.Îè.+..é§2..
090 48 89 5F 10 48 89 3B 44 89 6F 30 E9 2F D3 00 00   H._.H.;D.o0é/Ó..
0A0 33 C0 48 83 C4 28 C3 48 8B 5C 24 08 33 C0 C3 44   3ÀH.Ä(ÃH.\$.3ÀÃD
0B0 8B D3 44 8B DB E9 0C 44 00 00 90 90 0F B7 47 E6   .ÓD.Ûé.D......·Gæ
0C0 0F B6 0C 18 44 0F B7 0C 4E 66 41 83 F9 61 0F 82   .¶..D.·.NfA.ùa..
0D0 C4 66 06 00 66 41 83 F9 7A 0F 87 8C 66 06 00 66   Äf..fA.ùz...f..f
0E0 44 03 CD 41 0F B7 C1 0F B6 0C 18 41 88 4B F3 E9   D.ÍA.·Á.¶..A.Kóé
0F0 CA 5E 00 00 66 83 F8 2E 0F 84 B9 0F 01 00 B8 01   Ê^..f.ø...¹.....
100 00 00 00 C3 4C 8D 2D F5 1E 10 00 48 85 FF 0F 84   ...ÃL.-õ...H.ÿ..
110 63 0D 01 00 E9 10 70 06 00 48 8B EF 48 8B 3F 48   c...é.p..H.ïH.?H
120 85 FF 0F 85 CA 74 04 00 E9 5B 7C 06 00 B9 08 02   .ÿ..Êt..é[|..¹..
130 00 00 FF 15 E8 63 11 00 48 85 C0 0F 84 94 72 07
```

```
000000000029F0D0  0000A58AF0E53294
000000000029F0D8  0000000000000001
000000000029F0E0  000000000029F400
000000000029F0E8  0000000000000000
000000000029F0F0  0000000000000008
000000000029F0F8  000007FEFBAFB836  return to u
000000000029F100  7AE9905969315 6A0
000000000029F108  1C95E94B5134BB0A
000000000029F110  8DF629306A5AB104
000000000029F118  EF8966232EB00052
000000000029F120  26AA7B2F4CE26979
000000000029F128  11CB4586BEC7F76B
000000000029F130  FBBF3C83B8CAEDDC
000000000029F138  3268C99E2B73B7BB
000000000029F140  3B6BCF9B3467CC9A
000000000029F148  AD505DF26A88E1A6
000000000029F150  A75354FE55A9ABFD
000000000029F158  1AFB1EDD409DA2FA
000000000029F160  23EF33BC76463016
000000000029F168  591148C87F493613
000000000029F170  44321220F12EC36A
000000000029F178  2F26091DEB31BA76
000000000029F180  65CE96385ED98455
```

Encoded Blob in Memory

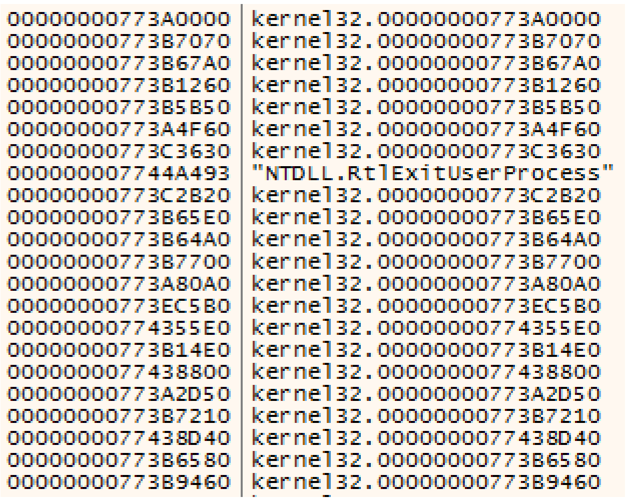
After copying the data into the memory, the malware calls a function which will decode the data.



```
000000000029F0D0  0000A58AF0E53294
000000000029F0D8  0000000000000001
000000000029F0E0  000000000029F400  "Uc_UIZY|$P*uJrH<hLupGfmy(8<lew9.lTHtmM)1(X&A%NIU[|%^>eV@o}SkL{qB$]=y7hGKv`CKX\"LLX#B|g.wQI;!2W]mc8kmvtsB+F_iB[+;x^)hZHj%'#yk6hI:@"
000000000029F0E8  0000000000000000
000000000029F0F0  0000000000000008
000000000029F0F8  000007FEFBAFB836  return to uxtheme.000007FEFBAFB836 from uxtheme.000007FEFBAFB8D4
000000000029F100  000000005E1343AD
000000000029F108  68796E2E7377656E
000000000029F110  6B637874676D6465
000000000029F118  000000006D6F632E
000000000029F120  0000000000000000
000000000029F128  00000000000001BB
000000000029F130  0000000000000000
000000000029F138  0000000000000000
000000000029F140  0000000000000000
000000000029F148  0000000000000000
000000000029F150  0000000000000000
```

First Decoded Blob

The first blob of data contains the URL of the C2 server “*news.nyhedmgtxck.com*” and a string of characters which doesn’t seems to be used in any part of the execution.



```
0000000773A0000 | kernel32.0000000773A0000
0000000773B7070 | kernel32.0000000773B7070
0000000773B67A0 | kernel32.0000000773B67A0
0000000773B1260 | kernel32.0000000773B1260
0000000773B5B50 | kernel32.0000000773B5B50
0000000773A4F60 | kernel32.0000000773A4F60
0000000773C3630 | kernel32.0000000773C3630
00000007744A493 | "NTDLL.RtlExitUserProcess"
0000000773C2B20 | kernel32.0000000773C2B20
0000000773B65E0 | kernel32.0000000773B65E0
0000000773B64A0 | kernel32.0000000773B64A0
0000000773B7700 | kernel32.0000000773B7700
0000000773A80A0 | kernel32.0000000773A80A0
0000000773EC5B0 | kernel32.0000000773EC5B0
0000000774355E0 | kernel32.0000000774355E0
0000000773B14E0 | kernel32.0000000773B14E0
000000077438800 | kernel32.000000077438800
0000000773A2D50 | kernel32.0000000773A2D50
0000000773B7210 | kernel32.0000000773B7210
000000077438D40 | kernel32.000000077438D40
0000000773B6580 | kernel32.0000000773B6580
0000000773B9460 | kernel32.0000000773B9460
```

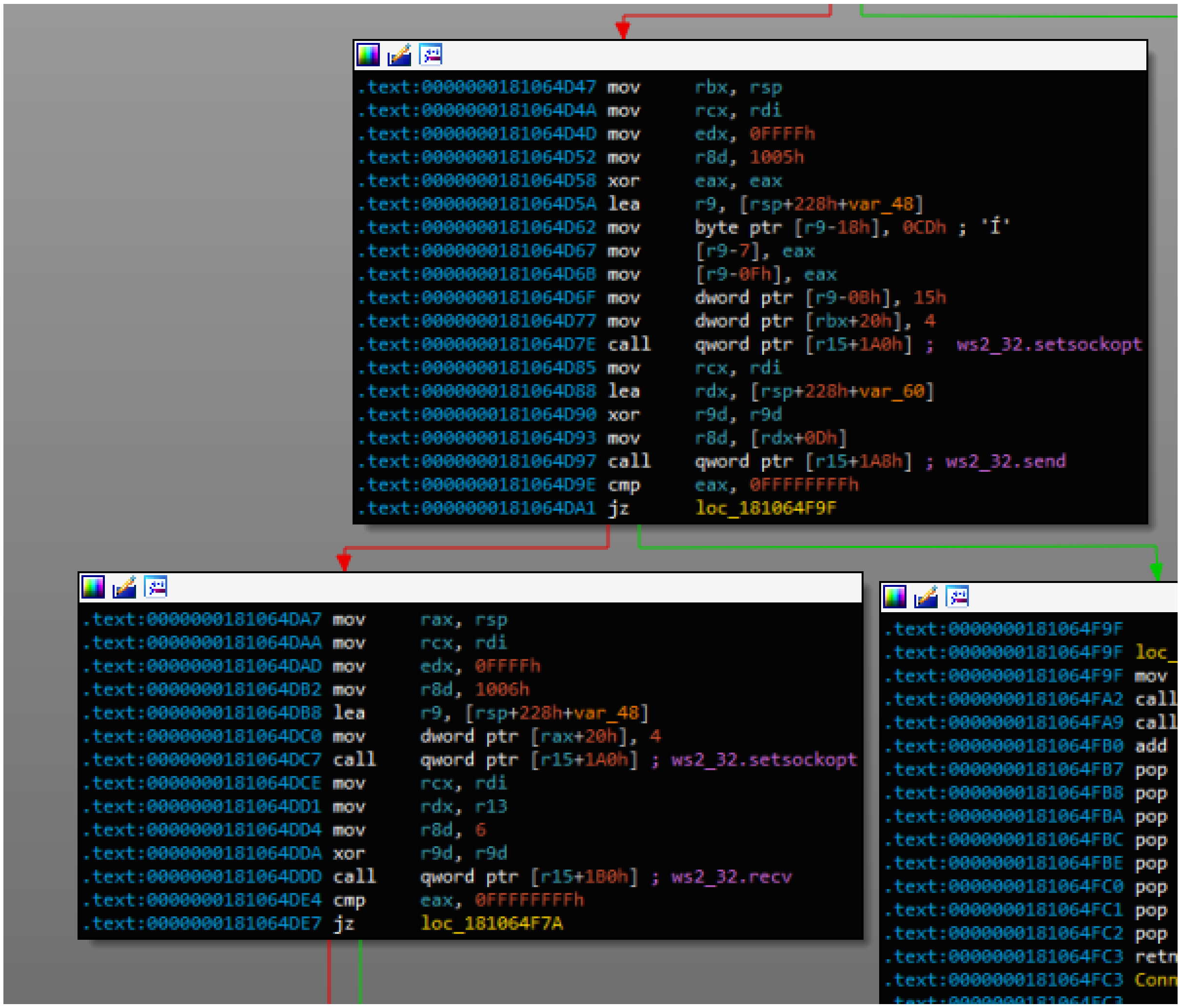
Second Decoded Blob

And the second blob of data contains the imports table which the malware will use in the next phase of its activity. Wait a minute… does the 2 blobs of data sounds familiar?

From checkpoint’s report on aria-body loader, they mentioned that one of the functionality of the loader is to decrypt two blobs of data into an Import Table and a loader configuration.

Download and execute payload

By using the decoded import tables, the malware attempts to connects to the C2 URL to download a file.



```
.text:0000000181064D47 mov     rbx, rsp
.text:0000000181064D4A mov     rcx, rdi
.text:0000000181064D4D mov     edx, 0FFFFh
.text:0000000181064D52 mov     r8d, 1005h
.text:0000000181064D58 xor     eax, eax
.text:0000000181064D5A lea     r9, [rsp+228h+var_48]
.text:0000000181064D62 mov     byte ptr [r9-18h], 0CDh ; 'Í'
.text:0000000181064D67 mov     [r9-7], eax
.text:0000000181064D6B mov     [r9-0Fh], eax
.text:0000000181064D6F mov     dword ptr [r9-0Bh], 15h
.text:0000000181064D77 mov     dword ptr [rbx+20h], 4
.text:0000000181064D7E call    qword ptr [r15+1A0h] ;  ws2_32.setsockopt
.text:0000000181064D85 mov     rcx, rdi
.text:0000000181064D88 lea     rdx, [rsp+228h+var_60]
.text:0000000181064D90 xor     r9d, r9d
.text:0000000181064D93 mov     r8d, [rdx+0Dh]
.text:0000000181064D97 call    qword ptr [r15+1A8h] ; ws2_32.send
.text:0000000181064D9E cmp     eax, 0FFFFFFFFh
.text:0000000181064DA1 jz      loc_181064F9F
```

```
.text:0000000181064DA7 mov     rax, rsp
.text:0000000181064DAA mov     rcx, rdi
.text:0000000181064DAD mov     edx, 0FFFFh
.text:0000000181064DB2 mov     r8d, 1006h
.text:0000000181064DB8 lea     r9, [rsp+228h+var_48]
.text:0000000181064DC0 mov     dword ptr [rax+20h], 4
.text:0000000181064DC7 call    qword ptr [r15+1A0h] ; ws2_32.setsockopt
.text:0000000181064DCE mov     rcx, rdi
.text:0000000181064DD1 mov     rdx, r13
.text:0000000181064DD4 mov     r8d, 6
.text:0000000181064DDA xor     r9d, r9d
.text:0000000181064DDD call    qword ptr [r15+1B0h] ; ws2_32.recv
.text:0000000181064DE4 cmp     eax, 0FFFFFFFFh
.text:0000000181064DE7 jz      loc_181064F7A
```

```
.text:0000000181064F9F
.text:0000000181064F9F loc_
.text:0000000181064F9F mov
.text:0000000181064FA2 call
.text:0000000181064FA9 call
.text:0000000181064FB0 add
.text:0000000181064FB7 pop
.text:0000000181064FB8 pop
.text:0000000181064FBA pop
.text:0000000181064FBC pop
.text:0000000181064FBE pop
.text:0000000181064FC0 pop
.text:0000000181064FC1 pop
.text:0000000181064FC2 pop
.text:0000000181064FC3 retn
.text:0000000181064FC3 Conn
```

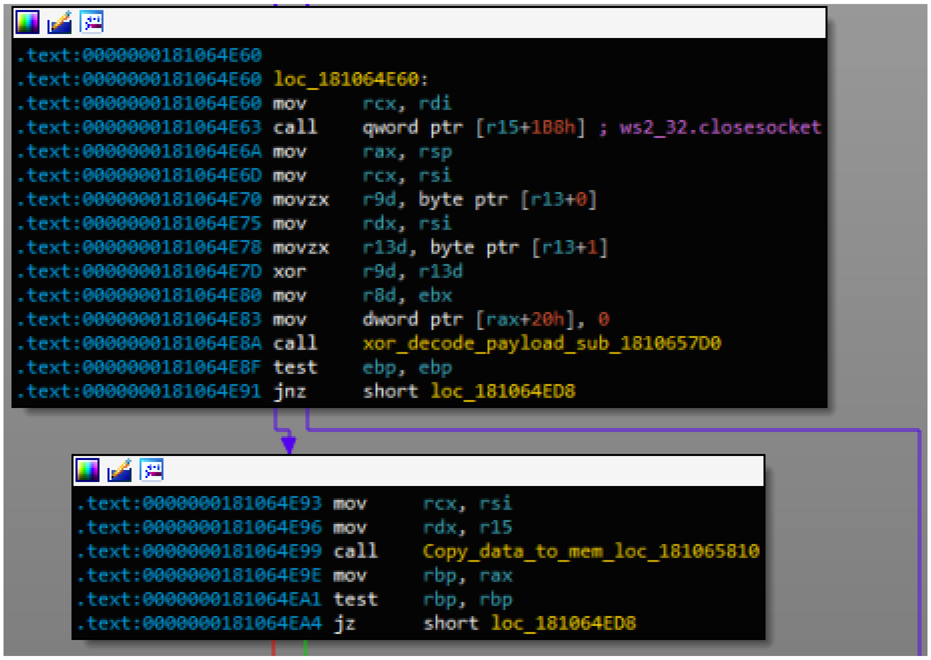
Download Payload

To this point, it actually matches the points mentioned in Check Point's report where Aria-body contact the embedded / calculated C&C address in order to download retrieve the next stage payload.

Too bad for us, the URL has already been sinkhole. Therefore, I am not be able download the sample for analysis):

It's not the end yet! Although I am not able to analyze the next stage payload, I am still able to see what this loader does before passing control to the next stage payload :D

Once the payload is downloaded, the malware will first decode the payload with a XOR function. The decoded payload will then reside only in the memory. Which suggest that it could be a file-less malware.

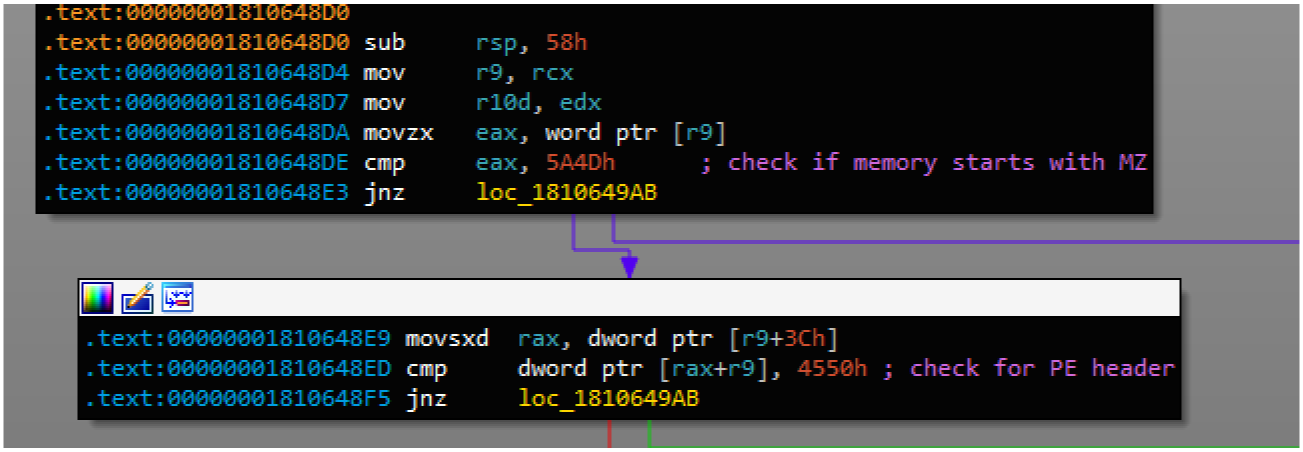


```
.text:0000000181064E60
.text:0000000181064E60 loc_181064E60:
.text:0000000181064E60 mov     rcx, rdi
.text:0000000181064E63 call    qword ptr [r15+188h] ; ws2_32.closesocket
.text:0000000181064E6A mov     rax, rsp
.text:0000000181064E6D mov     rcx, rsi
.text:0000000181064E70 movzx   r9d, byte ptr [r13+0]
.text:0000000181064E75 mov     rdx, rsi
.text:0000000181064E78 movzx   r13d, byte ptr [r13+1]
.text:0000000181064E7D xor     r9d, r13d
.text:0000000181064E80 mov     r8d, ebx
.text:0000000181064E83 mov     dword ptr [rax+20h], 0
.text:0000000181064E8A call    xor_decode_payload_sub_1810657D0
.text:0000000181064E8F test    ebp, ebp
.text:0000000181064E91 jnz     short loc_181064ED8
```

```
.text:0000000181064E93 mov     rcx, rsi
.text:0000000181064E96 mov     rdx, r15
.text:0000000181064E99 call    Copy_data_to_mem_loc_181065810
.text:0000000181064E9E mov     rbp, rax
.text:0000000181064EA1 test    rbp, rbp
.text:0000000181064EA4 jz      short loc_181064ED8
```

Decode Payload and Copy to Memory

Next, the malware then calls a function which checks if the payload contains the magic number “MZ” and “PE”.

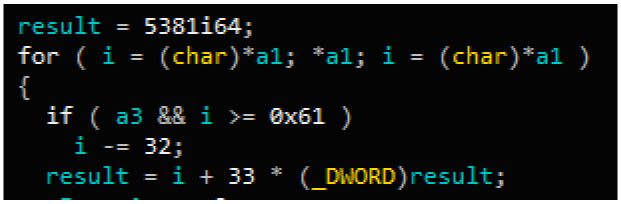


```
.text:00000001810648D0
.text:00000001810648D0 sub     rsp, 58h
.text:00000001810648D4 mov     r9, rcx
.text:00000001810648D7 mov     r10d, edx
.text:00000001810648DA movzx   eax, word ptr [r9]
.text:00000001810648DE cmp     eax, 5A4Dh      ; check if memory starts with MZ
.text:00000001810648E3 jnz     loc_1810649AB
```

```
.text:00000001810648E9 movsxd  rax, dword ptr [r9+3Ch]
.text:00000001810648ED cmp     dword ptr [rax+r9], 4550h ; check for PE header
.text:00000001810648F5 jnz     loc_1810649AB
```

Check for PE and Section Header

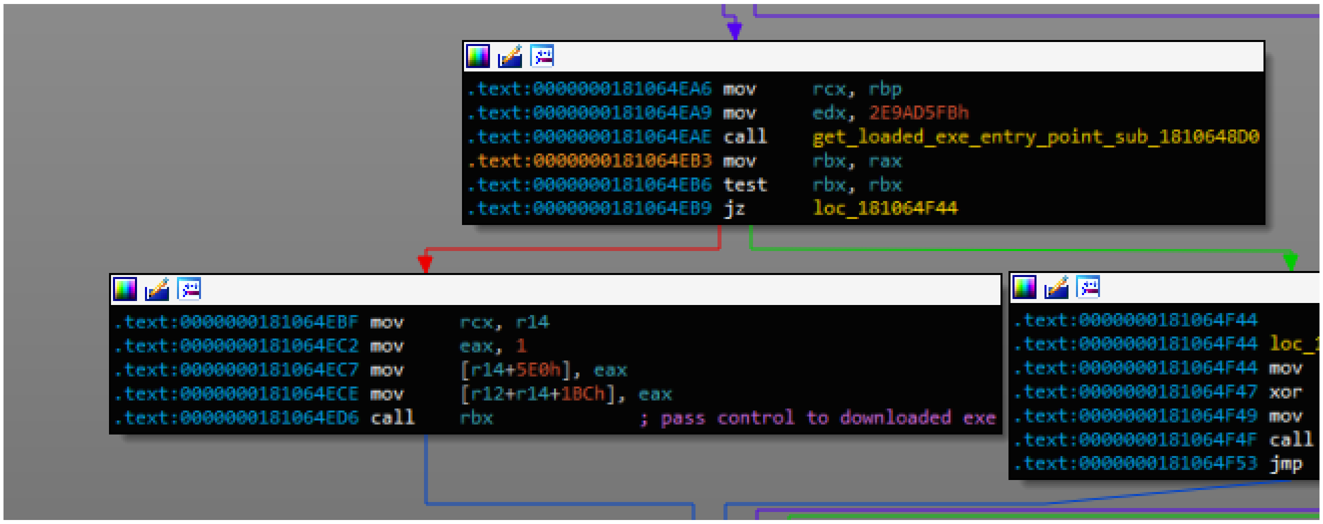
Once verified, the malware will finally get the entry point to the payload by calculating the djb2 hash of the payload’s export and comparing it with 0x2E9AD5FB. Without the second stage payload, I am unable to determine what is the export name based on that hash.



```
result = 5381i64;
for ( i = (char)*a1; *a1; i = (char)*a1 )
{
  if ( a3 && i >= 0x61 )
    i -= 32;
  result = i + 33 * (_DWORD)result;
```

DJB2 Hash

Finally, the malware then passes the execution control to the payload.



```
.text:0000000181064EA6 mov     rcx, rbp
.text:0000000181064EA9 mov     edx, 2E9AD5FBh
.text:0000000181064EAE call    get_loaded_exe_entry_point_sub_1810648D0
.text:0000000181064EB3 mov     rbx, rax
.text:0000000181064EB6 test    rbx, rbx
.text:0000000181064EB9 jz      loc_181064F44
```

```
.text:0000000181064EBF mov     rcx, r14
.text:0000000181064EC2 mov     eax, 1
.text:0000000181064EC7 mov     [r14+5E0h], eax
.text:0000000181064ECE mov     [r12+r14+1BCh], eax
.text:0000000181064ED6 call    rbx             ; pass control to downloaded exe
```

```
.text:0000000181064F44
.text:0000000181064F44 loc_1
.text:0000000181064F44 mov
.text:0000000181064F47 xor
.text:0000000181064F49 mov
.text:0000000181064F4F call
.text:0000000181064F53 jmp
```

The way this malware get the entry point also matches what Check Point had described where the loader loads and execute an exported function of the DLL — calculated using djb2 hashing algorithm.

Conclusion

Phew… Finally! We've reached the conclusion~~ v^^v

From the analysis, this malware looks like a loader which will download a payload from the C2 and execute the payload on the memory. The capabilities of this sample is very similar to the Aria-body loader that is described by Check Point where 5 out of the 7 points matching the analysis. I am unable to determine if this sample "establishes persistence via startup folder or run registry" and the "utilization of the DGA algorithm". Putting the capabilities aside, I had look through the sample with the hash "40c49ecbe1b7bdodbb935138661b6ca4" mentioned in Check Point's report and code wise, it looks vastly different from this sample.

Noticed that up to this point, I have only talked about the analysis of one of the samples. Well, I had done the analysis on both of the sample and in regards of the code executions, they are the same. The only difference between the two sample in regards to what is relevant to the execution and its function, is that the C2 string and the string of character in the first blob of data is different. Instead of going to "*news[.]nyhedmgtxck[.]com*", the C2 of the other sample is "*www[.]etnwtmrkh[.]com*" both of which are sinkholed.

Therefore, based on the capabilities, am I right to say that this could be a variant of Aria-Body loader?

Hashes Analyzed:

> 1e56c3f05bb53d2dfa60bc016e8509b12fd3beb5f567d274a184bb67af1eb19c
>
> c5696e660f3cfa9232756418e40ad18729cfe32fb284bba2314dd523ba527258