MalwareAnalysisReports/WikiLoader/WikiLoader notepad.md at main · VenzoV/MalwareAnalysisReports

By VenzoV

Archived: 2026-04-05 19:26:23 UTC

Sample Information

I stumbled upon this sample checking the following post:

Cryptolaemus
@Cryptolaemus1

#WikiLoader - #TA544 - pdf > url > .zip > .js > .js > .dll

wscript.exe DSV 101.js

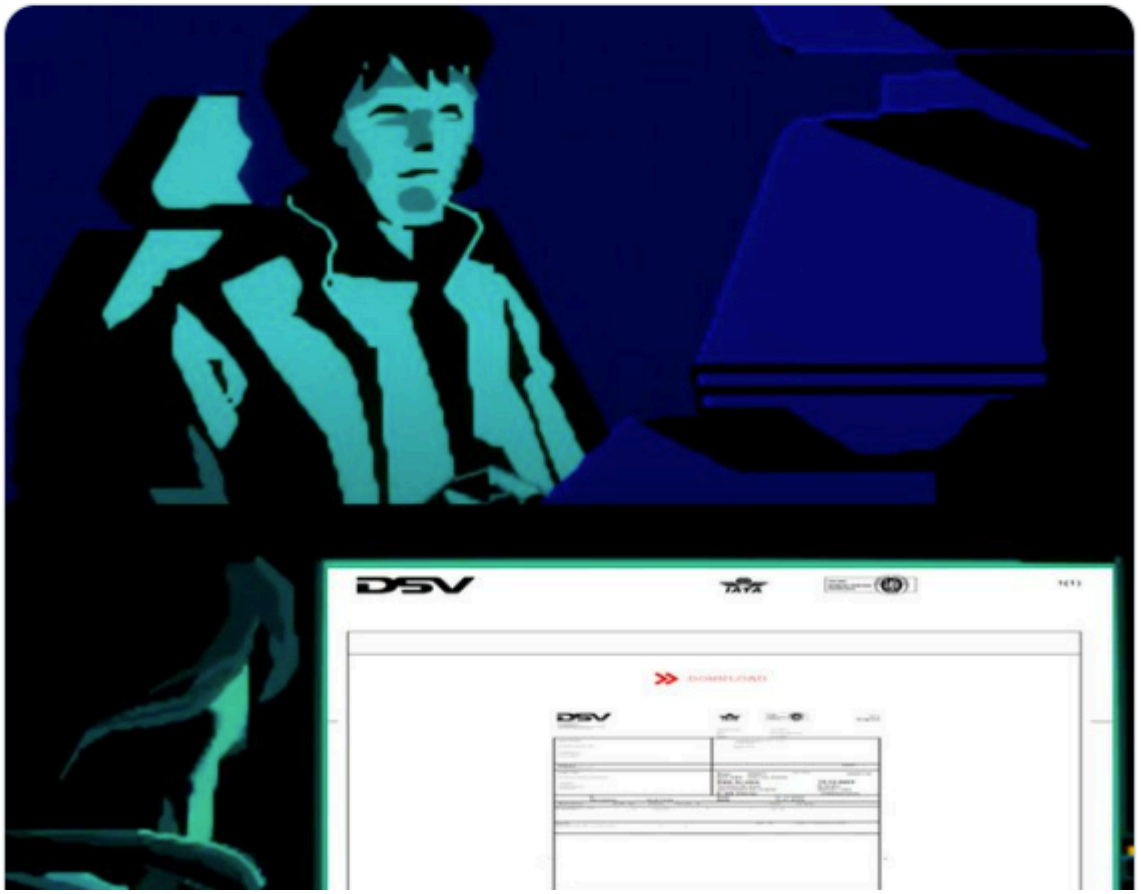
wscript.exe out.js

C:\Users\User\AppData\Local\Temp\npp.8.6.portable.x64\notepad.exe (sideload) 👇

\npp.8.6.portable.x64\plugins\mimeTools.dll

(1/3) 👇

IOC's
github.com/pr0xylife/Wiki...



The following hash is for the malicious .dll "MimeTools.dll"

SHA256
67283e154b86612e325030e5a5f7995a6fe552d20655283ea5de8b53ff405f69

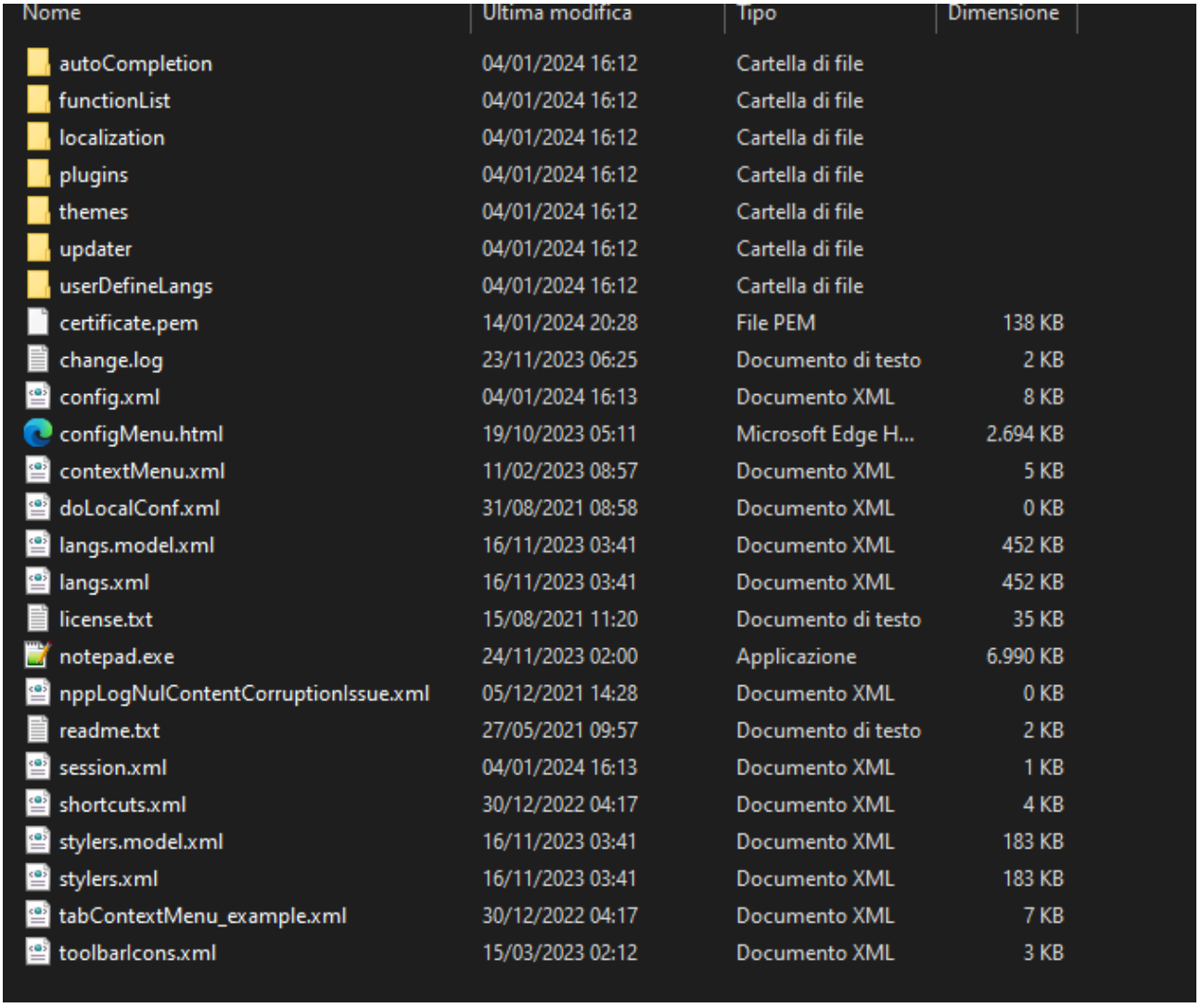
Following the hashes for the .zip file which contains the .dll.

SHA256
bef04e3b2b81f2dee39c42ab9be781f3db0059ec722aeee3b5434c2e63512a68

ZIP file contents and "notepad"

The zip file initially contains files that mimic the notepad++ file structure and files. The end game is for the "notepad.exe" to side load the malicious .dll "mimetools.dll"

Malicious zip file contents:



Nome	Ultima modifica	Tipo	Dimensione
autoCompletion	04/01/2024 16:12	Cartella di file	
functionList	04/01/2024 16:12	Cartella di file	
localization	04/01/2024 16:12	Cartella di file	
plugins	04/01/2024 16:12	Cartella di file	
themes	04/01/2024 16:12	Cartella di file	
updater	04/01/2024 16:12	Cartella di file	
userDefineLangs	04/01/2024 16:12	Cartella di file	
certificate.pem	14/01/2024 20:28	File PEM	138 KB
change.log	23/11/2023 06:25	Documento di testo	2 KB
config.xml	04/01/2024 16:13	Documento XML	8 KB
configMenu.html	19/10/2023 05:11	Microsoft Edge H...	2.694 KB
contextMenu.xml	11/02/2023 08:57	Documento XML	5 KB
doLocalConf.xml	31/08/2021 08:58	Documento XML	0 KB
langs.model.xml	16/11/2023 03:41	Documento XML	452 KB
langs.xml	16/11/2023 03:41	Documento XML	452 KB
license.txt	15/08/2021 11:20	Documento di testo	35 KB
notepad.exe	24/11/2023 02:00	Applicazione	6.990 KB
nppLogNulContentCorruptionIssue.xml	05/12/2021 14:28	Documento XML	0 KB
readme.txt	27/05/2021 09:57	Documento di testo	2 KB
session.xml	04/01/2024 16:13	Documento XML	1 KB
shortcuts.xml	30/12/2022 04:17	Documento XML	4 KB
stylers.model.xml	16/11/2023 03:41	Documento XML	183 KB
stylers.xml	16/11/2023 03:41	Documento XML	183 KB
tabContextMenu_example.xml	30/12/2022 04:17	Documento XML	7 KB
toolbarIcons.xml	15/03/2023 02:12	Documento XML	3 KB

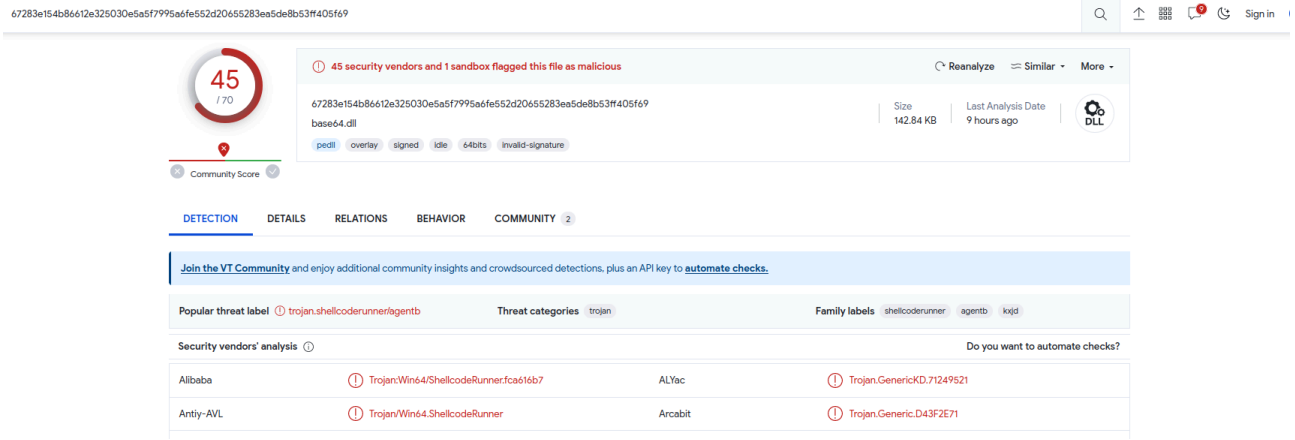
Legitimate notepade++.exe folder structure.

autoCompletion	27/11/2022 15:58	Cartella di file	
functionList	27/11/2022 15:58	Cartella di file	
localization	27/11/2022 15:58	Cartella di file	
plugins	27/11/2022 15:58	Cartella di file	
themes	27/11/2022 15:58	Cartella di file	
updater	27/11/2022 15:58	Cartella di file	
change.log	08/11/2022 12:42	Documento di testo	2 KB
contextMenu.xml	17/01/2022 01:43	Documento XML	5 KB
langs.model.xml	31/10/2022 19:34	Documento XML	343 KB
LICENSE	15/08/2021 02:20	File	35 KB
notepad++.exe	08/11/2022 13:04	Applicazione	6.258 KB
nppLogNulContentCorruptionIssue.xml	05/12/2021 05:28	Documento XML	0 KB
NppShell_06.dll	08/11/2022 13:04	Estensione dell'ap...	228 KB
readme.txt	27/05/2021 00:57	Documento di testo	2 KB
shortcuts.xml	27/05/2021 00:57	Documento XML	1 KB
stylers.model.xml	31/10/2022 01:57	Documento XML	172 KB
uninstall.exe	27/11/2022 15:58	Applicazione	262 KB

The malicious .dll is located also in the same path notepad++ contains the file. This is of course so that the sideloading can work. The path for the .dll:

- plugins\mimeTools\

Comparing hashes on VT or any other site like unpac.me, we can observe that they are different and one of them is also detected by many engines.



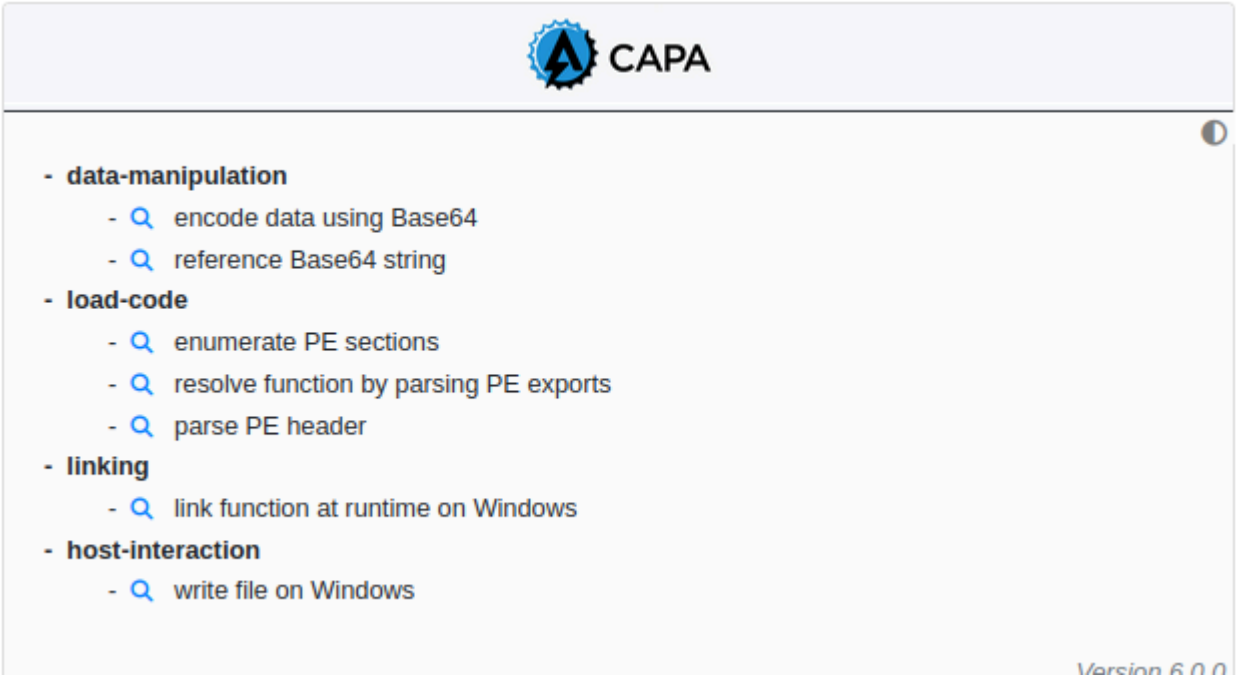
Static Analysis

Using IDA or any decompiler won't really help initially. The sample uses a lot of control flow obfuscation essentially changing calls for jmp instructions. This will confuse IDA and we don't get clear disassembly view nor decompiler. So, for the moment we will go with dynamic analysis with x64 dbg.

Information Gathering

Some information based on OSINT and other resources just to get ahead and have some hits on what to look for. I was able to find only one report from proofpoint with some useful information. Also online sandboxes gives us some hints of behaviors to expect.

We will expect PE parsing to occur:



I also observed from IDA some potential stack strings, I ran FLOSS to try and extract some. Following the results, which we will user to help our analysis. Although it may not be useful in the end.

```
 FLOSS Stack Strings

FLOSS decoded 1 strings
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

FLOSS extracted 67 stackstrings
nXm21XOk}Z
nXm21XOk}Z
VirtualAlloc
CryptStringToBinaryA
CryptStringToBinaryA
ex28mBHjNU7
ex28mBHjNU7
CryptStringToBinaryA
8mBHjNU7
HjGwPX65nXm21XOkX4nMqex28mBHjNU7
ex28mBHjNU7
LoadLibraryA
ZBAA
Virtu
```

```
VirtualAlloc
nXm21XOk}Z
VirtualAlloc
VirtualAlloc
8mBHjNU7
VirtualFreeA
VirtualFreeA
8mBHjNU7
VirtualFreeA
VirtualAlloc
ex28mBHjNU7
HjGwPX65nXm21XOkX4nMqex28mBHjNU7
LoadLibraryA
nXm21XOk}Z
VirtualFreeA
HjGwPX65nXm21XOkX4nMqex28mBHjNU7
HjGwPX65nXm21XOkX4nMqex28mBHjNU7
LoadLibraryA
CryptStringToBinaryA
LoadLibraryA
8mBHjNU7
Virtu
LoadLibraryA
VirtualFreeA
HjGwPX65nXm21XOkX4nMqex28mBHjNU7
nXm21XOk}Z
VirtualFreeA
Virtu
8mBHjNU7
Virtu
ex28mBHjNU7
CryptStringToBinaryA
LoadLibraryA
LoadLibraryA
8mBHjNU7
LoadLibraryA
HjGwPX65nXm21XOkX4nMqex28mBHjNU7
8mBHjNU7
VirtualAlloc
ex28mBHjNU7
CryptStringToBinaryA
ex28mBHjNU7
nXm21XOk}Z
HjGwPX65nXm21XOkX4nMqex28mBHjNU7
nXm21XOk}Z
CryptStringToBinaryA
VirtualAlloc
```

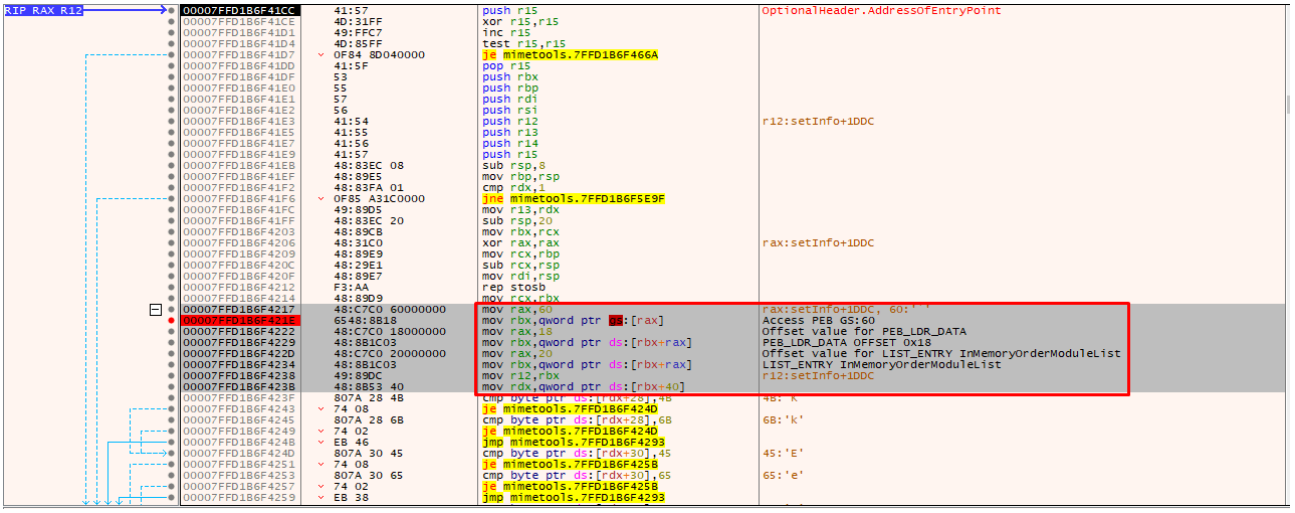
```
Virtu
VirtualFreeA
CreateThread
VirtualFreeA
LoadLibraryA
VirtualFreeA
```

Dynamic Analysis

Getting Kernel32.dll & GetProcAddress

First thing the malware gets a handle to the PEB from the TIB, and then sets up relevant offsets to access the LIST_ENTRY InMemoryOrderModuleList structure. As per microsoft docs it is a double linked list that contains the loaded modules for the process. This will be used the malware to go though the dlls loaded. Ntdll.dll is the first .dll of the process, followed by kernel32.dll.

- Offset 0x60 is passed to RAX regiser
- RAX is used with gs: special register to obtain the PEB location.
- Offset 0x18 is passed to RBX
- Offset RBX+RAX is used to access PEB_LDR_DATA struct and saved to RBX
- Offset 0x20 is passed to RAX
- Offset RBX+RAX now leads to LIST_ENTRY InMemoryOrderModuleList
- The first of the list InMemoryOrderModuleList is now saved in r12 register, this will be used to check for end of list. Since it is a doubled link list, if the code goes through all the modules, it will eventually end up at the start.
- Lastly it gets the 0x40 offset to add to the InMemoryOrderModuleList. This is for the Dllbase address.



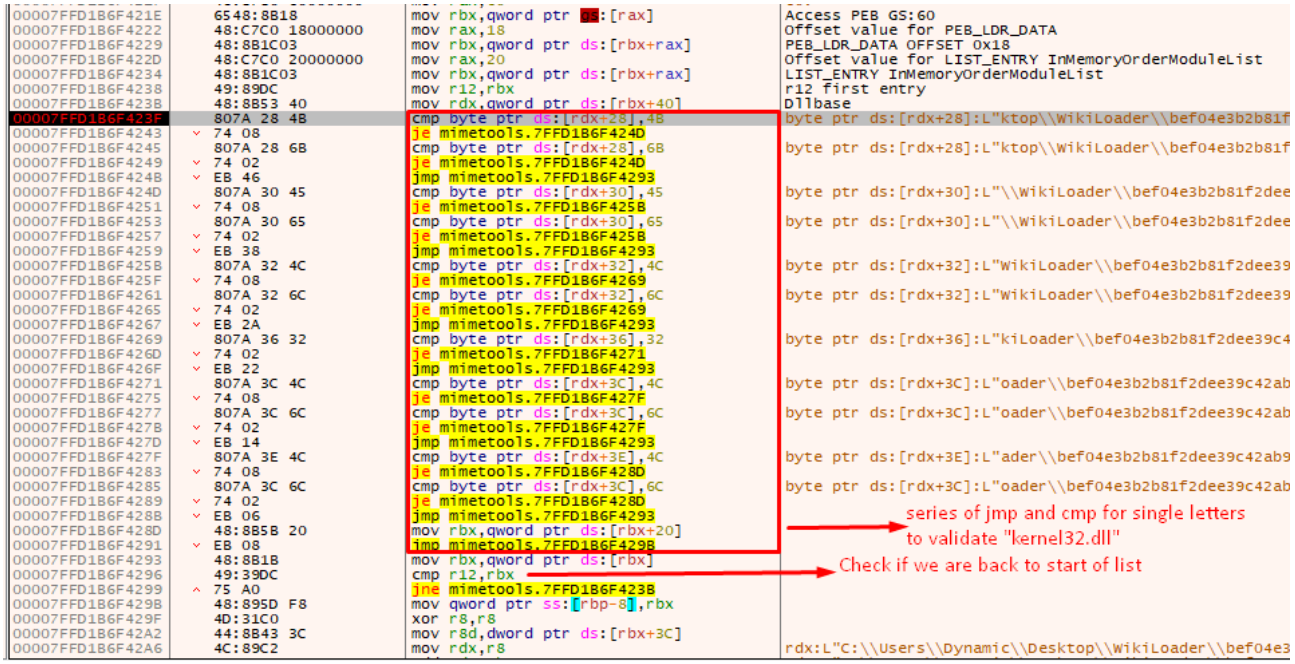
The objective of this first loop is to find kernel32.dll. By walking the PEB, we expect to find the following module order:

- Executing Process
- ntdll.dll

- kernel32.dll

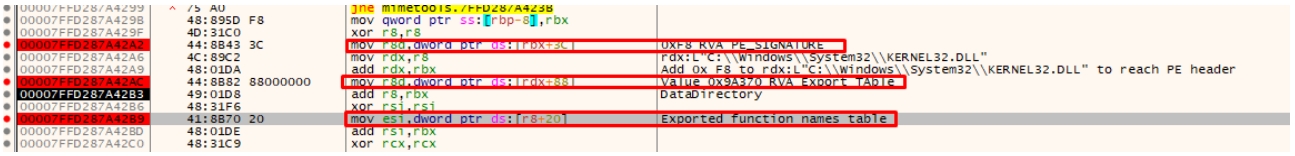
So, the code will have to go through 3 forward pointers. The way the malware checks if it has the right dll reference is through series of jumps. A certain offset is added to the Dllbase address, this will be a letter in the name, then it is compared to a letter. The malware will check these letters for the kernel32.dll

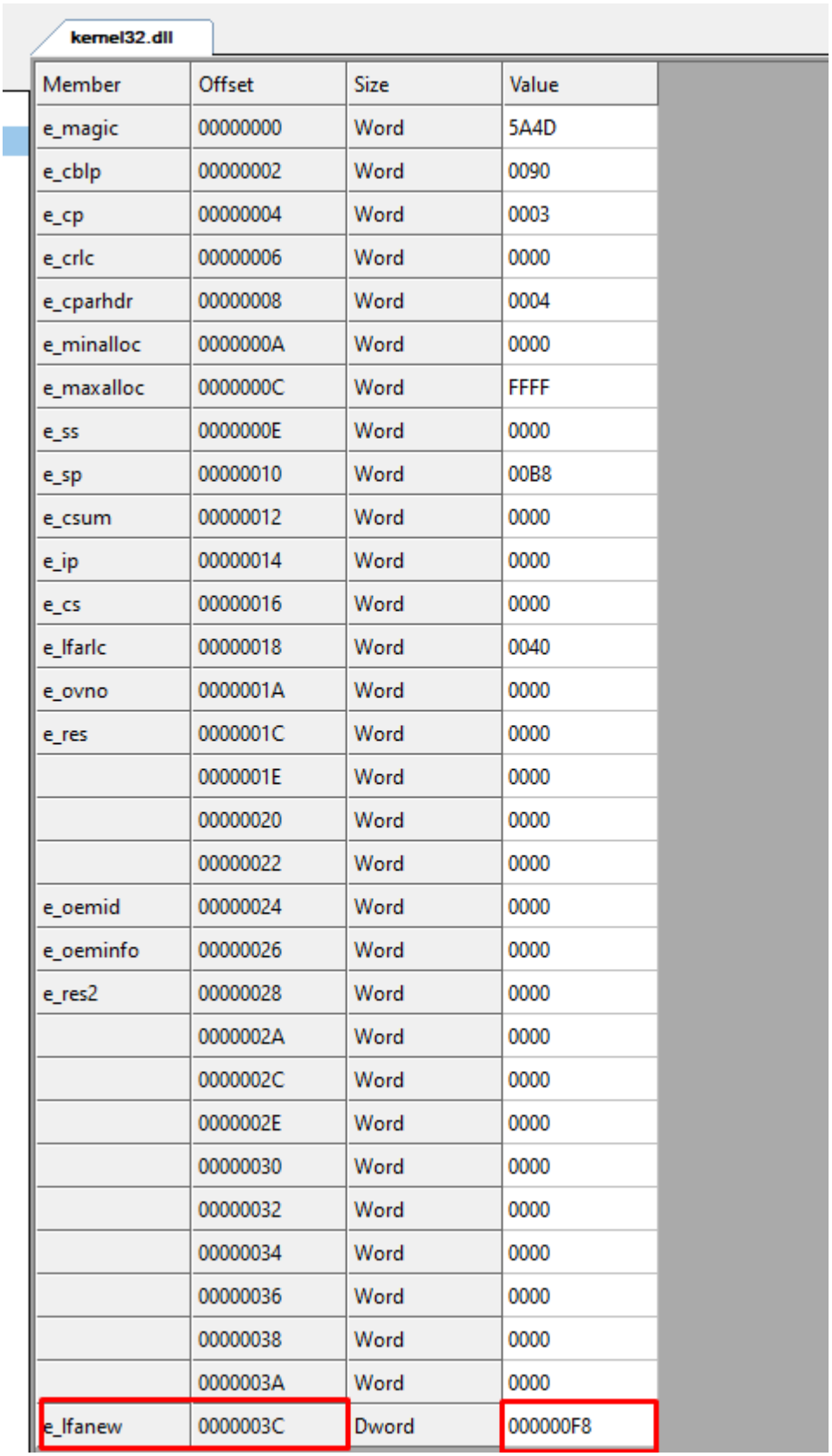
- 4b->'K'
- 45->'E'
- 4c->'L'
- 32->'2'
- 4c->'L'
- 4c->'L'



Now that the malware has the base address to Kernel32.dll, it will go through all the functions to search for GetProcAddress. It parses the PE header of kernel32.dll to reach the export table.

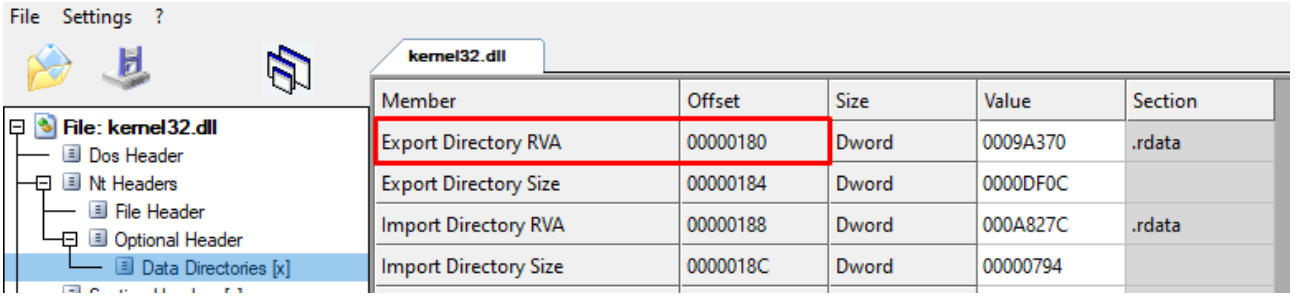
1. Offset 3C to reach PE header pointer with value F8.
2. Offset 88 to reach the export table, this value is calculated with: RVA of export table - F8. In our case 0x180-F8=0x88
3. Offset 0x20 to finally reach the exported functions names table.





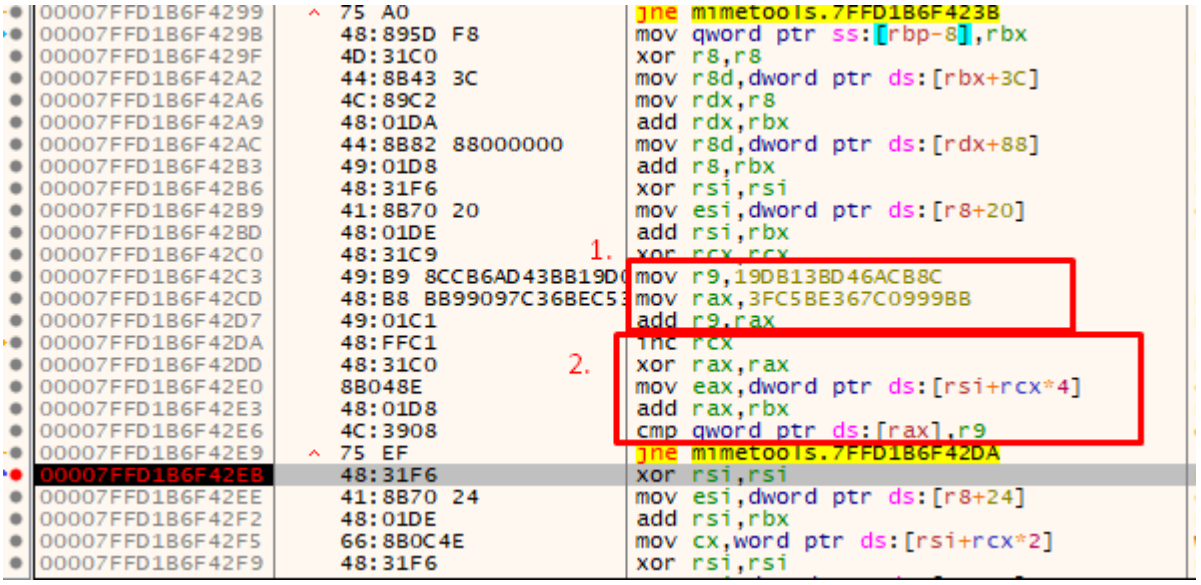
kernel32.dll

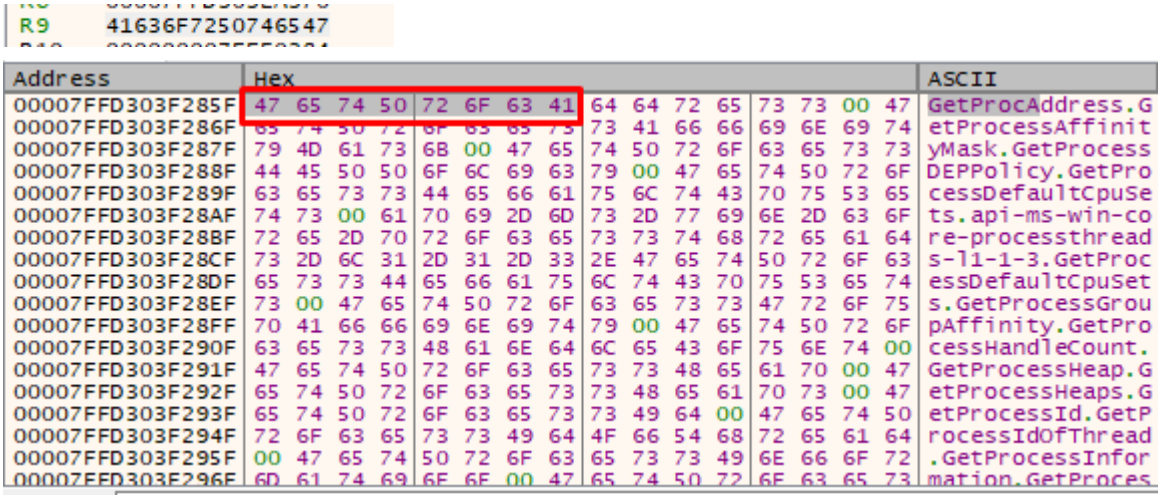
Member	Offset	Size	Value
e_magic	00000000	Word	5A4D
e_cblp	00000002	Word	0090
e_cp	00000004	Word	0003
e_crlc	00000006	Word	0000
e_cparhdr	00000008	Word	0004
e_minalloc	0000000A	Word	0000
e_maxalloc	0000000C	Word	FFFF
e_ss	0000000E	Word	0000
e_sp	00000010	Word	00B8
e_csum	00000012	Word	0000
e_ip	00000014	Word	0000
e_cs	00000016	Word	0000
e_lfarlc	00000018	Word	0040
e_ovno	0000001A	Word	0000
e_res	0000001C	Word	0000
	0000001E	Word	0000
	00000020	Word	0000
	00000022	Word	0000
e_oemid	00000024	Word	0000
e_oeminfo	00000026	Word	0000
e_res2	00000028	Word	0000
	0000002A	Word	0000
	0000002C	Word	0000
	0000002E	Word	0000
	00000030	Word	0000
	00000032	Word	0000
	00000034	Word	0000
	00000036	Word	0000
	00000038	Word	0000
	0000003A	Word	0000
e_lfanew	0000003C	Dword	000000F8



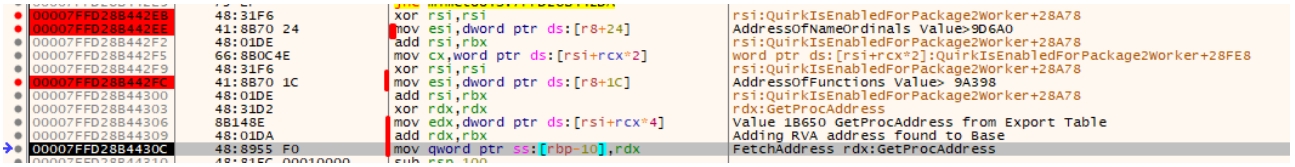
To check it performs some calculations to build a hex value which correspond to the first 8 bytes of "GetProcAddress" keeping in mind endianness. It then loops through all the functions in kernel32.dll to perform compare.

1. Calculates the first 8 bytes in hex for GetProcAddress string (GetProcA)
2. Loops through all the functions and compares the 8 bytes to r9





Next, the malware need to retrieve the function address. To to this it will make use of the ordinal name stored in RCX. The ordinal will be searched withing the AddressOfFunctions struct to get the value.



kernel32.dll

Member	Offset	Size	Value
Characteristics	00099370	Dword	00000000
TimeDateStamp	00099374	Dword	71A43E4A
MajorVersion	00099378	Word	0000
MinorVersion	0009937A	Word	0000
Name	0009937C	Dword	0009E362
Base	00099380	Dword	00000001
NumberOfFunctions	00099384	Dword	00000661
NumberOfNames	00099388	Dword	00000661
AddressOfFunctions	0009938C	Dword	0009A398
AddressOfNames	00099390	Dword	0009BD1C
AddressOfNameOrdinals	00099394	Dword	0009D6A0

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	00099E78	0009CC10	0009B7FC	000A185F
(nFunctions)	Dword	Word	Dword	szAnsi
000002B5	00015840	02B4	000A27FB	GetPrivateProfileStringA
000002B6	00012360	02B5	000A2814	GetPrivateProfileStringW
000002B7	00060CC0	02B6	000A282D	GetPrivateProfileStructA
000002B8	00060E60	02B7	000A2846	GetPrivateProfileStructW
000002B9	0001B650	02B8	000A285F	GetProcAddress
000002BA	0001C890	02B9	000A286E	GetProcessAffinityMask
000002BB	0003A650	02BA	000A2885	GetProcessDEPPolicy
000002BC	000A28B2	02BB	000A2899	GetProcessDefaultCpuSets
000002BD	0003B8F0	02BC	000A28F1	GetProcessGroupAffinity
000002BE	0003B910	02BD	000A2909	GetProcessHandleCount
000002BF	00016190	02BE	000A291F	GetProcessHeap
000002C0	0003B930	02BF	000A292F	GetProcessHeaps

So all this PE parsing is to retrieve finally GetProcAddress and use this API to load others.

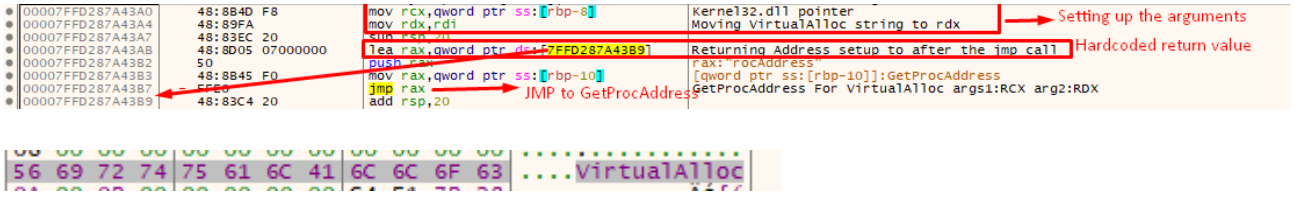
String builder and Control flow manipulation

Next the malware builds the string "VirtualAlloc" and proceeds to user GetProcAddress to fetch the function from kernel32.dll. This is done through jmp instructions, where the function needed is loaded into the appropriate registry.

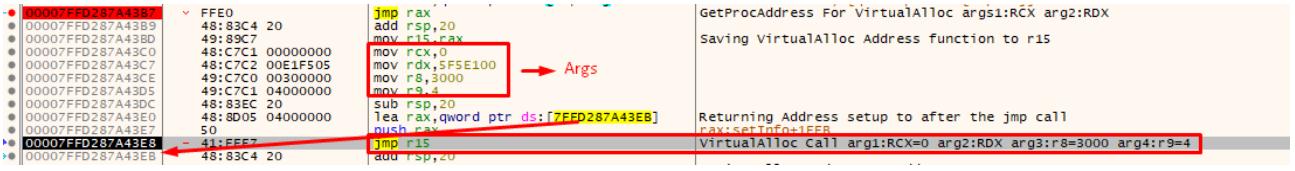
The string is built by using the lower registry al which contains 1 byte. It simply adds two hex values to generate a char. At the end the null byte is used as terminator.

The block of code after this is responsible for the following:

- Sets up the arguments for GetProcAddress in RCX and RDX in this case the module kernel32.dll and API to fetch VirtualAlloc
- Return address following the jmp is saved to rax which is pushed to the stack, this makes it so that when the jmp returns the return address will be on the stack. This points simply to the code following the jmp rax instruction. This is repeated throughout the next jmp instructions to other API.
- Finally it jmp to the GetProcAddress API with the args mentioned above returning the address to VirutalAlloc from kernel32.dll



Next thing of course, the VirtualAlloc is called to allocate some memory space with PAGE_EXECUTE_READWRITE.



Allocated memory buffer:

Throught the entire sample, all API strings to be fetched or DLLs to be loaded are built in a similar fashion as shown above. All calls to any API are changed with JMP instructions where the return value is pushed onto the stack before execution, this is so following the ret from the JMP, the code can continue.

Thread manipulation

Using the same string creation as for VirtualAlloc() and same usage of GetProcAddress(), the malware proceeds to call GetCurrentThreadID() API.

```
00007FFD287A4484   8847 11          mov byte ptr ds:[rdi+11],al          byte ptr ds:[rdi+11]:'d\v'
00007FFD287A4487   C647 12 00       mov byte ptr ds:[rdi+12],0           Second String: GetCurrentThreadId
00007FFD287A448B   48:8B4D F8       mov rcx,qword ptr ss:[rbp-8]         rcx:NtAllocateVirtualMemory+14
00007FFD287A448F   48:89FA          mov rdx,rdi                          rdi:"GetCurrentThreadId\v"
00007FFD287A4492   48:83EC 20       sub rsp,20
00007FFD287A4496   48:8D05 07000000 lea rax,qword ptr ds:[7FFD287A44A4]
00007FFD287A449D   50               push rax
00007FFD287A449E   48:8B45 F0       mov rax,qword ptr ss:[rbp-10]        [qword ptr ss:[rbp-10]]:GetProcAddres
00007FFD287A44A2   FFE0             jmp rax                              GetProcAddress
00007FFD287A44A4   48:83C4 20       add rsp,20
00007FFD287A44A8   49:89C7          mov r15,rax                          r15:VirtualAlloc
00007FFD287A44AB   48:83EC 20       sub rsp,20
00007FFD287A44AF   48:8D05 04000000 lea rax,qword ptr ds:[7FFD287A44BA]
00007FFD287A44B6   50               push rax
00007FFD287A44B7   41:FFE7          jmp r15                              GetCurrentThreadId
00007FFD287A44BA   48:83C4 20       add rsp,20
```

Following OpenThread() call.

```
00007FFD287A4503   8847 09          mov byte ptr ds:[rdi+9],al
00007FFD287A4506   C647 0A 00       mov byte ptr ds:[rdi+A],0            OpenThread
00007FFD287A450A   48:8B4D F8       mov rcx,qword ptr ss:[rbp-8]
00007FFD287A450E   48:89FA          mov rdx,rdi                          rdi:"OpenThread"
00007FFD287A4511   48:83EC 20       sub rsp,20
00007FFD287A4515   48:8D05 07000000 lea rax,qword ptr ds:[7FFD287A4523]  rax:setInfo+215A
00007FFD287A451C   50               push rax                             rax:setInfo+215A
00007FFD287A451D   48:8B45 F0       mov rax,qword ptr ss:[rbp-10]        rax:setInfo+215A, [qword ptr ss:[rbp-10]]:GetProcAddress
00007FFD287A4521   FFE0             jmp rax                              rax:setInfo+215A
00007FFD287A4523   48:83C4 20       add rsp,20
00007FFD287A4527   49:89C7          mov r15,rax                          r15:OpenThread, rax:setInfo+215A
00007FFD287A452A   48:C7C1 02000000 mov rcx,2                            arg1
00007FFD287A4531   48:C7C2 00000000 mov rdx,0                            arg2
00007FFD287A4538   4D:89E0          mov r8,r12                           r12=CurrentThreadId
00007FFD287A453B   48:83EC 20       sub rsp,20
00007FFD287A453F   48:8D05 04000000 lea rax,qword ptr ds:[7FFD287A454A]  rax:setInfo+215A
00007FFD287A4546   50               push rax                             rax:setInfo+215A
00007FFD287A4547   41:FFE7          jmp r15                              jmp to OpenThread
00007FFD287A454A   48:83C4 20       add rsp,20
```

Finally a new thread is created by calling CreateThread(). The arguments passed:

- Pointer to start address in R8
- Pointer to parameter passed to thread in R9, in this case seems to be the ID handle to that thread.

Image

Thread	DLLLoader64_D313.exe (2540): 4168	0xf0

The malware will then follow by suspending the main thread and leave execution to the newly created one. For debugging purposes, the debugger was acting unpredictably maybe because of race conditions and threads colliding. Once the code reached the new thread start, I manually killed the main thread this seems to have worked but not 100% sure if it is just a fluke.

Decrypting Shellcode

Shellcode is decrypted from the certificate.pem file. Malware uses CreateFileA() to open the file and then ReadFile() to read the contents. It is then decrypted using the key, with AES in CBC mode.

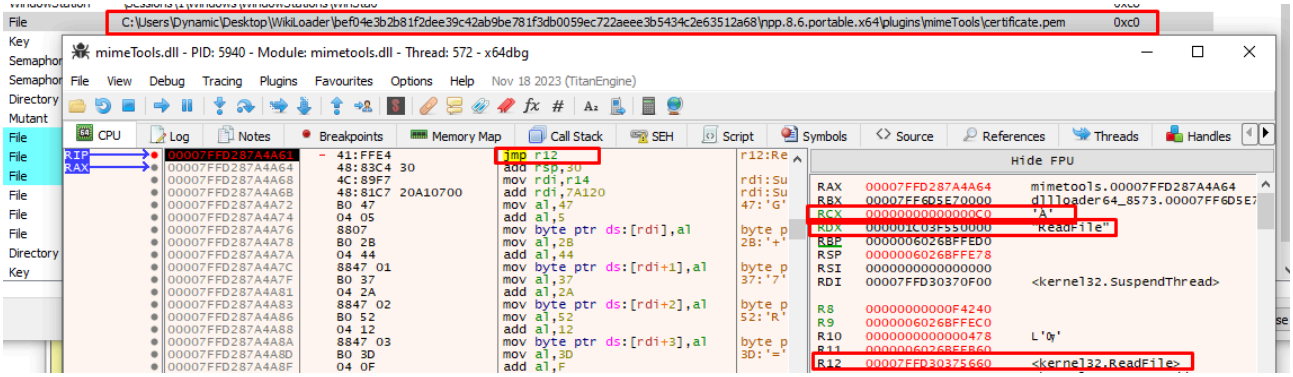
- key: HjGwPX65nXm21XOkX4nMqex28mBHjNU7

The decryption functions are not custome, and uses the bcrypt library which is loaded after file is read.

```
00007FFD287A49B5   41:FFE4          jmp r12                              Hide FPU
00007FFD287A49B8   48:83C4 40       add rsp,40                           RAX 00007FFD287A49B8  mimetools.00007FFD287A
00007FFD287A49BC   48:83F8 FF       cmp rax,FFFFFFFFFFFFFFFF             RBX 00007FFE6D5E70000 dllloader64_8573.00007
00007FFD287A49C0   0F84 BA130000    je mimetools.7FFD287A5D80            RCX 000001C03F550000  "certificate.pem"
00007FFD287A49C6   48:8945 F8       mov qword ptr ss:[rbp-8],ra          RDX 0000000080000000
00007FFD287A49CA   B0 11            mov al,11                            RBP 000000026BFFED0
00007FFD287A49CC   04 41            add al,41                            RSP 000000026BFFE68
00007FFD287A49CE   41:8806          mov byte ptr ds:[r14],al             RSI 0000000000000000
00007FFD287A49D1   B0 45            mov al,45                            RDI 00007FFD30370F00  <kernel32.SuspendThrea
00007FFD287A49D3   04 20            add al,20
00007FFD287A49D5   41:8846 01       mov byte ptr ds:[r14+1],al           R8  0000000000000000
00007FFD287A49D9   B0 36            mov al,36                            R9  0000000000000000
00007FFD287A49DB   04 2B            add al,2B                            R10 00000000000000C6  '£'
00007FFD287A49DD   41:8846 02       mov byte ptr ds:[r14+2],al           R11 000000026BFFB60
00007FFD287A49E1   B0 33            mov al,33                            R12 00007FFD303752D0  <kernel32.CreateFileA>
00007FFD287A49E3   04 31            add al,31
00007FFD287A49E5   41:8846 03       mov byte ptr ds:[r14+3],al
00007FFD287A49E9   41:57            push r15
```

For ReadFile the two arguments correspond to:

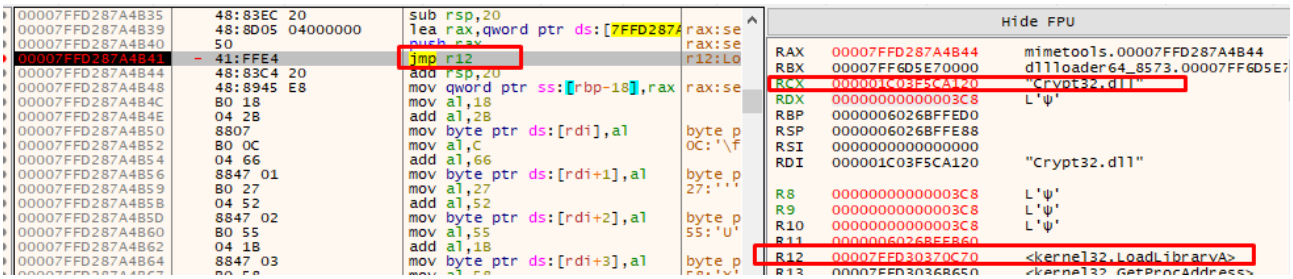
- File Handle
- Memory buffer to store data that is read.



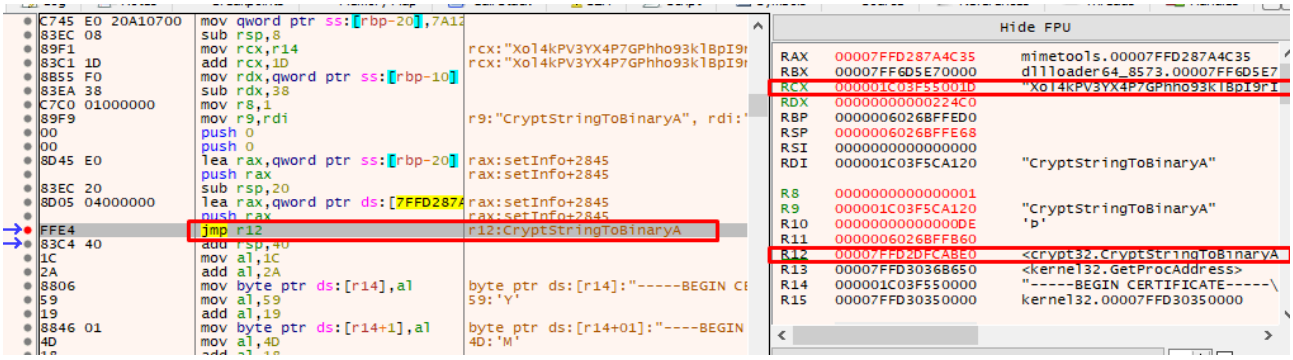
Address	Hex	ASCII
000001C03F550000	2D 2D 2D 2D 2D 42 45 47 49 4E 20 43 45 52 54 49	-----BEGIN CERTI
000001C03F550010	46 49 43 41 54 45 2D 2D 2D 2D 2D 0D 0A 58 6F 6C	FICATE-----..Xol
000001C03F550020	34 6B 50 56 33 59 58 34 50 37 47 50 68 68 6F 39	4kPV3YX4P7GPhho9
000001C03F550030	33 6B 6C 42 70 49 39 72 49 50 44 77 41 34 30 78	3klBpI9rIPDwA40x
000001C03F550040	50 44 55 30 68 2B 69 46 6E 32 46 76 68 6E 41 6C	PDU0h+iFn2FvhnAl
000001C03F550050	6F 6E 39 68 52 42 65 66 4B 36 77 54 45 37 63 44	on9hRBefK6wTE7cD
000001C03F550060	38 45 54 72 79 5A 4D 72 64 34 41 6F 59 4D 44 77	8ETryZMrd4AoYMDw
000001C03F550070	56 42 70 61 73 59 4A 43 62 6D 4A 62 7A 77 58 38	VBpasYJCbmJbzwX8
000001C03F550080	4E 70 37 4E 55 75 39 57 69 2B 70 30 68 52 6A 75	Np7NUu9Wi+p0hRju
000001C03F550090	55 64 31 68 66 49 39 6F 36 33 48 6B 35 77 78 49	Ud1hfI9o63Hk5wxI
000001C03F5500A0	4E 36 4F 6A 72 37 51 65 32 4D 7A 78 65 66 50 62	N6Ojr7Qe2MzxefPb
000001C03F5500B0	48 4A 4C 78 4C 47 4C 5A 47 43 76 6A 61 34 5A 57	HJLxLGLZGCvja4ZW
000001C03F5500C0	63 6F 34 76 35 69 77 50 37 48 73 6F 51 4B 62 52	co4v5iwP7HsoQKbR
000001C03F5500D0	6D 36 2F 30 44 48 74 34 61 33 76 70 59 42 4B 4F	m6/0DHt4a3vpYBKO
000001C03F5500E0	62 49 55 44 36 56 76 41 77 56 79 78 46 66 7A 6E	bIUD6VvAwVyxFfzn
000001C03F5500F0	6C 4E 56 33 79 51 2B 62 68 7A 6F 73 64 55 30 54	lNV3yQ+bhzosdU0T
000001C03F550100	39 4E 67 5A 46 71 2F 45 7A 50 74 6E 46 6B 76 49	9NgZFq/EzPtnFkvI
000001C03F550110	64 4E 6C 42 70 49 6D 53 64 55 41 31 4B 66 6F 6E	dNlBpImSdUA1Kfon
000001C03F550120	52 44 2F 73 72 6F 44 74 45 30 48 69 39 48 67 45	RD/sroDtE0Hi9HgE
000001C03F550130	42 43 2B 6B 51 6C 34 44 34 64 64 5A 77 74 54 46	BC+kQl4D4ddZwtTF
000001C03F550140	36 77 39 63 47 55 62 67 35 30 2B 30 50 5A 75 35	6w9cGUbg50+0PZu5
000001C03F550150	66 4F 57 53 57 47 53 59 78 42 53 4E 68 4E 72 61	fOWSWGSYxBSNhNra

Now to proceed with decryption part some setup is necessary.

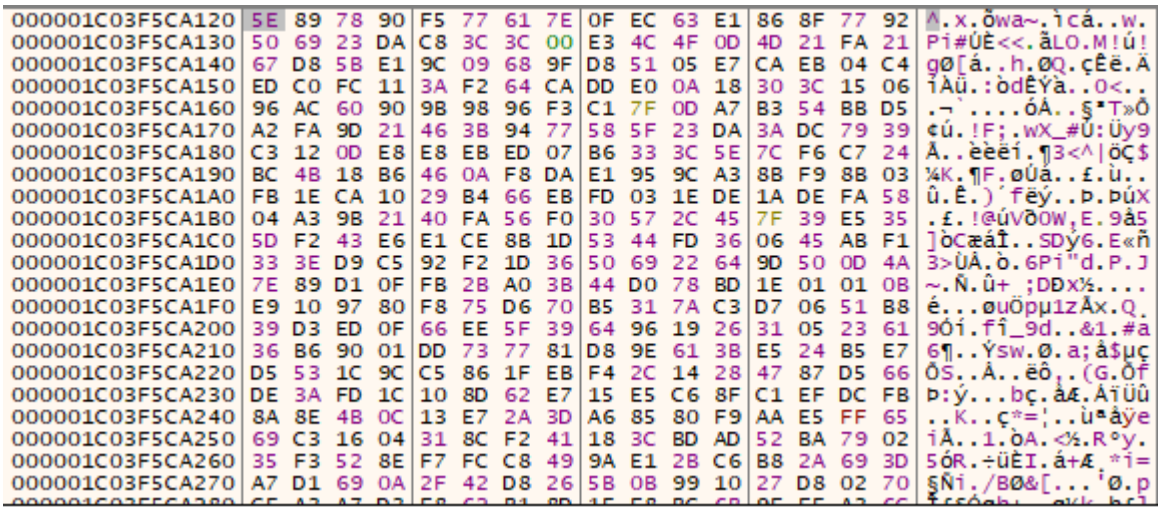
- Crypt32.dll is loaded
- CryptStringToBinaryA is called on the data of the certificate.pem file saved in the memory.



CryptStringToBinaryA is used to convert a formatted string into an array of bytes. The flags supplied is in R8 and is 0x1 which means format of the string to be converted is base64. R9 contains the buffer that will receive the output of the function.

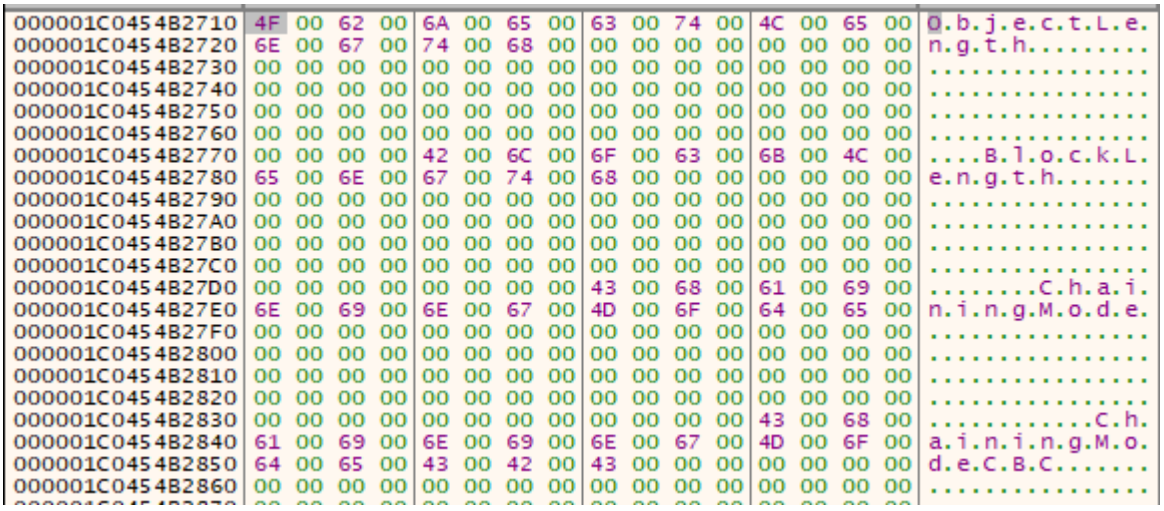


The result is the following:



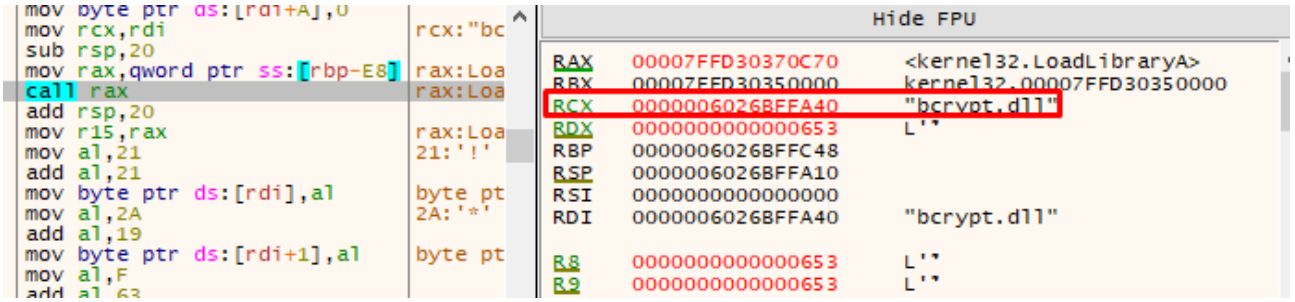
Next setup phase, the malware builds string that will be used later.

- ObjectLength
- BlockLength
- ChainingMode
- ChainingModeCBC



bcrypt.dll module is loaded and proceeds to get addresses for these functions and the jmps to them.

- BCryptOpenAlgorithmProvider
- BCryptGetProperty
- BCryptSetProperty
- BCryptGenerateSymmetricKey
- BCryptDecrypt
- BCryptDestroyKey
- BCryptCloseAlgorithmProvider

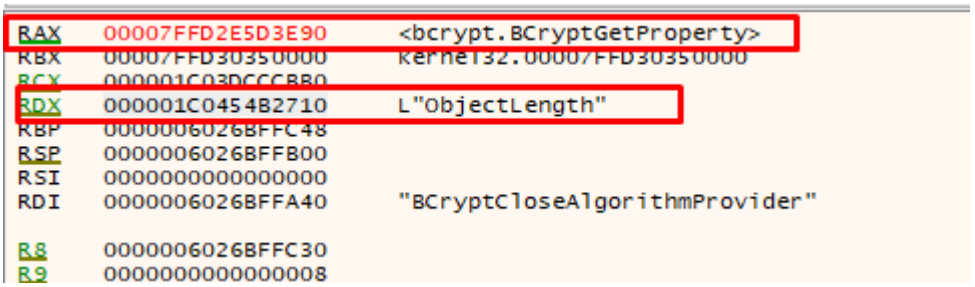


First call of the chain is of course BCryptOpenAlgorithmProvider with "AES" supplied as argument. All these functions are from Microsoft DLL so can be searched on official docs for more information.

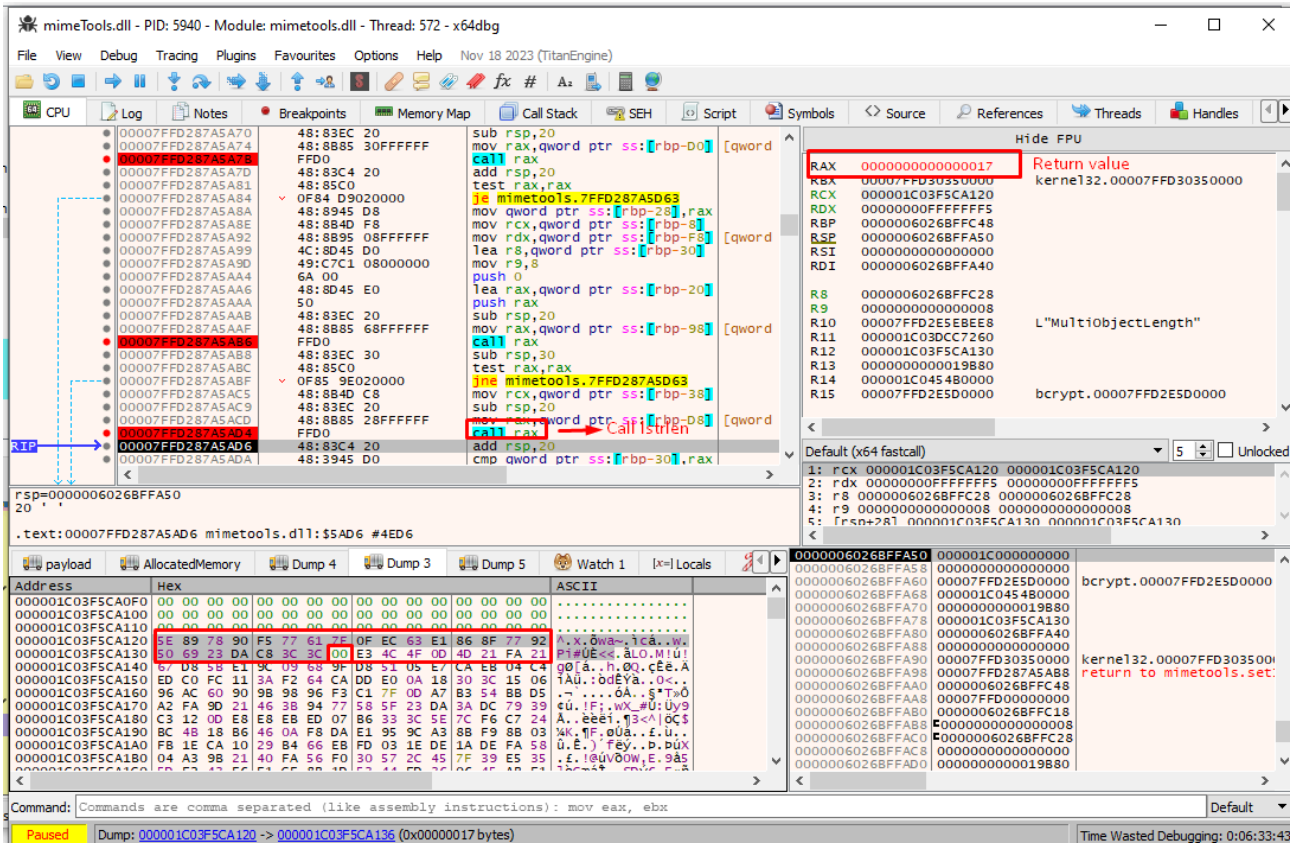
Second call makes use of the initial strings saved at the first setup stage. It calls on BCryptGetProperty for:

- ObjectLength
- BlockLength -> blocks of 4 bytes
- ChainingMode
- ChainingModeCBC

After the first one, some memory is allocated. After the second, lstrlen is called to get size of the payload that needs to be decrypted but reads only until first null byte so 17 bytes. Also, VirtuaAlloc() is called again and the first 10 bytes of the initial payload are written to it.



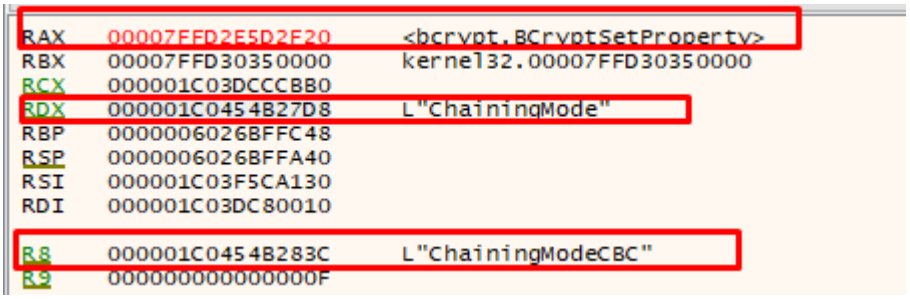
```
RAX   00007FFD2E5D3E90    <bcrypt.BCryptGetProperty>
RBX   00007FFD30350000    kernel32.00007FFD30350000
RCX   000001C03DCCCBB0
RDX   000001C0454B2774    L"BlockLength"
RBP   0000006026BFFC48
RSP   0000006026BFFAA0
RSI   0000000000000000
RDI   0000006026BFFA40

R8    0000006026BFFC18
R9    0000000000000008
```



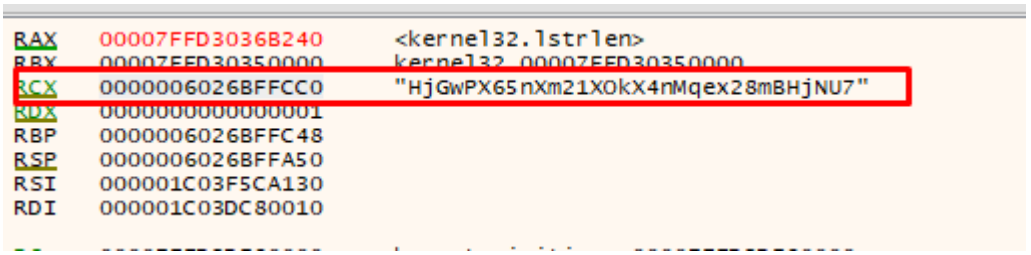
lstrlenw() is called to count "ChainingModeCBC" string. This is necessary for the next call, which is BCryptSetProperty(). The arguements relevant are:

- szProperty -> "A pointer to a null-terminated Unicode string that contains the name of the property to set." In this case "ChainingMode"
- pbInput -> "The address of a buffer that contains the new property value. The cbInput parameter contains the size of this buffer." In our case this would be "ChainingModeCBC" with size 15 calculated in the step before.

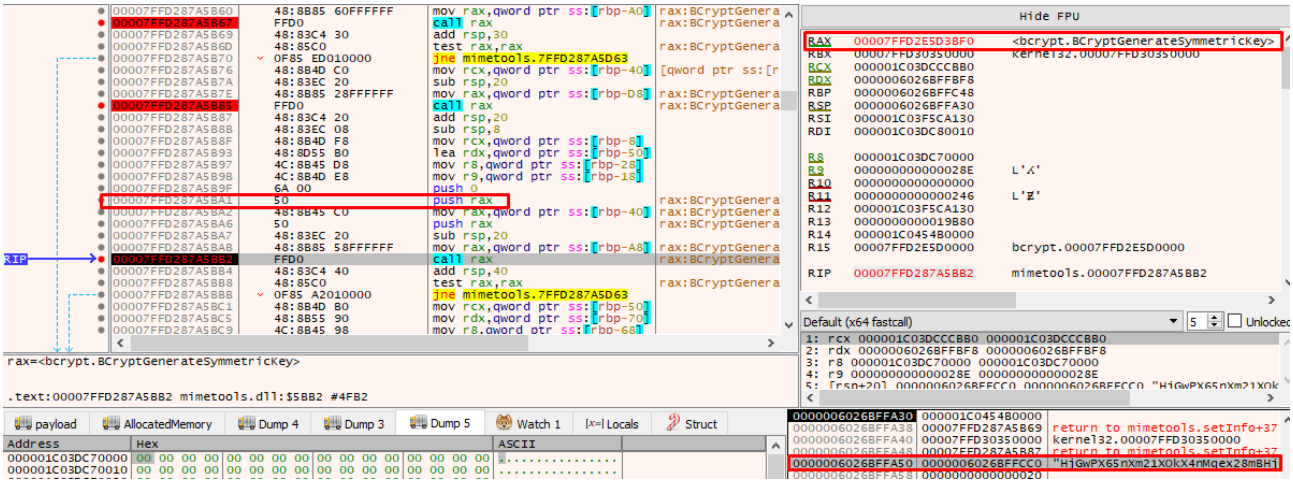
```
RAX   00007FFD303675E0    <kernel32.lstrlenW>
RBX   00007FFD30350000    kernel32.00007FFD30350000
RCX   000001C0454B283C    L"ChainingModeCBC"
RDX   0000000000000000
RBP   0000006026BFFC48
RSP   0000006026BFFA50
RSI   000001C03F5CA130
RDI   000001C03DC80010

R8    0000006026BFFA08
R9    0000006026BFFC48
```



Next call is to get size of the key:



symmetric key is generated with BCryptGenerateSymmetricKey(). The fifth arguement is the actual key and is passed as reference onto the stack.

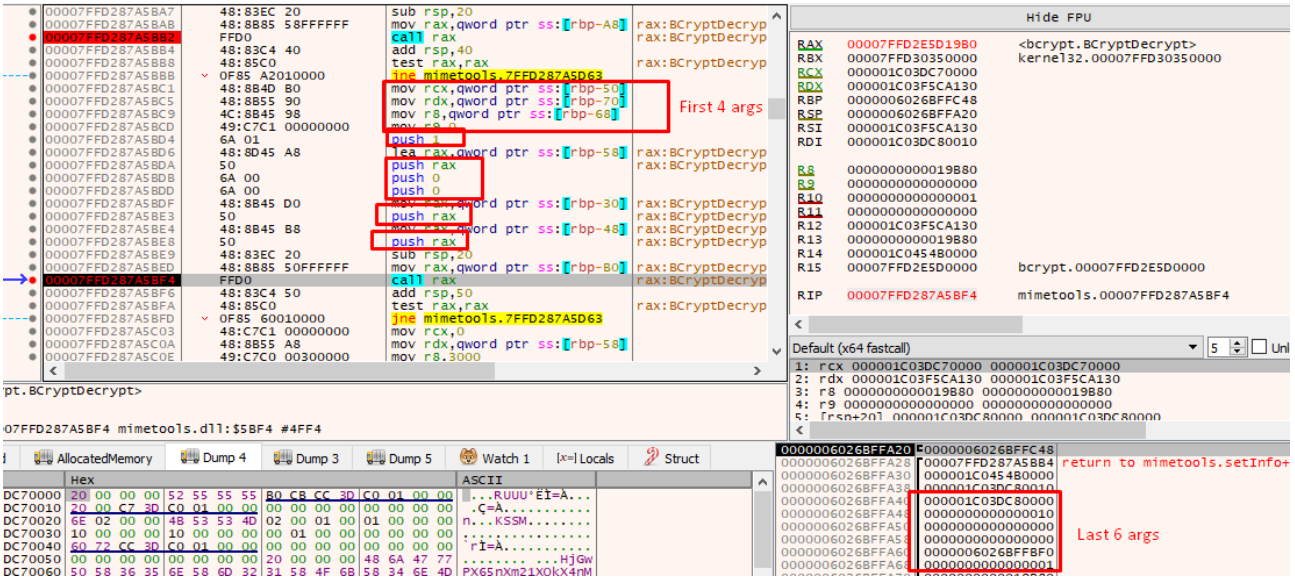


Now, the interesting part BCryptDecrypt() is called. This is responsible for decrypting the shellcode. Let's look at the arguments in the debugger. We can use the Microsoft documentation.

```
C++

NTSTATUS BCryptDecrypt(
  [in, out]           BCRYPT_KEY_HANDLE hKey,
  [in]                PUCHAR            pbInput,
  [in]                ULONG             cbInput,
  [in, optional]      VOID              *pPaddingInfo,
  [in, out, optional] PUCHAR            pbIV,
  [in]                ULONG             cbIV,
  [out, optional]     PUCHAR            pbOutput,
  [in]                ULONG             cbOutput,
  [out]               ULONG             *pcbResult,
  [in]                ULONG             dwFlags
);
```

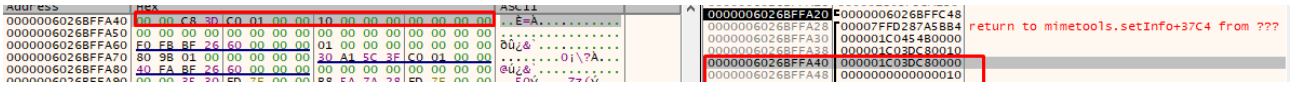
First 4 args are passed in: RDX,RCX,R8,R9 The rest are passed as reference onto the stack. The arguments we are interested in are:

- pbInput The address of a buffer that contains the ciphertext to be decrypted. Second argument.
- pbIV The address of a buffer that contains the initialization vector (IV) to use during decryption. Fifth argument.
- pbOutput The address of a buffer to receive the plaintext produced by this function. Seventh argument.



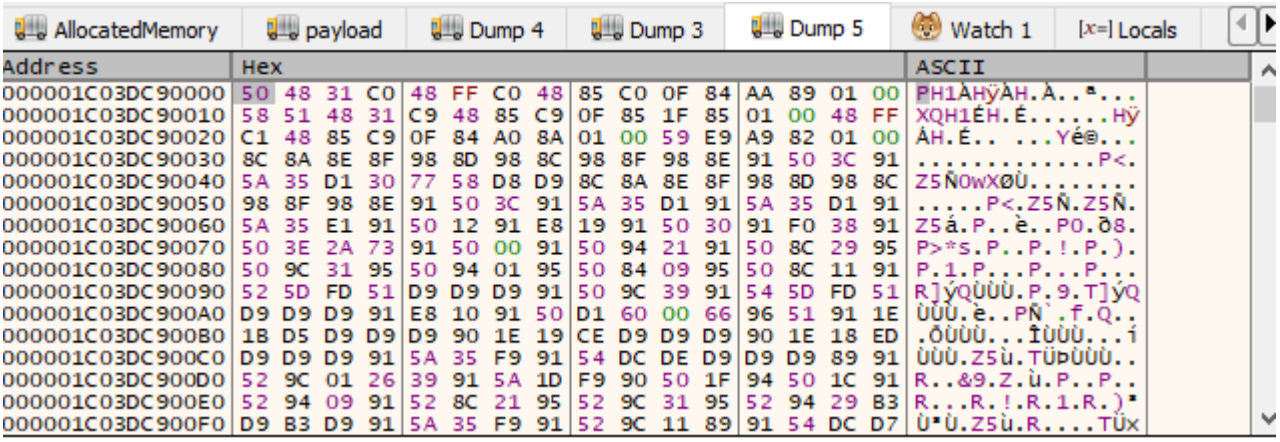
Second Argument (RCX) contains the certificate.pem data that was read into memory before.

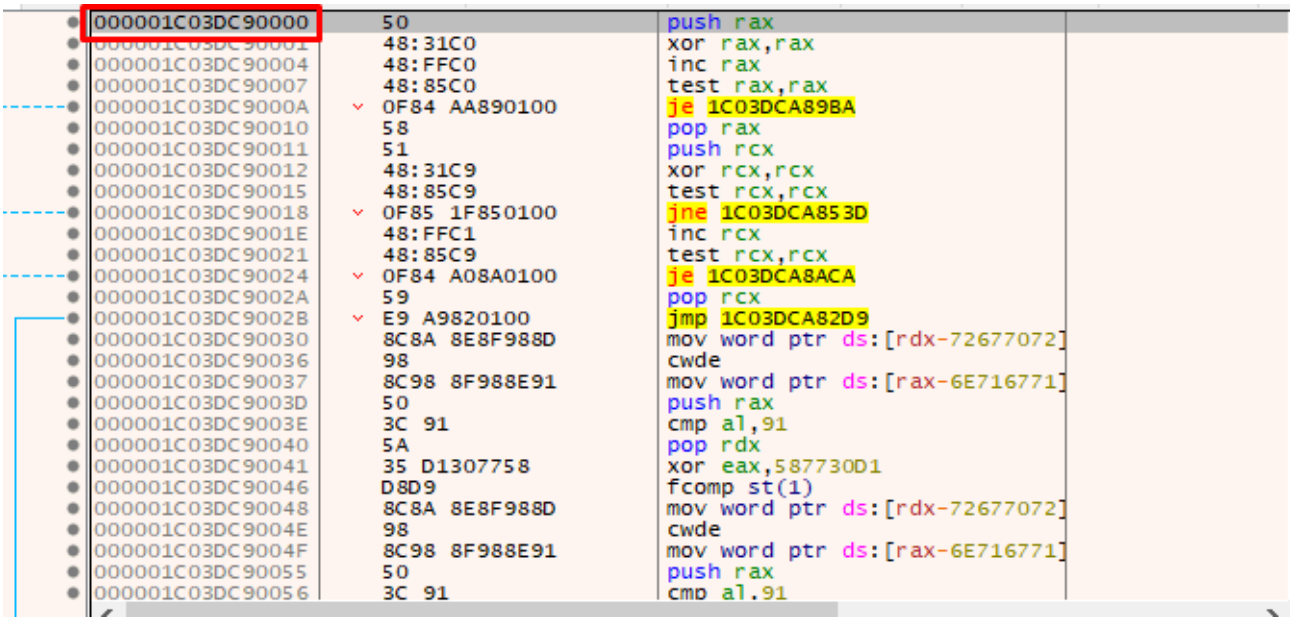
The Fifth argument has the IV. This is on the stack, and it is 10 bytes according the the sixth argument.



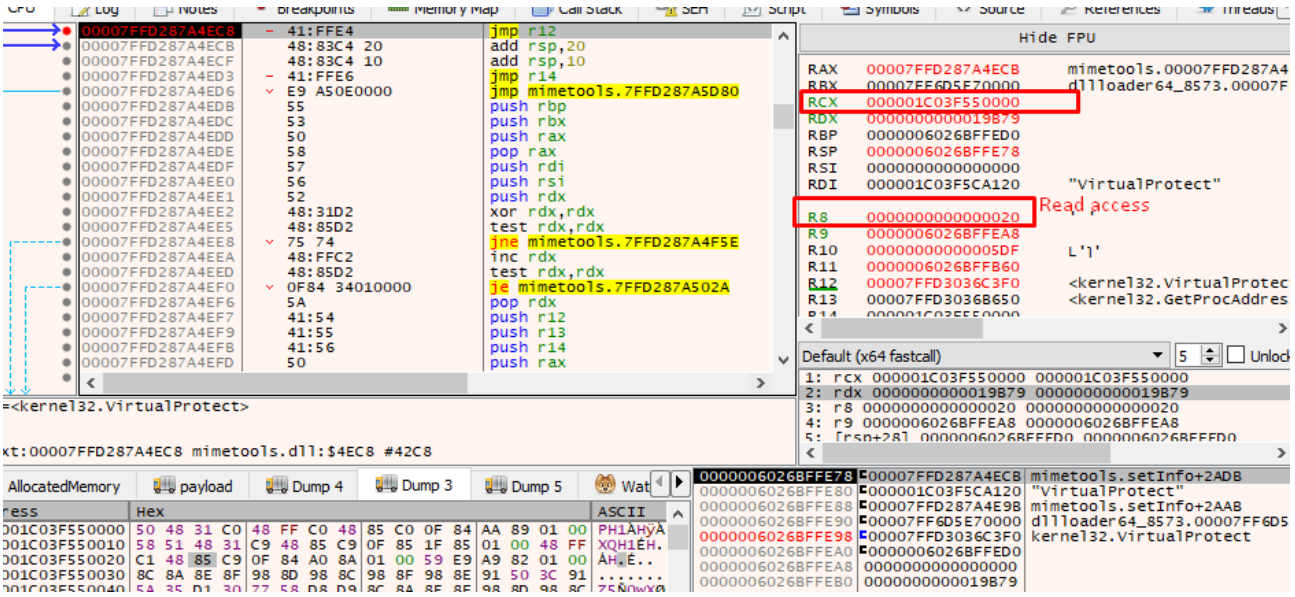
The pbOutput is null in this case, so no output found.

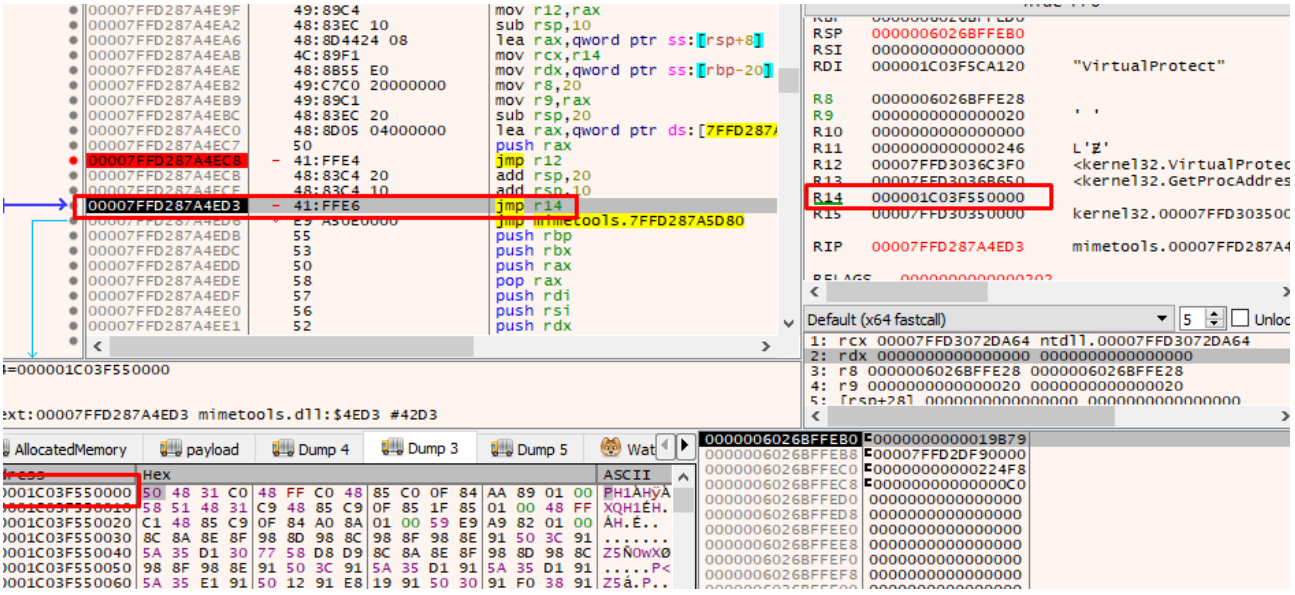
Following this VirtualAlloc is used to allocate a memory buffer and BCryptDecrypt() is called once again, this time the pbOutput is a pointer to the newly allocated memory. In this case the output can be observed, and we can see the decrypted shellcode.



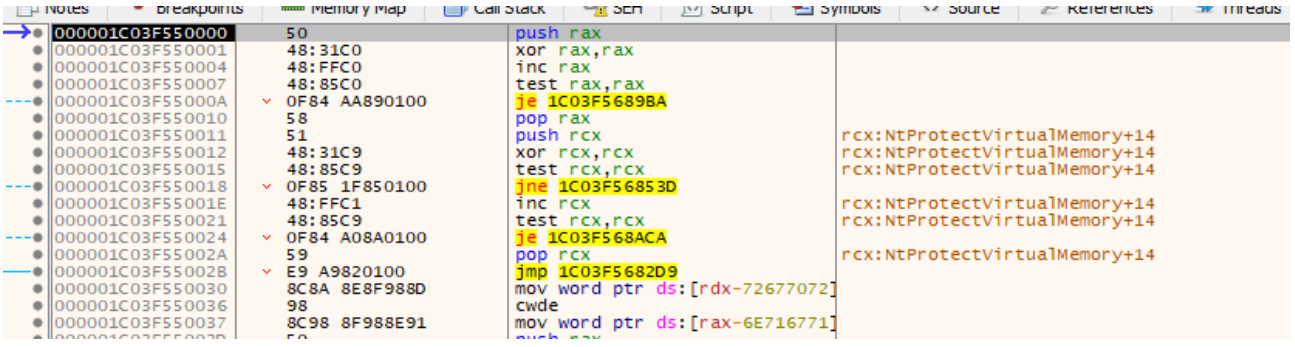


To make memory section which will contain the payload executable, the malware calls on VirtualProtect() onto that section. The call is used to give the section of memory 0x20-> PAGE_EXECUTE_READ. Finally, the malware can jump to the memory location and resume execution!





With this we have finally made it to the first decrypted shellcode!



Since this was pretty long I will continue with a part 2 soon!

References

- https://bazaar.abuse.ch/sample/bef04e3b2b81f2dee39c42ab9be781f3db0059ec722aeee3b5434c2e63512a68/

- https://www.unpac.me/results/612d6d2c-c45d-47ba-a2bb-a218ec753d3f
- https://twitter.com/Cryptolaemus1/status/1747394506331160736
- https://www.proofpoint.com/us/blog/threat-insight/out-sandbox-wikiloader-digs-sophisticated-evasion
- https://www.geoffchappell.com/studies/windows/km/ntoskrnl/inc/api/pebteb/peb/index.htm
- https://mohamed-fakroud.gitbook.io/red-teamings-dojo/shellcoding/leveraging-from-pe-parsing-technique-to-write-x86-shellcode

---