

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

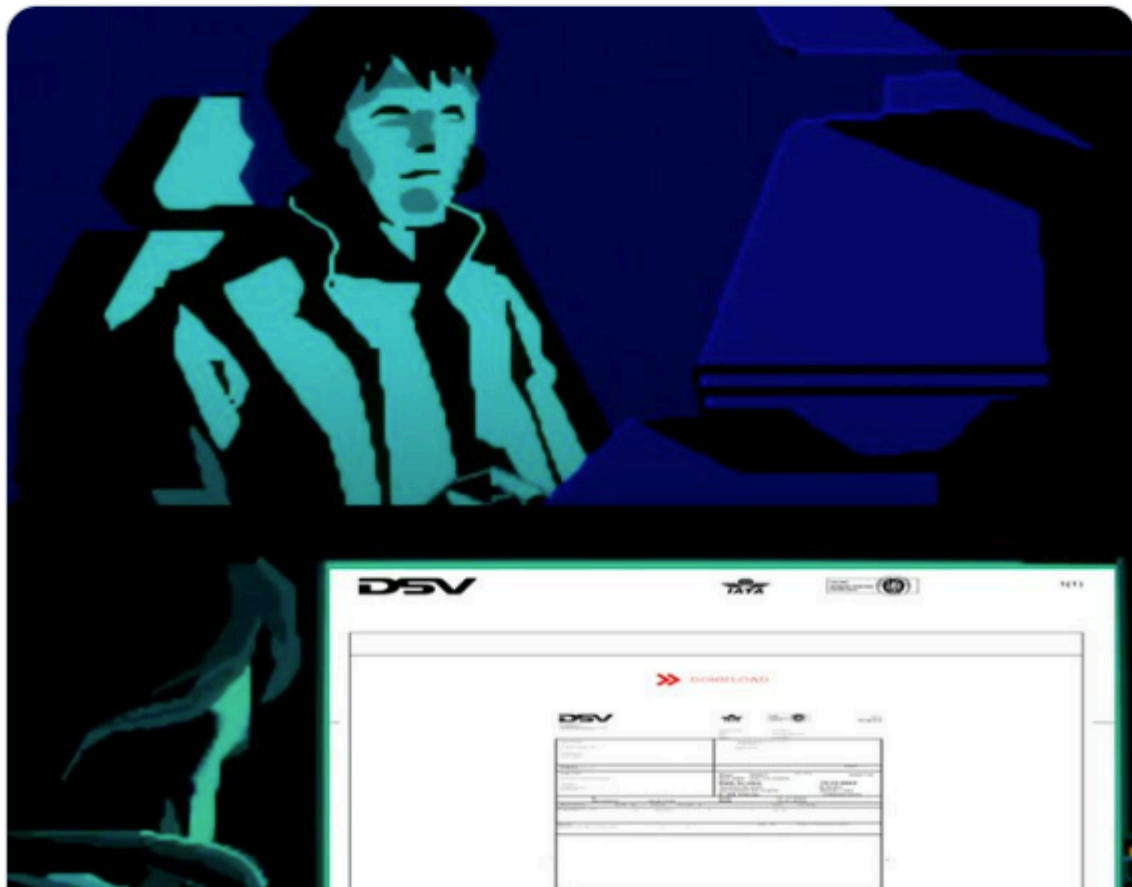
C:\Users\User\AppData\Local\Temp\npp.8.6.portable.x64\notepad.exe
(soload) 📌

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

(1/3) 📌

IOC's

github.com/prOxylife/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

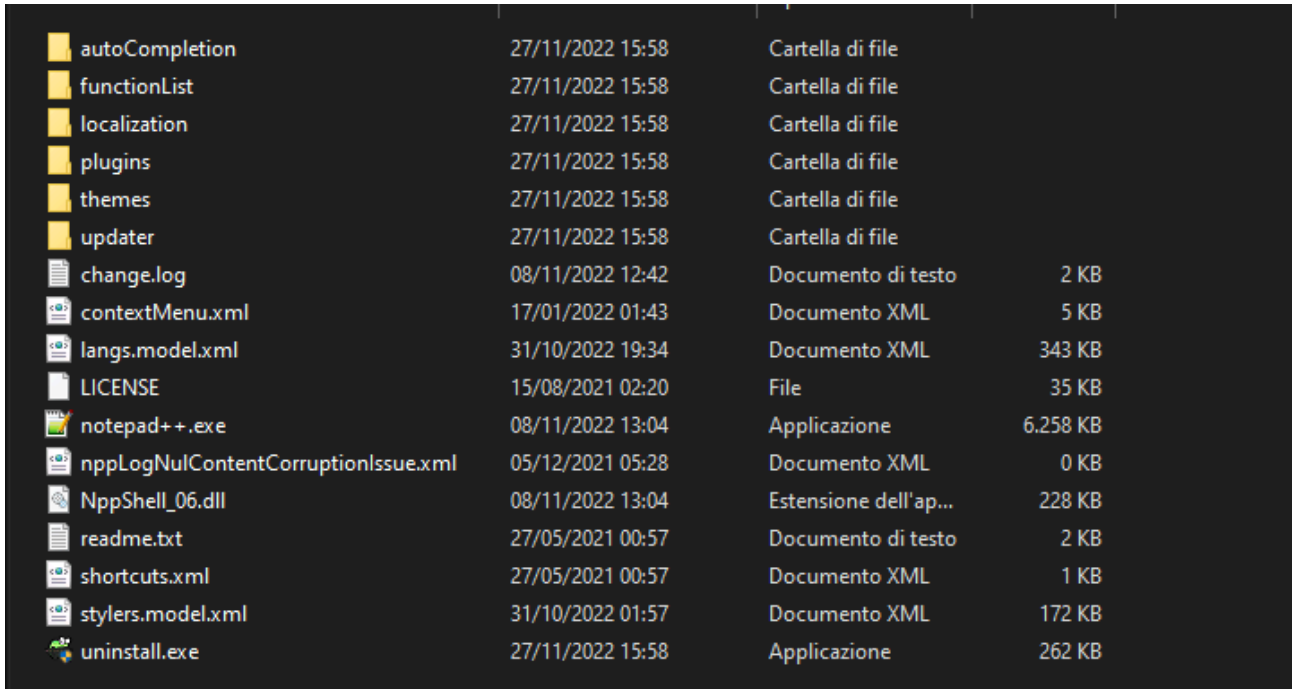
ZIP file contents and "notepad"

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
toolbarcons.xml	15/03/2023 02:12	Documento XML	3 KB

Legitimate notepad++.exe folder structure.

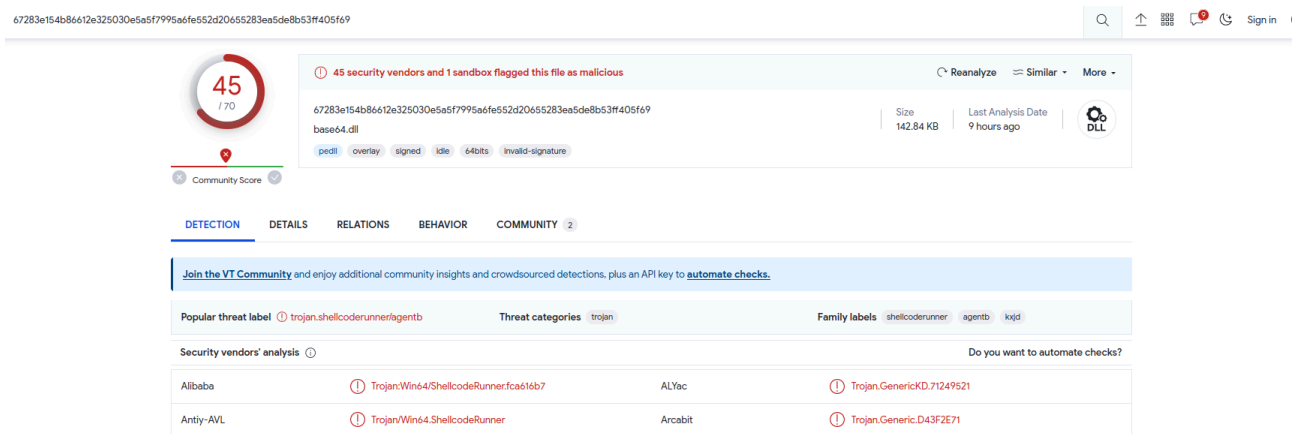


File/Folder	Creation Date	Type	Size
autoCompletion	27/11/2022 15:58	Cartella di file	
functionList	27/11/2022 15:58	Cartella di file	
localization	27/11/2022 15:58	Cartella di file	
plugins	27/11/2022 15:58	Cartella di file	
themes	27/11/2022 15:58	Cartella di file	
updater	27/11/2022 15:58	Cartella 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.



67283e154b86612e325030e5a5f7995a6fe552d20655283ea5de8b53f1405f69

45 / 170

45 security vendors and 1 sandbox flagged this file as malicious

base64.dll

Size: 142.84 KB | Last Analysis Date: 9 hours ago

pedi overlay signed idle 64bits invalid-signature

Community Score

DETECTION DETAILS RELATIONS BEHAVIOR COMMUNITY

Join the VT Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label: trojan.shellcoderrunneragentb | Threat categories: trojan | Family labels: shellcoderrunner, agentb, kxjd

Security vendors' analysis

Vendor	Detection	Vendor	Detection
Allbaba	Trojan:Win64/ShellcodeRunner:fa616b7	ALYac	Trojan.GenericKD.71249521
Antiy-AVL	Trojan:Win64.ShellcodeRunner	Arcabit	Trojan.Generic.D43F2E71

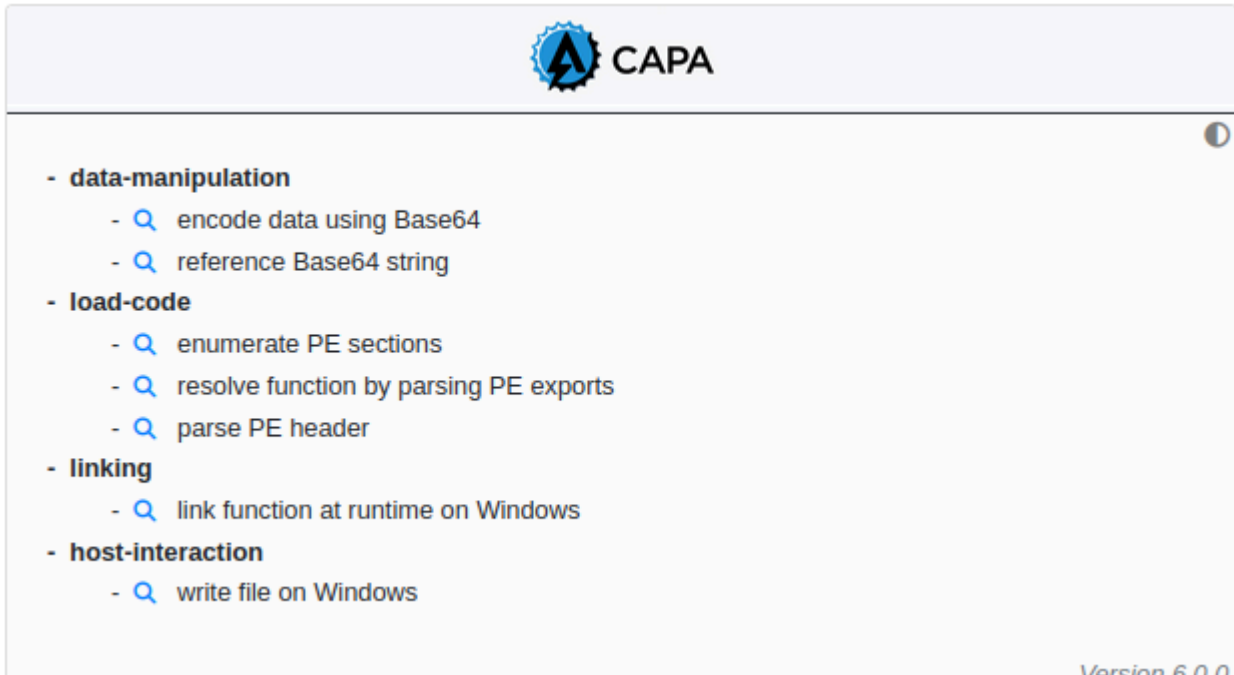
Static Analysis

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 use to help our analysis. Although it may not be useful in the end.

```
FLOSS Stack Strings

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

FLOSS extracted 67 stackstrings
nXm21X0k}Z
nXm21X0k}Z
VirtualAlloc
CryptStringToBinaryA
CryptStringToBinaryA
ex28mBHjNU7
ex28mBHjNU7
CryptStringToBinaryA
8mBHjNU7
HjGwPX65nXm21X0kX4nMqex28mBHjNU7
ex28mBHjNU7
LoadLibraryA
ZBAA
Virtu
```

VirtualAlloc
nXm21X0k}Z
VirtualAlloc
VirtualAlloc
8mBHjNU7
VirtualFreeA
VirtualFreeA
8mBHjNU7
VirtualFreeA
VirtualAlloc
ex28mBHjNU7
HjGwPX65nXm21X0kX4nMqex28mBHjNU7
LoadLibraryA
nXm21X0k}Z
VirtualFreeA
HjGwPX65nXm21X0kX4nMqex28mBHjNU7
HjGwPX65nXm21X0kX4nMqex28mBHjNU7
LoadLibraryA
CryptStringToBinaryA
LoadLibraryA
8mBHjNU7
Virtu
LoadLibraryA
VirtualFreeA
HjGwPX65nXm21X0kX4nMqex28mBHjNU7
nXm21X0k}Z
VirtualFreeA
Virtu
8mBHjNU7
Virtu
ex28mBHjNU7
CryptStringToBinaryA
LoadLibraryA
LoadLibraryA
8mBHjNU7
LoadLibraryA
HjGwPX65nXm21X0kX4nMqex28mBHjNU7
8mBHjNU7
VirtualAlloc
ex28mBHjNU7
CryptStringToBinaryA
ex28mBHjNU7
nXm21X0k}Z
HjGwPX65nXm21X0kX4nMqex28mBHjNU7
nXm21X0k}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 register
- 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.

RIP	RAX	R12	Disassembly	Comment
00007FFD186F41CC			41:57 push r15	OptionalHeader.AddressOfEntryPoint
00007FFD186F41CE			40:31FF xor r15,r15	
00007FFD186F41D1			49:9FC7 inc r15	
00007FFD186F41D4			40:85FF test r15,r15	
00007FFD186F41D7			0F84 8D040000 jle mimetool1s.7FFD186F466A	
00007FFD186F41D8			41:5F pop r15	
00007FFD186F41DF			53 push rbx	
00007FFD186F41E0			55 push rbp	
00007FFD186F41E1			57 push rdi	
00007FFD186F41E2			56 push rsi	
00007FFD186F41E3			41:54 push r12	r12:setInfo+1DDC
00007FFD186F41E5			41:55 push r13	
00007FFD186F41E7			41:56 push r14	
00007FFD186F41E9			41:57 push r15	
00007FFD186F41EB			48:83EC 08 sub rsp,8	
00007FFD186F41EF			48:89E5 mov rbp,rbp	
00007FFD186F41F2			48:83FA 01 cmp rdx,1	
00007FFD186F41F6			0F85 A31C0000 jne mimetool1s.7FFD186F5E9F	
00007FFD186F41FC			49:89D5 mov r13,rdx	
00007FFD186F41FF			48:83EC 20 sub rsp,20	
00007FFD186F4203			48:89CB mov rbx,rcx	rax:setInfo+1DDC
00007FFD186F4206			48:31C0 xor rax,rax	
00007FFD186F4209			48:89E9 mov rcx,rbp	
00007FFD186F420C			48:29E1 sub rcx,rbp	
00007FFD186F420F			48:89E7 mov rdi,rbp	
00007FFD186F4212			F3:AA rep stosb	
00007FFD186F4214			48:89D9 mov rcx,rbx	
00007FFD186F4217			48:C7C0 60000000 mov rax,60	rax:setInfo+1DDC, 60: Access PEB GS:60
00007FFD186F421E			65:45:8B15 mov rbx,qword ptr ds:[rax]	Offset value for PEB_LDR_DATA
00007FFD186F4222			48:C7C0 18000000 mov rax,18	PEB_LDR_DATA OFFSET 0x18
00007FFD186F4229			48:8B1C03 mov rbx,qword ptr ds:[rbx+rax]	Offset value for LIST_ENTRY InMemoryOrderModuleList
00007FFD186F4220			48:C7C0 20000000 mov rax,20	LIST_ENTRY InMemoryOrderModuleList
00007FFD186F4234			48:8B1C03 mov rbx,qword ptr ds:[rbx+rax]	r12:setInfo+1DDC
00007FFD186F4238			49:89DC mov r12,rbx	
00007FFD186F423B			48:8B53 40 mov rdx,qword ptr ds:[rbx+40]	
00007FFD186F423F			807A 28 48 cmp byte ptr ds:[rdx+28],48	48: 'k'
00007FFD186F4243			74 08 jle mimetool1s.7FFD186F4240	68: 'k'
00007FFD186F4245			807A 28 68 cmp byte ptr ds:[rdx+28],68	
00007FFD186F4249			74 02 jle mimetool1s.7FFD186F4240	
00007FFD186F424B			EB 46 jmp mimetool1s.7FFD186F4239	
00007FFD186F424D			807A 30 45 cmp byte ptr ds:[rdx+30],45	45: 'e'
00007FFD186F4251			74 08 jle mimetool1s.7FFD186F4258	
00007FFD186F4253			807A 30 65 cmp byte ptr ds:[rdx+30],65	65: 'e'
00007FFD186F4257			74 02 jle mimetool1s.7FFD186F4258	
00007FFD186F4259			EB 38 jmp mimetool1s.7FFD186F4239	

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'

00007FFD186F421E	6548:8B18	mov rbx,qword ptr [rax]	Access PEB GS:60
00007FFD186F4222	48:C7C0 18000000	mov rax,18	Offset value for PEB_LDR_DATA
00007FFD186F4229	48:8B1C03	mov rbx,qword ptr ds:[rbx+rax]	PEB_LDR_DATA OFFSET 0x18
00007FFD186F422D	48:C7C0 20000000	mov rax,20	Offset value for LIST_ENTRY InMemoryOrderModuleList
00007FFD186F4234	48:8B1C03	mov rbx,qword ptr ds:[rbx+rax]	LIST_ENTRY InMemoryOrderModuleList
00007FFD186F4238	49:89DC	mov r12,rbx	r12 first entry
00007FFD186F423B	48:8B53 40	mov rdx,qword ptr ds:[rbx+40]	Dllbase
00007FFD186F423F	807A 28 4B	cmp byte ptr ds:[rdx+28],4B	byte ptr ds:[rdx+28]:L"ktop\WikiLoader\bef04e3b2b81f
00007FFD186F4243	74 08	je mimetools.7FFD186F424D	
00007FFD186F4245	807A 28 6B	cmp byte ptr ds:[rdx+28],6B	byte ptr ds:[rdx+28]:L"ktop\WikiLoader\bef04e3b2b81f
00007FFD186F4249	74 02	je mimetools.7FFD186F424D	
00007FFD186F424B	EB 46	jmp mimetools.7FFD186F4293	
00007FFD186F424D	807A 30 45	cmp byte ptr ds:[rdx+30],45	byte ptr ds:[rdx+30]:L"\WikiLoader\bef04e3b2b81f2dee
00007FFD186F4251	74 08	je mimetools.7FFD186F425B	
00007FFD186F4253	807A 30 65	cmp byte ptr ds:[rdx+30],65	byte ptr ds:[rdx+30]:L"\WikiLoader\bef04e3b2b81f2dee
00007FFD186F4257	74 02	je mimetools.7FFD186F425B	
00007FFD186F4259	EB 38	jmp mimetools.7FFD186F4293	
00007FFD186F425B	807A 32 4C	cmp byte ptr ds:[rdx+32],4C	byte ptr ds:[rdx+32]:L"WikiLoader\bef04e3b2b81f2dee39
00007FFD186F425F	74 08	je mimetools.7FFD186F4269	
00007FFD186F4261	807A 32 6C	cmp byte ptr ds:[rdx+32],6C	byte ptr ds:[rdx+32]:L"WikiLoader\bef04e3b2b81f2dee39
00007FFD186F4265	74 02	je mimetools.7FFD186F4269	
00007FFD186F4267	EB 2A	jmp mimetools.7FFD186F4293	
00007FFD186F4269	807A 36 32	cmp byte ptr ds:[rdx+36],32	byte ptr ds:[rdx+36]:L"kiLoader\bef04e3b2b81f2dee39c4
00007FFD186F426D	74 02	je mimetools.7FFD186F4271	
00007FFD186F426F	EB 22	jmp mimetools.7FFD186F4293	
00007FFD186F4271	807A 3C 4C	cmp byte ptr ds:[rdx+3C],4C	byte ptr ds:[rdx+3C]:L"oader\bef04e3b2b81f2dee39c42ab
00007FFD186F4275	74 08	je mimetools.7FFD186F427F	
00007FFD186F4277	807A 3C 6C	cmp byte ptr ds:[rdx+3C],6C	byte ptr ds:[rdx+3C]:L"oader\bef04e3b2b81f2dee39c42ab
00007FFD186F427B	74 02	je mimetools.7FFD186F427F	
00007FFD186F427D	EB 14	jmp mimetools.7FFD186F4293	
00007FFD186F427F	807A 3E 4C	cmp byte ptr ds:[rdx+3E],4C	byte ptr ds:[rdx+3E]:L"ader\bef04e3b2b81f2dee39c42ab9
00007FFD186F4283	74 08	je mimetools.7FFD186F428D	
00007FFD186F4285	807A 3C 6C	cmp byte ptr ds:[rdx+3C],6C	byte ptr ds:[rdx+3C]:L"oader\bef04e3b2b81f2dee39c42ab
00007FFD186F4289	74 02	je mimetools.7FFD186F428D	
00007FFD186F428B	EB 06	jmp mimetools.7FFD186F4293	
00007FFD186F428D	48:8B5B 20	mov rbx,qword ptr ds:[rbx+20]	series of jmp and cmp for single letters to validate "kernel32.dll"
00007FFD186F4291	EB 08	jmp mimetools.7FFD186F429B	
00007FFD186F4293	48:8B1B	mov rbx,qword ptr ds:[rbx]	Check if we are back to start of list
00007FFD186F4296	49:39DC	cmp r12,rbx	
00007FFD186F4299	75 A0	jne mimetools.7FFD186F423B	
00007FFD186F429B	48:895D F8	mov qword ptr ss:[rbp-8],rbx	
00007FFD186F429F	4D:31C0	xor r8,r8	
00007FFD186F42A2	44:8B43 3C	mov r8,dword ptr ds:[rbx+3C]	0xF8 RVA PE_SIGNATURE
00007FFD186F42A6	4C:89C2	add rdx,rbx	rdx:L"C:\Windows\System32\Kernel32.DLL"
00007FFD186F42A9	48:01DA	add rdx,rbx	Add 0x F8 to rdx:L"C:\Windows\System32\Kernel32.DLL" to reach PE header
00007FFD186F42AB	44:8B82 88000000	mov r8d,qword ptr ds:[rdx+8]	Value 0x9A370 RVA Export TABLE
00007FFD186F42B3	49:01D8	add r8,rbx	DataDirectory
00007FFD186F42B5	48:31F6	xor rsi,rsi	
00007FFD186F42B9	41:8B70 20	mov esi,dword ptr ds:[r8+20]	Exported function names table
00007FFD186F42BD	48:01DE	add rsi,rbx	
00007FFD186F42C0	48:31C9	xor rcx,rcx	
00007FFD186F42A6	4C:89C2	mov rdx,r8	rdx:L"C:\Users\Dynamic\Desktop\WikiLoader\bef04e3

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.

00007FFD287A4299	75 AU	jne mimetools.7FFD287A423B	
00007FFD287A429B	48:895D F8	mov qword ptr ss:[rbp-8],rbx	
00007FFD287A429F	4D:31C0	xor r8,r8	
00007FFD287A42A2	44:8B43 3C	mov r8,dword ptr ds:[rbx+3C]	0xF8 RVA PE_SIGNATURE
00007FFD287A42A6	4C:89C2	add rdx,rbx	rdx:L"C:\Windows\System32\Kernel32.DLL"
00007FFD287A42A9	48:01DA	add rdx,rbx	Add 0x F8 to rdx:L"C:\Windows\System32\Kernel32.DLL" to reach PE header
00007FFD287A42AB	44:8B82 88000000	mov r8d,qword ptr ds:[rdx+8]	Value 0x9A370 RVA Export TABLE
00007FFD287A42B3	49:01D8	add r8,rbx	DataDirectory
00007FFD287A42B5	48:31F6	xor rsi,rsi	
00007FFD287A42B9	41:8B70 20	mov esi,dword ptr ds:[r8+20]	Exported function names table
00007FFD287A42BD	48:01DE	add rsi,rbx	
00007FFD287A42C0	48:31C9	xor rcx,rcx	

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

Member	Offset	Size	Value	Section
Export Directory RVA	00000180	Dword	0009A370	.rdata
Export Directory Size	00000184	Dword	0000DF0C	
Import Directory RVA	00000188	Dword	000A827C	.rdata
Import Directory Size	0000018C	Dword	00000794	

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 (GetProcAddress)
2. Loops through all the functions and compares the 8 bytes to r9

```

00007FFD1B6F4299 75 A0 jne mimetools.7FFD1B6F423B
00007FFD1B6F429B 48:895D F8 mov qword ptr ss:[rbp-8],rbx
00007FFD1B6F429F 4D:31C0 xor r8,r8
00007FFD1B6F42A2 44:8B43 3C mov r8d,dword ptr ds:[rbx+3C]
00007FFD1B6F42A6 4C:89C2 mov rdx,r8
00007FFD1B6F42A9 48:01DA add rdx,rbx
00007FFD1B6F42AC 44:8B82 88000000 mov r8d,dword ptr ds:[rdx+88]
00007FFD1B6F42B3 49:01D8 add r8,rbx
00007FFD1B6F42B6 48:31F6 xor rsi,rsi
00007FFD1B6F42B9 41:8B70 20 mov esi,dword ptr ds:[r8+20]
00007FFD1B6F42BD 48:01DE add rsi,rbx
00007FFD1B6F42C0 48:31C9 xor rcx,rcx
00007FFD1B6F42C3 49:B9 8CCB6AD43BB19D00 mov r9,19DB13BD46ACB8C
00007FFD1B6F42CD 48:B8 BB99097C368EC53 mov rax,3FC5BE367C09998B
00007FFD1B6F42D7 49:01C1 add r9,rax
00007FFD1B6F42DA 48:FFC1 the rcx
00007FFD1B6F42DD 48:31C0 xor rax,rax
00007FFD1B6F42E0 8B048E mov eax,dword ptr ds:[rsi+rcx*4]
00007FFD1B6F42E3 48:01D8 add rax,rbx
00007FFD1B6F42E6 4C:3908 cmp qword ptr ds:[rax],r9
00007FFD1B6F42E9 75 EF jne mimetools.7FFD1B6F42DA
00007FFD1B6F42EB 48:31F6 xor rsi,rsi
00007FFD1B6F42EE 41:8B70 24 mov esi,dword ptr ds:[r8+24]
00007FFD1B6F42F2 48:01DE add rsi,rbx
00007FFD1B6F42F5 66:8B0C4E mov cx,word ptr ds:[rsi+rcx*2]
00007FFD1B6F42F9 48:31F6 xor rsi,rsi
    
```

```

R9 41636F7250746547
    
```

Address	Hex	ASCII
00007FFD303F285F	47 65 74 50 72 6F 63 41	GetProcAddress.G
00007FFD303F286F	65 74 50 72 6F 65 65 73	etProcessAffinit
00007FFD303F287F	79 4D 61 73 68 00 47 65	yMask.GetProcess
00007FFD303F288F	44 45 50 50 6F 6C 69 63	DEPPolicy.GetPro
00007FFD303F289F	63 65 73 73 44 65 66 61	cessDefaultCpuSe
00007FFD303F28AF	74 73 00 61 70 69 2D 6D	ts.api-ms-win-co
00007FFD303F28BF	72 65 2D 70 72 6F 63 65	re-processthread
00007FFD303F28CF	73 2D 6C 31 2D 31 2D 33	s-11-1-3.GetProc
00007FFD303F28DF	65 73 73 44 65 66 61 75	essDefaultCpuSet
00007FFD303F28EF	73 00 47 65 74 50 72 6F	s.GetProcessGrou
00007FFD303F28FF	70 41 66 66 69 6E 69 74	pAffinity.GetPro
00007FFD303F290F	63 65 73 73 48 61 6E 64	cessHandleCount.
00007FFD303F291F	47 65 74 50 72 6F 63 65	GetProcessHeap.G
00007FFD303F292F	65 74 50 72 6F 63 65 73	etProcessHeaps.G
00007FFD303F293F	65 74 50 72 6F 63 65 73	etProcessId.
00007FFD303F294F	72 6F 63 65 73 73 49 64	rocessIdOfThread
00007FFD303F295F	00 47 65 74 50 72 6F 63	.GetProcessInfor
00007FFD303F296F	6D 61 74 69 6E 6E 00 47	mation.GetProces

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.

```

00007FD288442E8 48:31F6 xor esi,rsi
00007FD288442EE 41:8B70 24 mov esi,dword ptr ds:[r8+24]
00007FD288442F2 49:01DE add rsi,rbx
00007FD288442F5 66:8B0C4E mov cx,word ptr ds:[rsi+rcx*2]
00007FD288442F9 48:31F6 xor rsi,rsi
00007FD288442FC 41:8B70 1C mov esi,dword ptr ds:[r8+1C]
00007FD28844300 48:01DE add rsi,rbx
00007FD28844303 48:31D2 xor rdx,rdx
00007FD28844306 8B148E mov edx,dword ptr ds:[rsi+rcx*4]
00007FD28844309 48:01DA add rdx,rbx
00007FD2884430C 48:8955 F0 mov qword ptr [rbp-10],rdx
00007FD28844310 48:815C 00010000 sub rbp,100
    
```

```

rsi:QuirkIsEnabledForPackage2worker+28A78
AddressOfNameOrdinals Value>9D6A0
rsi:QuirkIsEnabledForPackage2worker+28A78
word ptr ds:[rsi+rcx*2]:QuirkIsEnabledForPackage2worker+28F8
rsi:QuirkIsEnabledForPackage2worker+28A78
AddressOfFunctions Value> 9A398
rsi:QuirkIsEnabledForPackage2worker+28A78
rdx:GetProcAddress
Value 18650 GetProcAddress from Export Table
Adding RVA address found to Base
FetchAddress rdx:GetProcAddress
    
```

kernel32.dll				
Member	Offset	Size	Value	
Characteristics	00099370	Dword	00000000	
TimeStamp	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	000A2882	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.

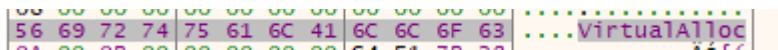


The block of code after this is responsible for the following:

- 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 VirtualAlloc from kernel32.dll

```

00007FFD287A43A0 48:8840 F8      mov rcx,qword ptr ss:[rbp-8]      kernel32.dll pointer
00007FFD287A43A4 48:89FA      mov rdx,rdi                      Moving VirtualAlloc string to rdx
00007FFD287A43A7 48:83EC 20      sub rsc,20
00007FFD287A43AB 48:8D05 07000000  lea rax,qword ptr ds:[7FFD287A43B9] Returning Address setup to after the jmp call Hardcoded return value
00007FFD287A43B2 50          push rax
00007FFD287A43B3 48:8845 F0      mov rax,qword ptr ss:[rbp-10]     rax: r0CAddress
00007FFD287A43B7 FFEB      jmp rax                          GetProcAddress For VirtualAlloc args1:RCX arg2:RDX
00007FFD287A43B9 48:83C4 20      add rsp,20
    
```



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

```

00007FFD287A43B7 FFEB      jmp rax                          GetProcAddress For VirtualAlloc args1:RCX arg2:RDX
00007FFD287A43B9 48:83C4 20      add rsp,20
00007FFD287A43BD 49:89C7      mov r15,rax                      Saving VirtualAlloc Address function to r15
00007FFD287A43C0 48:C7C1 00000000  mov rcx,0
00007FFD287A43C7 48:C7C2 00E1F505  mov rdx,5F5E100
00007FFD287A43CE 49:C7C0 00300000  mov r8,3000                      Args
00007FFD287A43D5 49:C7C1 04000000  mov r9,4
00007FFD287A43DC 48:83EC 20      sub rsp,20
00007FFD287A43E0 48:8D05 04000000  lea rax,qword ptr ds:[7FFD287A43E8] Returning Address setup to after the jmp call
00007FFD287A43E7 50          push rax                          rax:5E1F0A1EEB
00007FFD287A43E8 41:FFEB      jmp r15                          VirtualAlloc call arg1:RCX=0 arg2:RDX arg3:r8=3000 arg4:r9=4
00007FFD287A43EB 48:83C4 20      add rsp,20
    
```

Allocated memory buffer:



Through 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],a1 | byte ptr ds:[rdi+11]:"Q\
00007FFD287A4487 647 12 00 | mov byte ptr ds:[rdi+12],0 | Second String: GetCurrentThreadId
00007FFD287A448B 48:8B4D F8 | mov rcx,qword ptr ss:[rbp-8] | rcx:LocalCatevH_LuaMemory+14
00007FFD287A448F 48:89FA | mov rdx,rdi | rdi:"GetCurrentThreadId\
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]]:GetProcAdres
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:[7FFD287A448A] |
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+09],a1 |
00007FFD287A4506 647 0A 00 | mov byte ptr ds:[rdi+10],0 | OpenThread
00007FFD287A450A 48:8B4D F8 | mov rcx,qword ptr ss:[rbp-8] | rdi:"OpenThread"
00007FFD287A450E 48:89FA | mov rdx,rdi |
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 40:89E0 | 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.



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.

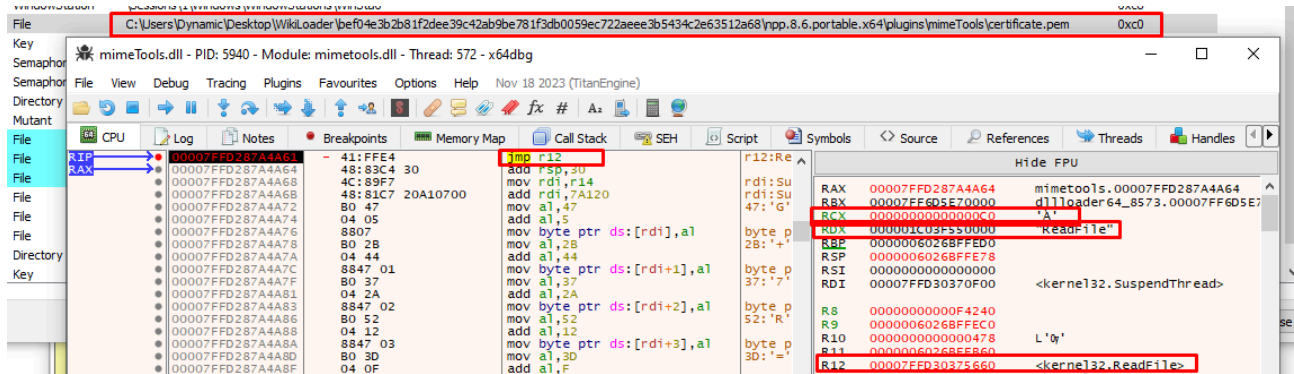
```

00007FFD287A4985 - 41:FFE4 | jmp r12 |
00007FFD287A4988 48:83C4 40 | add rsp,40 |
00007FFD287A498C 48:83F8 FF | cmp rax,FFFFFFFFFFFFFFFF |
00007FFD287A4990 0F84 BA130000 | je mimetools.7FFD287ASD80 |
00007FFD287A4996 80 11 | mov al,11 |
00007FFD287A499C 04 41 | add al,41 |
00007FFD287A499E 41:8806 | mov byte ptr ds:[r14],al |
00007FFD287A49D1 80 45 | mov al,45 |
00007FFD287A49D3 04 20 | add al,20 |
00007FFD287A49D5 41:8846 01 | mov byte ptr ds:[r14+1],al |
00007FFD287A49D9 80 36 | mov al,36 |
00007FFD287A49DB 04 28 | add al,28 |
00007FFD287A49DD 41:8846 02 | mov byte ptr ds:[r14+2],al |
00007FFD287A49E1 80 33 | mov al,33 |
00007FFD287A49E3 04 31 | add al,31 |
00007FFD287A49E5 41:8846 03 | mov byte ptr ds:[r14+3],al |
00007FFD287A49E9 41:57 | push r15 |
    
```

Hide FPU		
RAX	00007FFD287A4988	mimetools.00007FFD287A
RBX	00007FF6D5E70000	dllloader64.8573.00007
RCX	000001C03F550000	"certificate.pem"
RDX	0000000080000000	
RBP	00000060268FFED0	
RSP	00000060268FFE68	
RSI	0000000000000000	
RDI	00007FFD30370F00	<kernel32.SuspendThrea
R8	0000000000000000	
R9	0000000000000000	
R10	00000000000000C6	'&
R11	00000060268FFB60	
R12	00007FFD303752D0	<kernel32.CreateFileA>

For ReadFile the two arguments correspond to:

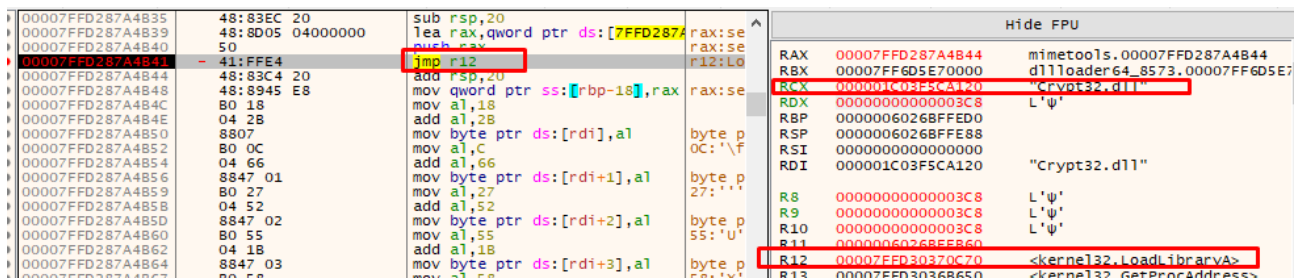
- File Handle
- Memory buffer to store data that is read.



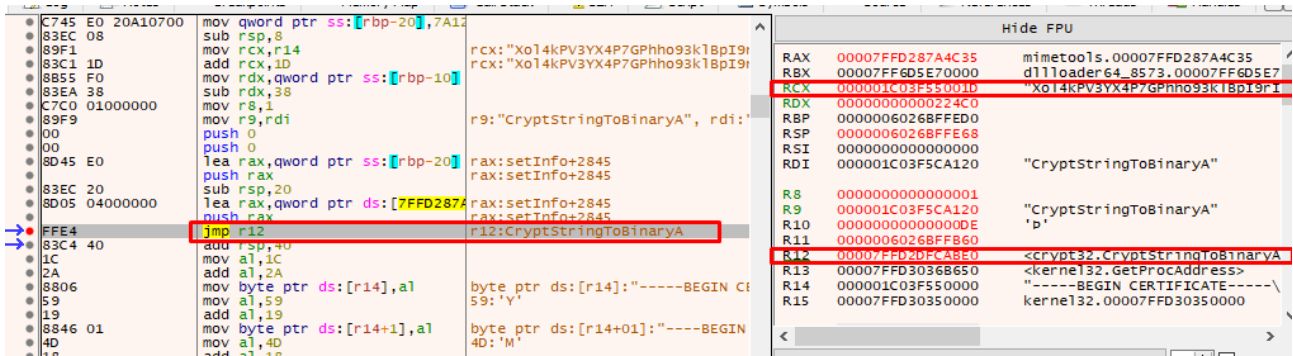
Address	Hex	ASCII
000001C03F550000	2D 2D 2D 2D 2D 42 45 47 49 4E 20 43 45 52 54 49	-----BEGIN CERTI
000001C03F550010	46 49 43 41 54 45 2D 2D 2D 2D 2D 0D 0A 58 6F 6C	FICATE-----..Xo1
000001C03F550020	34 68 50 56 33 59 58 34 50 37 47 50 68 68 6F 39	4kPV3YX4P7GPhho9
000001C03F550030	33 68 6C 42 70 49 39 72 49 50 44 77 41 34 30 78	3k1BpI9rIPDwA40x
000001C03F550040	50 44 55 30 68 28 69 46 6E 32 46 76 68 6E 41 6C	PDU0h+iFn2FvhnA1
000001C03F550050	6F 6E 39 68 52 42 65 66 48 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	VBpasYJCbmbjzwx8
000001C03F550080	4E 70 37 4E 55 75 39 57 69 28 70 30 68 52 6A 75	Np7NUu9Wi+p0hRju
000001C03F550090	55 64 31 68 66 49 39 6F 36 33 48 68 35 77 78 49	Ud1hfI9o63HksWxI
000001C03F5500A0	4E 36 4F 6A 72 37 51 65 32 4D 7A 78 65 66 50 62	N60jr7Qe2MzxfPb
000001C03F5500B0	48 4A 4C 78 4C 47 4C 5A 47 43 76 6A 61 34 5A 57	HJLXLGLZGCvjaZW
000001C03F5500C0	63 6F 34 76 35 69 77 50 37 48 73 6F 51 48 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	bIUD6VvAwVyxfzn
000001C03F5500F0	6C 4E 56 33 79 51 2B 62 68 7A 6F 73 64 55 30 54	1NV3yQ+bhzosDU0T
000001C03F550100	39 4E 67 5A 46 71 2F 45 7A 50 74 6E 46 68 76 49	9NgZFq/EzPtnFkvI
000001C03F550110	64 4E 6C 42 70 49 6D 53 64 55 41 31 48 66 6F 6E	dn1BpImSdUA1Kfon
000001C03F550120	52 44 2F 73 72 6F 44 74 45 30 48 69 39 48 67 45	RD/sroDtEOH19Hge
000001C03F550130	42 43 28 68 51 6C 34 44 34 64 64 5A 77 74 54 46	BC+kq14D4ddZwtTF
000001C03F550140	36 77 39 63 47 55 62 67 35 30 28 30 50 5A 75 35	6w9cGubg50+OPZu5
000001C03F550150	66 4F 57 53 57 47 53 59 78 42 53 4E 68 4E 72 61	FOWSWSGYxB5NHNrA

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:

000001C03F5CA120	5E 89 78 90	F5 77 61 7E	0F EC 63 E1	86 8F 77 92	À. x. òwa~. ì cá. .w.
000001C03F5CA130	50 69 23 DA	C8 3C 3C 00	E3 4C 4F 0D	4D 21 FA 21	Pi#ÙÉ<<. àLO.M!ú!
000001C03F5CA140	67 D8 5B E1	9C 09 68 9F	D8 51 05 E7	CA EB 04 C4	gò[á. .h.ò. çÈÈ.A
000001C03F5CA150	ED C0 FC 11	3A F2 64 CA	DD E0 0A 18	30 3C 15 06	ìÀü. :òdÈYà. .0<..
000001C03F5CA160	96 AC 60 90	98 98 96 F3	C1 7F 0D A7	B3 54 8B D5	.-óÁ. . \$*T»Ó
000001C03F5CA170	A2 FA 9D 21	46 3B 94 77	58 5F 23 DA	3A DC 79 39	éú. !F; wx_#ú: Úy9
000001C03F5CA180	C3 12 0D E8	E8 EB ED 07	B6 33 3C 5E	7C F6 C7 24	À. éèèì. η3<^ òç\$
000001C03F5CA190	BC 4B 18 B6	46 0A F8 DA	E1 95 9C A3	8B F9 8B 03	%K. ηF. óÚá. . é. ú. .
000001C03F5CA1A0	FB 1E CA 10	29 B4 66 EB	FD 03 1E DE	1A DE FA 58	ù. È.) fèy. .p. pUX
000001C03F5CA1B0	04 A3 9B 21	40 FA 56 F0	30 57 2C 45	7F 39 E5 35	. é. !@úVòW. E. 9ás
000001C03F5CA1C0	5D F2 43 E6	E1 CE 88 1D	53 44 FD 36	06 45 AB F1	ìòcæáì. .SDy6.E«ñ
000001C03F5CA1D0	33 3E D9 C5	92 F2 1D 36	50 69 22 64	9D 50 0D 4A	3>ÚÁ. ò. 6Pi" d. P. J
000001C03F5CA1E0	7E 89 D1 0F	F8 2B A0 38	44 D0 78 BD	1E 01 01 08	~. Ñ. ù+ ;Dòx)%... .
000001C03F5CA1F0	E9 10 97 80	F8 75 D6 70	B5 31 7A C3	D7 06 51 B8	é.òüòppuzÁx. Q.
000001C03F5CA200	39 D3 ED 0F	66 EE 5F 39	64 96 19 26	31 05 23 61	9óì. fì_9d. .&1.#a
000001C03F5CA210	36 B6 90 01	DD 73 77 81	D8 9E 61 38	E5 24 B5 E7	6η. .Ýsw. ø. a; àšuc
000001C03F5CA220	D5 53 1C 9C	C5 86 1F EB	F4 2C 14 2B	47 87 D5 66	òS. .Á. .èò. .(G.òç
000001C03F5CA230	DE 3A FD 1C	10 8D 62 E7	15 E5 C6 8F	C1 EF DC FB	p: y. . .bc. àé. ÁìÚÙ
000001C03F5CA240	8A 8E 4B 0C	13 E7 2A 3D	A6 85 80 F9	AA E5 FF 65	. .K. .ç*=: . .ùªàye
000001C03F5CA250	69 C3 16 04	31 8C F2 41	18 3C BD AD	52 BA 79 02	ìÁ. .1. òA. <%R. °y.
000001C03F5CA260	35 F3 52 8E	F7 FC C8 49	9A E1 2B C6	B8 2A 69 3D	5òR. +ùÈì. à+È. *ì=
000001C03F5CA270	A7 D1 69 0A	2F 42 D8 26	5B 08 99 10	27 D8 02 70	§Ni. /Bò&[... 'ò. p

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

- ObjectLength
- BlockLength
- ChainingMode
- ChainingModeCBC

000001C0454B2710	4F 00 62 00	6A 00 65 00	63 00 74 00	4C 00 65 00	0. b. j. e. c. t. L. e.
000001C0454B2720	6E 00 67 00	74 00 68 00	00 00 00 00	00 00 00 00	n. g. t. h.
000001C0454B2730	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B2740	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B2750	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B2760	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B2770	00 00 00 00	42 00 6C 00	6F 00 63 00	68 00 4C 00B. l. o. c. k. L.
000001C0454B2780	65 00 6E 00	67 00 74 00	68 00 00 00	00 00 00 00	e. n. g. t. h.
000001C0454B2790	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B27A0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B27B0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B27C0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B27D0	00 00 00 00	00 00 00 00	43 00 68 00	61 00 69 00C. h. a. i.
000001C0454B27E0	6E 00 69 00	6E 00 67 00	4D 00 6F 00	64 00 65 00	n. i. n. g. M. o. d. e.
000001C0454B27F0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B2800	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B2810	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B2820	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
000001C0454B2830	00 00 00 00	00 00 00 00	00 00 00 00	43 00 68 00C. h.
000001C0454B2840	61 00 69 00	6E 00 69 00	6E 00 67 00	4D 00 6F 00	a. i. n. i. n. g. M. o.
000001C0454B2850	64 00 65 00	43 00 42 00	43 00 00 00	00 00 00 00	d. e. C. B. C.
000001C0454B2860	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

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

<pre> mov byte ptr ds:[rdi+A],0 mov rcx,rdi sub rsp,20 mov rax,qword ptr ss:[rbp-E8] call rax add rsp,20 mov r15,rax mov al,21 add al,21 mov byte ptr ds:[rdi],al mov al,2A add al,19 mov byte ptr ds:[rdi+1],al mov al,F add al,63 </pre>	<pre> rcx:"bc" rax:Loa rax:Loa rax:Loa 21:'!' byte pt 2A:'*' byte pt </pre>	<p>Hide FPU</p> <table border="1"> <tr><td>RAX</td><td>00007FFD30370C70</td><td><kernel32.LoadLibraryA></td></tr> <tr><td>RBX</td><td>00007FFD30350000</td><td>kernel32.00007FFD30350000</td></tr> <tr><td>RCX</td><td>00000060268FFA40</td><td>"bcrypt.dll"</td></tr> <tr><td>RDX</td><td>0000000000000653</td><td>L''</td></tr> <tr><td>RBP</td><td>00000060268FFC48</td><td></td></tr> <tr><td>RSP</td><td>00000060268FFA10</td><td></td></tr> <tr><td>RSI</td><td>0000000000000000</td><td></td></tr> <tr><td>RDI</td><td>00000060268FFA40</td><td>"bcrypt.dll"</td></tr> <tr><td>R8</td><td>0000000000000653</td><td>L''</td></tr> <tr><td>R9</td><td>0000000000000653</td><td>L''</td></tr> </table>	RAX	00007FFD30370C70	<kernel32.LoadLibraryA>	RBX	00007FFD30350000	kernel32.00007FFD30350000	RCX	00000060268FFA40	"bcrypt.dll"	RDX	0000000000000653	L''	RBP	00000060268FFC48		RSP	00000060268FFA10		RSI	0000000000000000		RDI	00000060268FFA40	"bcrypt.dll"	R8	0000000000000653	L''	R9	0000000000000653	L''
RAX	00007FFD30370C70	<kernel32.LoadLibraryA>																														
RBX	00007FFD30350000	kernel32.00007FFD30350000																														
RCX	00000060268FFA40	"bcrypt.dll"																														
RDX	0000000000000653	L''																														
RBP	00000060268FFC48																															
RSP	00000060268FFA10																															
RSI	0000000000000000																															
RDI	00000060268FFA40	"bcrypt.dll"																														
R8	0000000000000653	L''																														
R9	0000000000000653	L''																														

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.

RAX	00007FFD2E5D51E0	<bcrypt.BCryptOpenAlgorithmProvider>
RBX	00007FFD30350000	kernel32.00007FFD30350000
RCX	00000060268FFC40	
RDX	00000060268FFC38	L"AES"
RBP	00000060268FFC48	
RSP	00000060268FFB10	
RSI	0000000000000000	
RDI	00000060268FFA40	"BCryptCloseAlgorithmProvider"
R8	0000000000000000	
R9	0000000000000000	

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	000001C03DCC8B0	
RDX	000001C0454B2710	L"ObjectLength"
RBP	00000060268FFC48	
RSP	00000060268FFB00	
RSI	0000000000000000	
RDI	00000060268FFA40	"BCryptCloseAlgorithmProvider"
R8	00000060268FFC30	
R9	0000000000000008	

```

RAX 00007FFD2E5D3E90 <bcrypt.BCryptGetProperty>
RBX 00007FFD30350000 kernel32.00007FFD30350000
RCX 000001C03DCCCB80
RDX 000001C0454B2774 L"BlockLength"
RBP 000006026BFFC48
RSP 000006026BFFAA0
RSI 0000000000000000
RDI 000006026BFFA40

R8 000006026BFFC18
R9 0000000000000008
    
```

mimeTools.dll - PID: 5940 - Module: mimetools.dll - Thread: 572 - x64dbg

File View Debug Tracing Plugins Favourites Options Help Nov 18 2023 (TitanEngine)

CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles

Hide FPU

```

RAX 0000000000000017 Return value
RBX 00007FFD30350000 kernel32.00007FFD30350000
RCX 000001C03F5CA120
RDX 00000000FFFFFFFF
RBP 000006026BFFC48
RSP 000006026BFFAA0
RSI 0000000000000000
RDI 000006026BFFA40

R8 000006026BFFC28
R9 0000000000000008
R10 00007FFD2E5E8EE8 L"MultiObjectLength"
R11 000001C03DC72600
R12 000001C03F5CA130
R13 0000000000019880
R14 000001C045480000
R15 00007FFD2E5D0000 bcrypt.00007FFD2E5D0000
    
```

Default (x64 fastcall) 5 Unlocked

```

1: rcx 000001C03F5CA120 000001C03F5CA120
2: rdx 00000000FFFFFFFF 00000000FFFFFFFF
3: r8 000006026BFFC28 000006026BFFC28
4: r9 0000000000000008 0000000000000008
5: rbp+28 000001C03F5CA130 000001C03F5CA130
    
```

Address Hex ASCII

```

000001C03F5CA0F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001C03F5CA100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001C03F5CA110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001C03F5CA120 5E 89 78 90 F5 77 61 25 0F EC 63 E1 86 8F 77 92 7xOwbyTcaWu
000001C03F5CA130 60 69 23 DA C8 3C 3C 00 E3 4C 4F 0D 4D 21 FA 21 P!UE<<.ALO.Mtu!
000001C03F5CA140 6D D8 58 E1 9C 09 68 9F D8 51 05 E7 CA EB 04 C4 g0!a..h.oq.cEe.A
000001C03F5CA150 ED C0 FC 11 3A F2 64 CA DD E0 0A 18 30 3C 15 06 1Au..odEYa..0c..
000001C03F5CA160 96 AC 60 90 98 98 96 F3 C1 7F 0D A7 B3 54 B8 D5 .~...0A..$*TxD
000001C03F5CA170 A2 FA 90 21 46 38 94 77 58 5F 23 DA 3A DC 79 39 cu.lF..wK.#U:Uy9
000001C03F5CA180 C3 12 0D E8 E8 EB 0D 87 B6 33 3C 5E 7C F6 C 24 A..eeei.73<A!OC$
000001C03F5CA190 BC 48 18 B6 46 0A F8 DA E1 95 9C A3 88 F9 88 03 4k.rf.oUa..f.u.
000001C03F5CA1A0 FB 1E CA 10 29 84 66 EB FD 03 1E DE 1A DE FA 58 u.E.)fey..p.duX
000001C03F5CA1B0 04 A3 98 21 40 FA 56 F0 30 57 2C 45 7F 39 E5 5 .:!@v0w.e.945
    
```

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused Dump: 000001C03F5CA120 -> 000001C03F5CA136 (0x00000017 bytes) Time Wasted Debugging: 0:06:33:43

Istrlenw() is called to count "ChainingModeCBC" string. This is necessary for the next call, which is BCryptSetProperty(). The arguments 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.Istrlenw>
RBX 00007FFD30350000 kernel32.00007FFD30350000
RCX 000001C0454B2774 L"ChainingModeCBC"
RDX 0000000000000000
RBP 000006026BFFC48
RSP 000006026BFFAA0
RSI 000001C03F5CA130
RDI 000001C03DC80010

R8 000006026BFFA08
R9 000006026BFFC48
    
```



```

C++

NTSTATUS BCryptDecrypt(
[in, out]          BCRYPT_KEY_HANDLE hKey,
[in]              PCHAR              pbInput,
[in]              ULONG              cbInput,
[in, optional]   VOID               *pPaddingInfo,
[in, out, optional] PCHAR          pbIV,
[in]              ULONG              cbIV,
[out, optional]  PCHAR              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.

The screenshot shows the assembly code for the `BCryptDecrypt` function. Red boxes highlight the first four arguments: `mov rcx,qword ptr ss:[rbp-50]`, `mov rdx,qword ptr ss:[rbp-70]`, `mov r8,qword ptr ss:[rbp-68]`, and `mov r9,qword ptr ss:[rbp-68]`. Another set of red boxes highlights the last six arguments: `push rax`, `push rcx`, `push rdx`, `push r8`, `push r9`, and `push rax`. The register window on the right shows the values for RAX, RBX, RCX, RDX, R8, and R9.

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 to the sixth argument.

The screenshot shows a memory dump with hex and ASCII views. Red boxes highlight the IV data at address `0000006026BFFA20` and the stack pointer `RBP` at `0000006026BFFA48`.

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.

Address	Hex	ASCII
000001C03DC90000	50 48 31 C0 48 FF C0 48 85 C0 0F 84 AA 89 01 00	PHIAÿAH.A...*
000001C03DC90010	58 51 48 31 C9 48 85 C9 0F 85 1F 85 01 00 48 FF	XQH1ÉH.É.....ÿ
000001C03DC90020	C1 48 85 C9 0F 84 A0 8A 01 00 59 E9 A9 82 01 00	ÀH.É...Yé@...
000001C03DC90030	8C 8A 8E 8F 98 8D 98 8C 98 8F 98 8E 91 50 3C 91P<.
000001C03DC90040	5A 35 D1 30 77 58 D8 D9 8C 8A 8E 8F 98 8D 98 8C	Z5ÑowX0Ù.....
000001C03DC90050	98 8F 98 8E 91 50 3C 91 5A 35 D1 91 5A 35 D1 91P<.Z5Ñ.Z5Ñ.
000001C03DC90060	5A 35 E1 91 50 12 91 E8 19 91 50 30 91 F0 38 91	Z5á.P..è..P0.08.
000001C03DC90070	50 3E 2A 73 91 50 00 91 50 94 21 91 50 8C 29 95	P>*s.P..P.!..P.).
000001C03DC90080	50 9C 31 95 50 94 01 95 50 84 09 95 50 8C 11 91	P..P...P...P...P...
000001C03DC90090	52 5D FD 51 D9 D9 D9 91 50 9C 39 91 54 5D FD 51	R]ýqÙÙÙ.P.9.T]ýq
000001C03DC900A0	D9 D9 D9 91 E8 10 91 50 D1 60 00 66 96 51 91 1E	ÙÙÙ.e..PÑ..f.Q..
000001C03DC900B0	1B D5 D9 D9 D9 90 1E 19 CE D9 D9 D9 90 1E 18 ED	.0ÙÙÙ...ÍÙÙÙ...í
000001C03DC900C0	D9 D9 D9 91 5A 35 F9 91 54 DC DE D9 D9 D9 89 91	ÙÙÙ.Z5ù.TÙpÙÙÙ..
000001C03DC900D0	52 9C 01 26 39 91 5A 1D F9 90 50 1F 94 50 1C 91	R..&9.Z.ù.P..P..
000001C03DC900E0	52 94 09 91 52 8C 21 95 52 9C 31 95 52 94 29 B3	R...R.!..R.1.R.)*
000001C03DC900F0	D9 B3 D9 91 5A 35 F9 91 52 9C 11 89 91 54 DC D7	Ù*Ù.Z5ù.R....TÙx

Address	Hex	Disassembly
000001C03DC90000	50	push rax
000001C03DC90004	48:31C0	xor rax,rax
000001C03DC90004	48:FFC0	inc rax
000001C03DC90007	48:85C0	test rax,rax
000001C03DC9000A	0F84 AA890100	je 1C03DCA89BA
000001C03DC90010	58	pop rax
000001C03DC90011	51	push rcx
000001C03DC90012	48:31C9	xor rcx,rcx
000001C03DC90015	48:85C9	test rcx,rcx
000001C03DC90018	0F85 1F850100	jne 1C03DCA853D
000001C03DC9001E	48:FFC1	inc rcx
000001C03DC90021	48:85C9	test rcx,rcx
000001C03DC90024	0F84 A08A0100	je 1C03DCA8ACA
000001C03DC9002A	59	pop rcx
000001C03DC90028	E9 A9820100	jmp 1C03DCA82D9
000001C03DC90030	8C8A 8E8F988D	mov word ptr ds:[rdx-72677072]
000001C03DC90036	98	cwde
000001C03DC90037	8C98 8F988E91	mov word ptr ds:[rax-6E716771]
000001C03DC9003D	50	push rax
000001C03DC9003E	3C 91	cmp al,91
000001C03DC90040	5A	pop rdx
000001C03DC90041	35 D1307758	xor eax,587730D1
000001C03DC90046	D8D9	fcomp st(1)
000001C03DC90048	8C8A 8E8F988D	mov word ptr ds:[rdx-72677072]
000001C03DC9004E	98	cwde
000001C03DC9004F	8C98 8F988E91	mov word ptr ds:[rax-6E716771]
000001C03DC90055	50	push rax
000001C03DC90056	3C 91	cmp al,91

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!

Hide FPU

RAX	00007FFD287A4ECB	mimetools.00007FFD287A4
RBX	00007FFD5E70000	dllloader64_8573.00007F
RCX	000001C03F550000	
RDX	000000000019B79	
RBP	0000006026BFFED0	
RSP	0000006026BFF78	
RSI	0000000000000000	
RDI	000001C03F5CA120	"VirtualProtect"
R8	0000000000000020	Read access
R9	0000006026BFFEA8	
R10	00000000000000DF	L'1'
R11	0000006026BFFB60	
R12	00007FFD3036C3F0	<kernel32.Virtual Protec
R13	00007FFD30368650	<kernel32.GetProcAdres
R14	000001C03F550000	

Default (x64 fastcall) 5 Unlocked

1:	rcx	000001C03F550000	000001C03F550000
2:	rdx	00000000000019B79	00000000000019B79
3:	r8	0000000000000020	0000000000000020
4:	r9	0000006026BFFEA8	0000006026BFFEA8
5:	[rsi+28]	0000006026BFFED0	0000006026BFFED0

Address	Hex	ASCII
001C03F550000	50 48 31 C0	PHIAHYA
001C03F550010	58 51 48 31	XQHIEH.
001C03F550020	C1 48 85 C9	AH.E...
001C03F550030	8C 8A 8E 8F	ZS.NOWXK
001C03F550040	9A 35 D1 30P<
001C03F550050	98 8F 98 8EP..
001C03F550060	5A 35 E1 91	ZS.A.P..

RAX	0000006026BFFED0	
RSP	0000006026BFFEB0	
RSI	0000000000000000	
RDI	000001C03F5CA120	"VirtualProtect"
R8	0000006026BFFED8	
R9	0000000000000020	
R10	0000000000000000	
R11	0000000000000246	L'Z'
R12	00007FFD3036C3F0	<kernel32.Virtual Protec
R13	00007FFD30368650	<kernel32.GetProcAdres
R14	000001C03F550000	
RIP	00007FFD287A4ED3	mimetools.00007FFD287A4

Default (x64 fastcall) 5 Unlocked

1:	rcx	00007FFD3072DA64	ntd11.00007FFD3072DA64
2:	rdx	0000000000000000	0000000000000000
3:	r8	0000006026BFFED8	0000006026BFFED8
4:	r9	0000000000000020	0000000000000020
5:	[rsi+28]	0000000000000000	0000000000000000

Address	Hex	ASCII
001C03F550000	50 48 31 C0	PHIAHYA
001C03F550010	58 51 48 31	XQHIEH.
001C03F550020	C1 48 85 C9	AH.E...
001C03F550030	8C 8A 8E 8F	ZS.NOWXK
001C03F550040	9A 35 D1 30P<
001C03F550050	98 8F 98 8EP..
001C03F550060	5A 35 E1 91	ZS.A.P..

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

Notes

000001C03F550000	50	push rax
000001C03F550001	48:31C0	xor rax,rax
000001C03F550004	48:FFC0	inc rax
000001C03F550007	48:85C0	test rax,rax
000001C03F55000A	0F84 AA890100	je IC03F5689BA
000001C03F550010	58	pop rax
000001C03F550011	51	push rcx
000001C03F550012	48:31C9	xor rcx,rcx
000001C03F550015	48:85C9	test rcx,rcx
000001C03F550018	0F85 1F850100	jne IC03F56853D
000001C03F55001E	48:FFC1	inc rcx
000001C03F550021	48:85C9	test rcx,rcx
000001C03F550024	0F84 A08A0100	je IC03F568ACA
000001C03F55002A	59	pop rcx
000001C03F550028	E9 A9820100	jmp IC03F5682D9
000001C03F550030	8C8A 8E8F988D	mov word ptr ds:[rdx-72677072]
000001C03F550036	98	cwde
000001C03F550037	8C98 8F988E91	mov word ptr ds:[rax-6E716771]
000001C03F550038	FA	push rax

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

References

- <https://bazaar.abuse.ch/sample/bef04e3b2b81f2dee39c42ab9be781f3db0059ec722aeec3b5434c2e63512a68/>

- <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>

Source: <https://github.com/VenzoV/MalwareAnalysisReports/blob/main/WikiLoader/WikiLoader%20notepad.md>