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

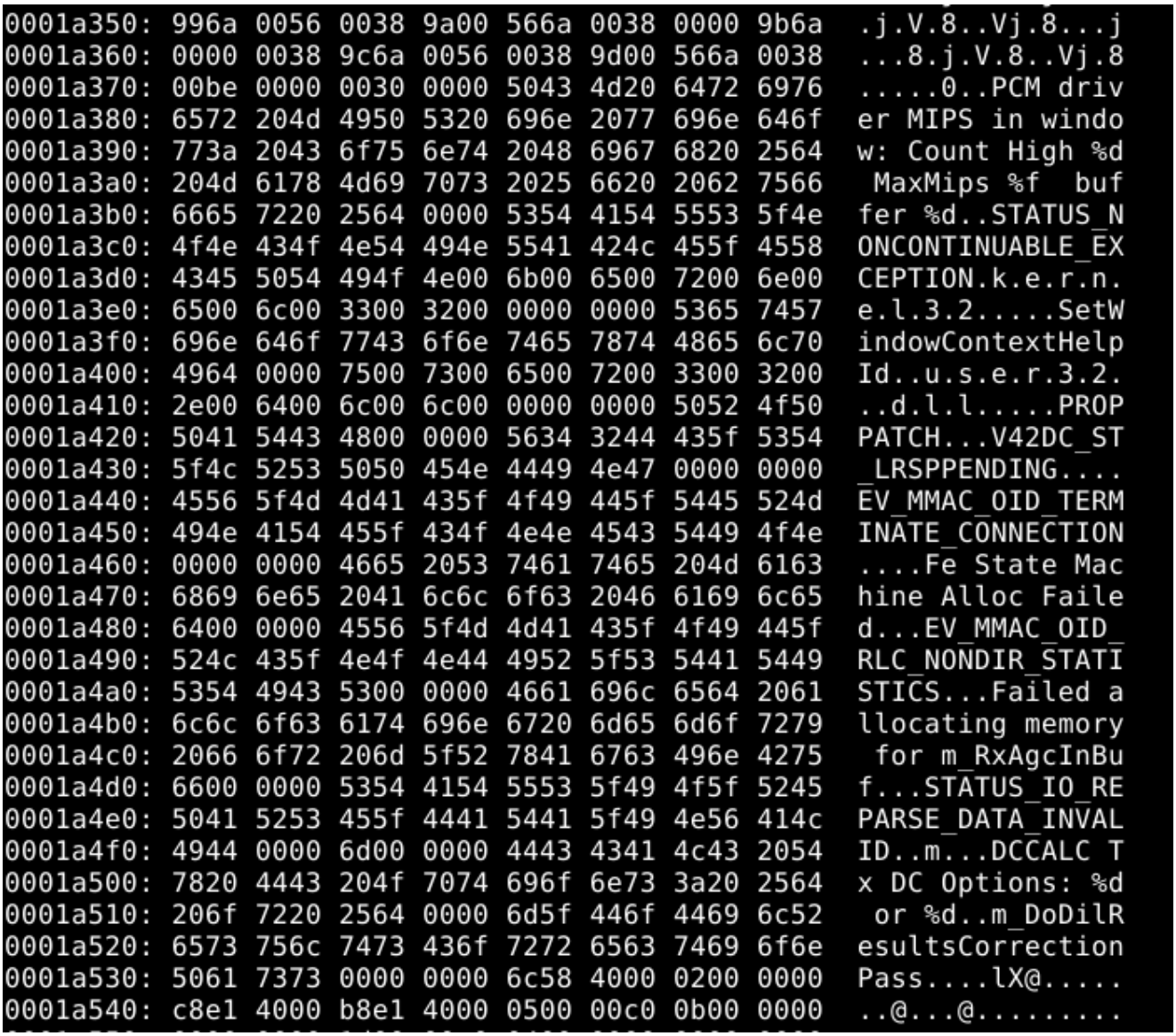
One of the very first steps an analyst should do when first interacting with a potentially malicious executable is to perform basic static analysis on the PE, e.g., look at its PE headers, sections, imports, strings, or any other information that can help him get an overall general idea of what the binary 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?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
EDD0	KERNEL32.dll	80	FALSE	EE0C	0	0	F104	E000
EDE4	USER32.dll	2	FALSE	EF50	0	0	F136	E144

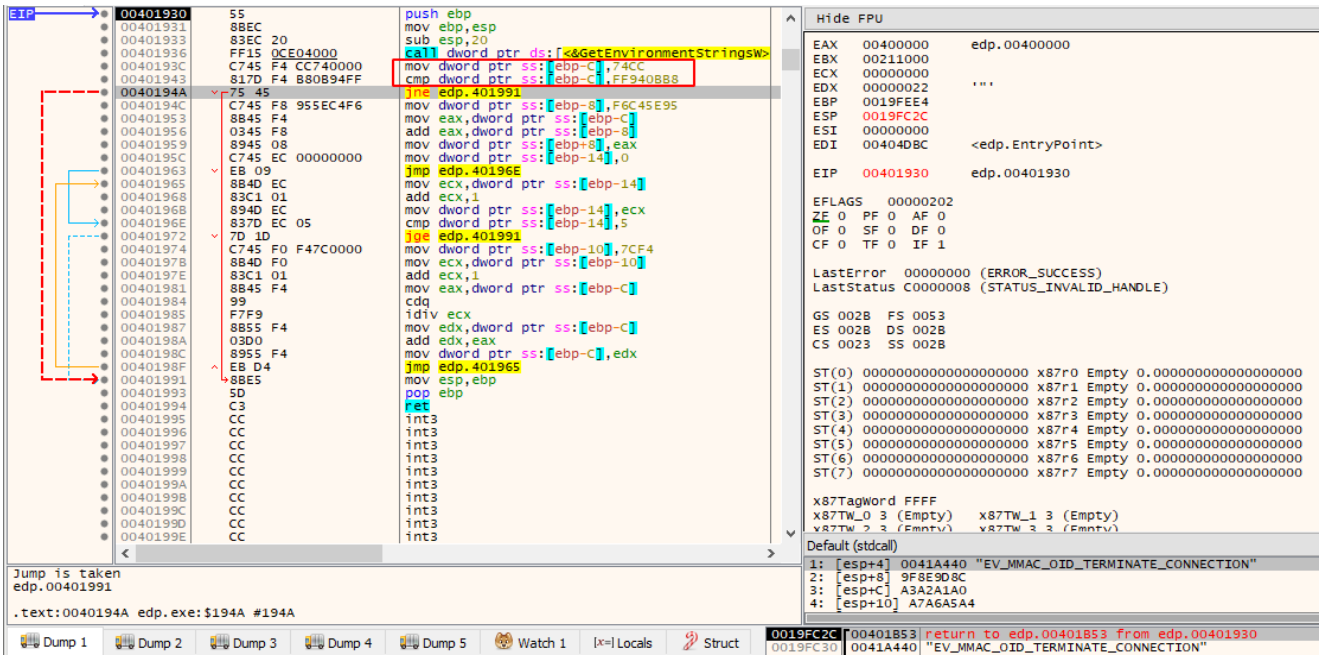
KERNEL32.dll [80 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
E01C	GetProcAddress	-	EFD2	EFD2	-	245
E020	GetCommandLineA	-	EFE4	EFE4	-	186
E024	LoadLibraryW	-	EFF6	EFF6	-	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

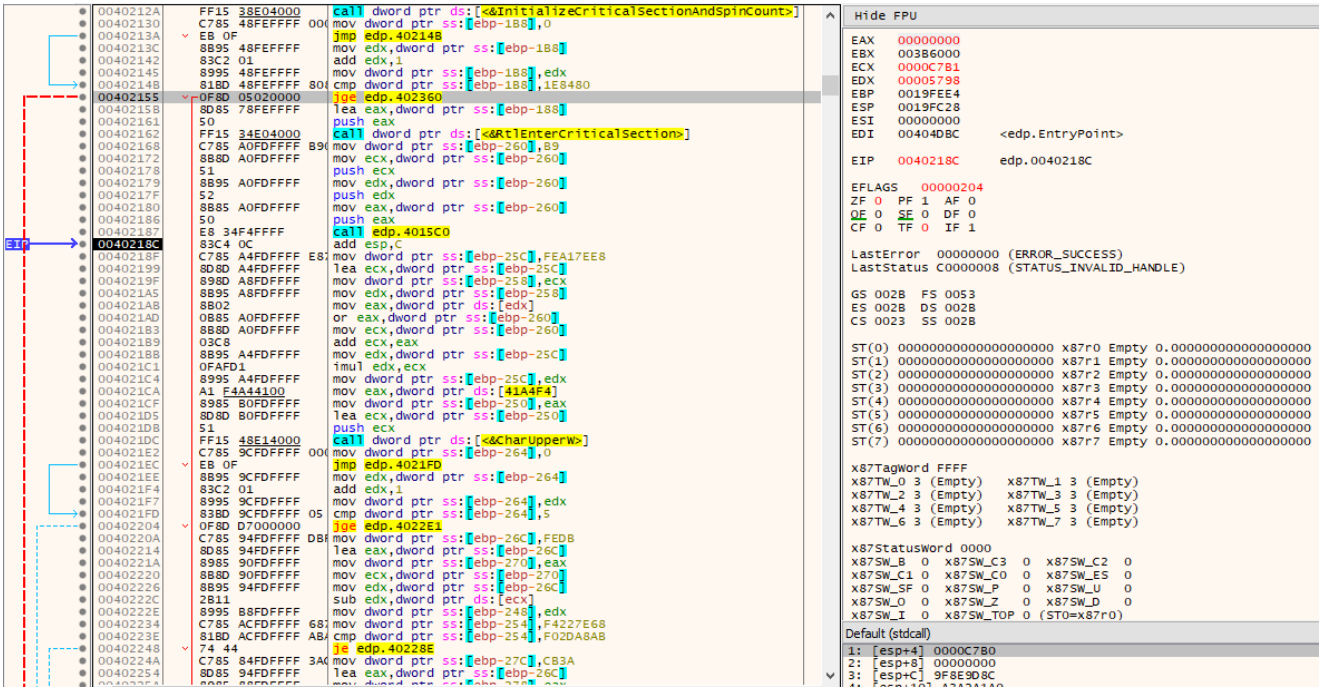
Hex dumping our target shows several strings, of which some, after looking them up on google, reveals pages related to malware and malware analysis services.



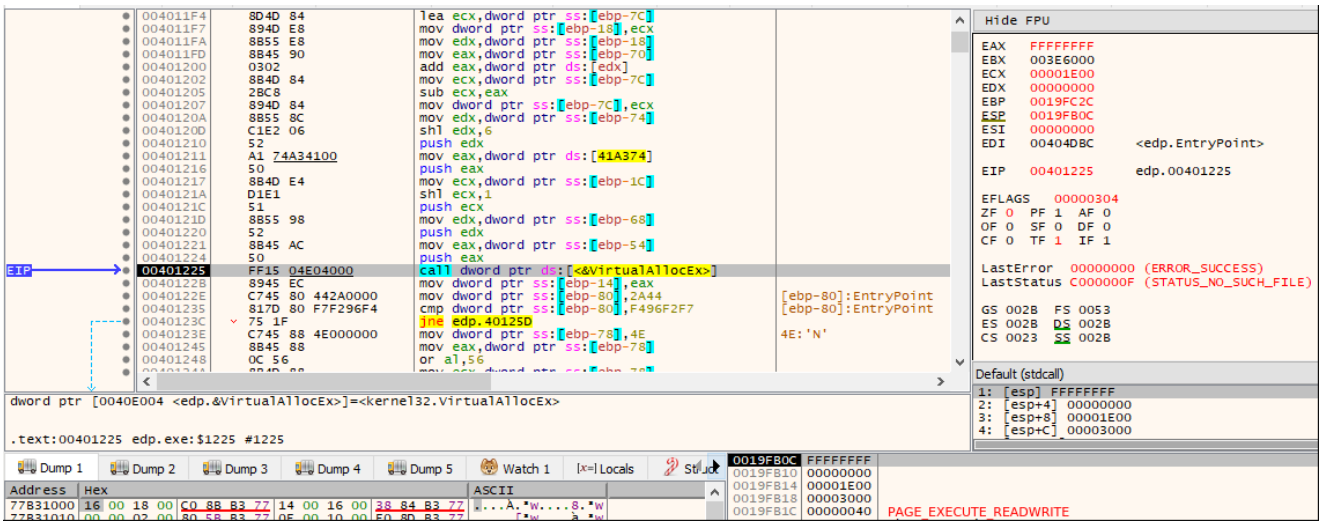
Once a more in depth analysis takes place, it becomes quite clear that the ransomware is, in some way, obfuscated. For example, and for demonstration purposes, the following image displays a function that is called with the string "EV_MMAC_OID_TERMINATE_CONNECTION" as argument, where the string is never 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).



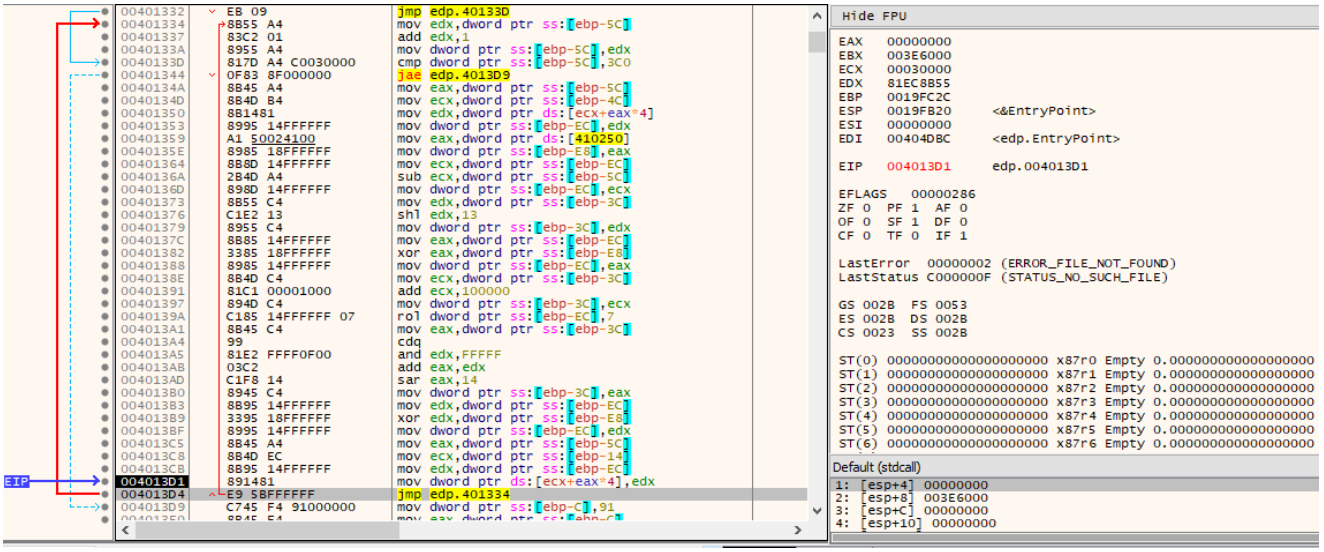
The thread synchronization APIs mentioned earlier also take part of the obfuscation (essentially junk code), where a loop is concluded after being executed 2000000 times, performing useless arithmetic operations and calling a function which always returns 0 along the way.



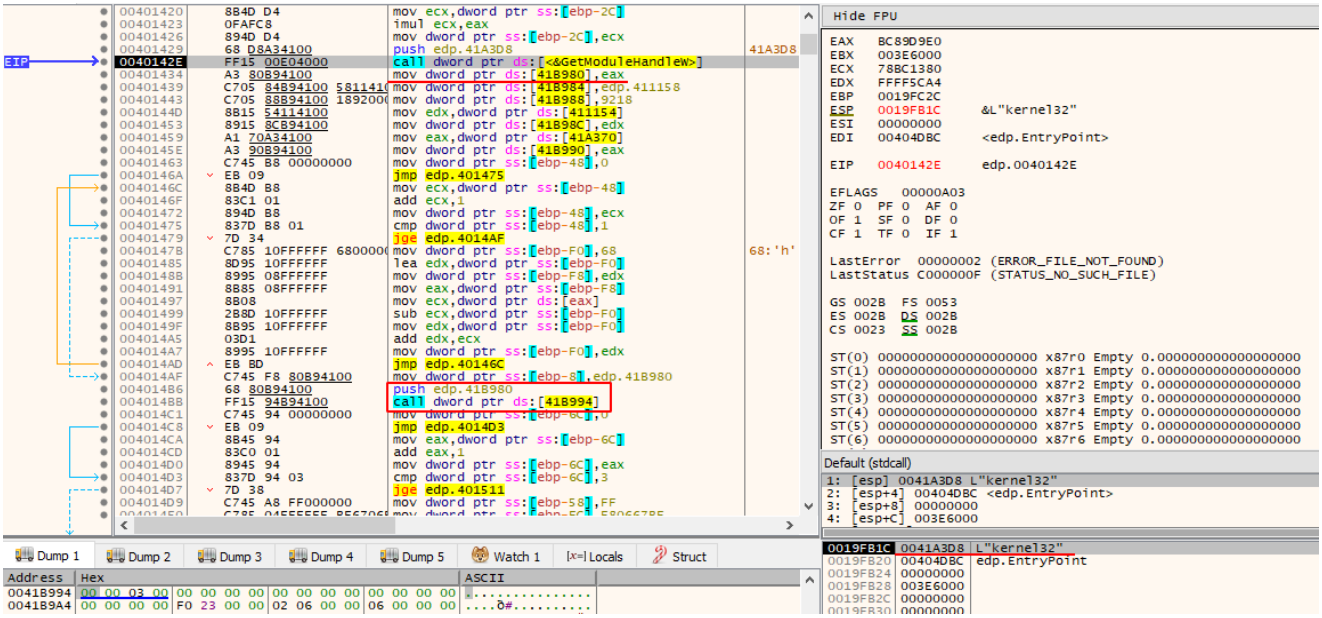
The interesting parts, from a malware analysis point of view, will only take place once the ransomware calls VirtualAllocEx() with PAGE_EXECUTE_READWRITE memory permissions (flProtect). The allocation of pages with such memory permissions is highly indicative that something interesting will be written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



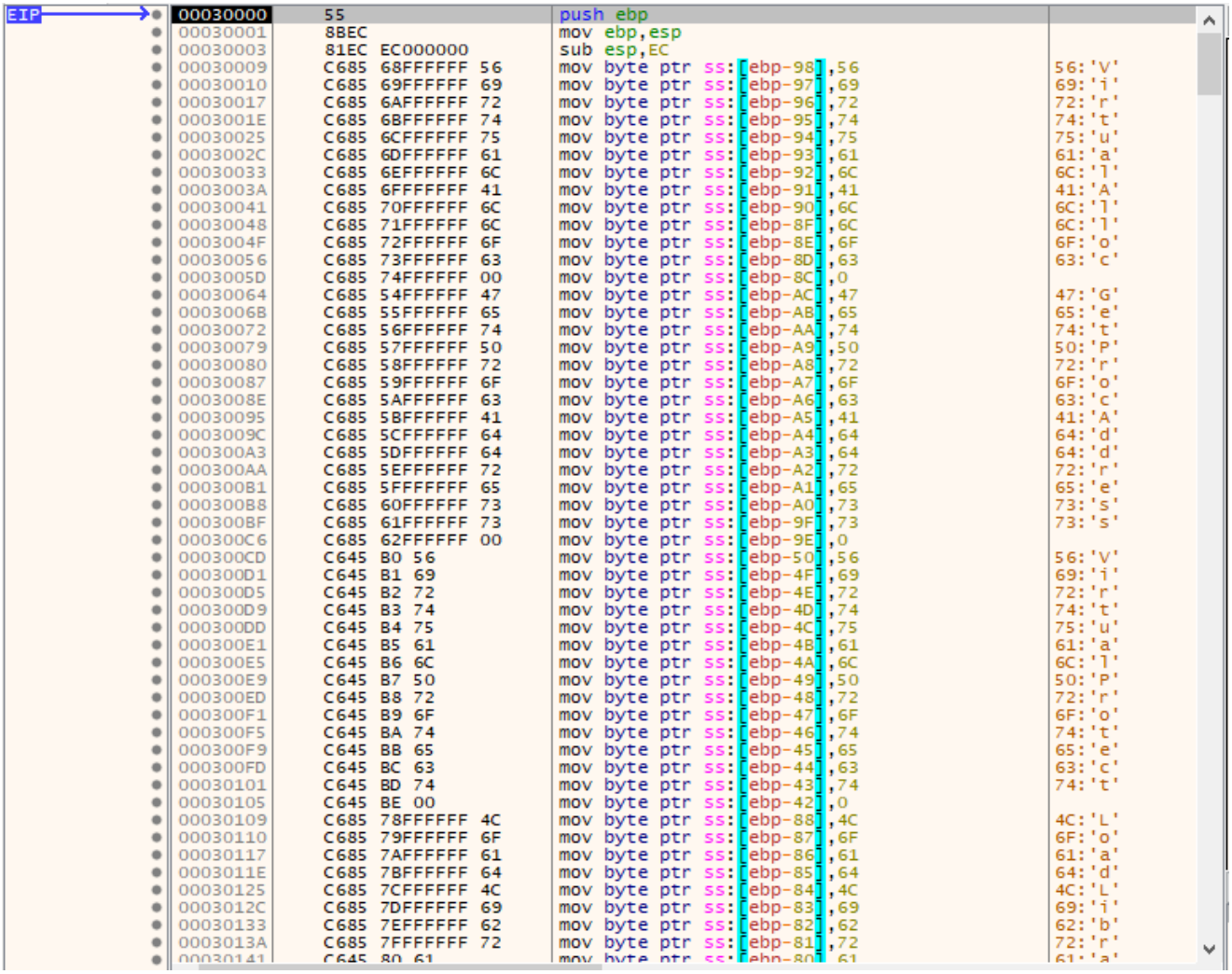
The following image demonstrates the algorithm being used by the ransomware, where it starts decompressing/decrypting and writing shellcode into the new memory area.



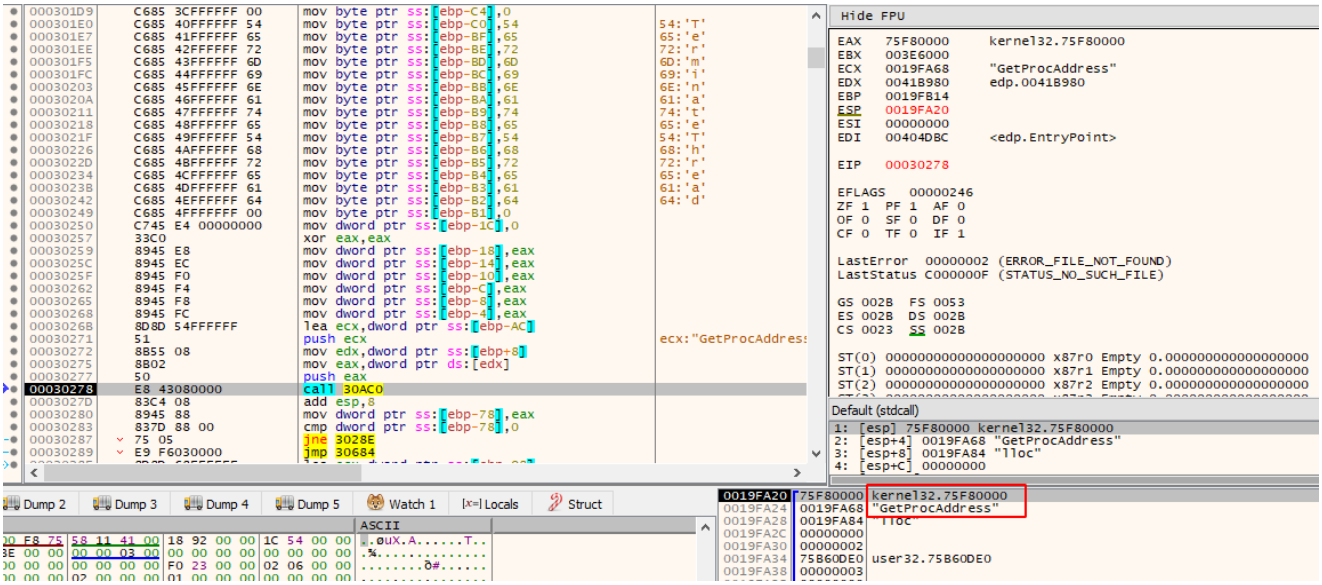
When it is finished writing the shellcode, it will call GetModuleHandleW(L"kernel32") in order to obtain the base address of kernel32.dll that is mapped in the current process address space. It will then transfer control-flow into the new RWX memory area containing the newly decrypted shellcode, passing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



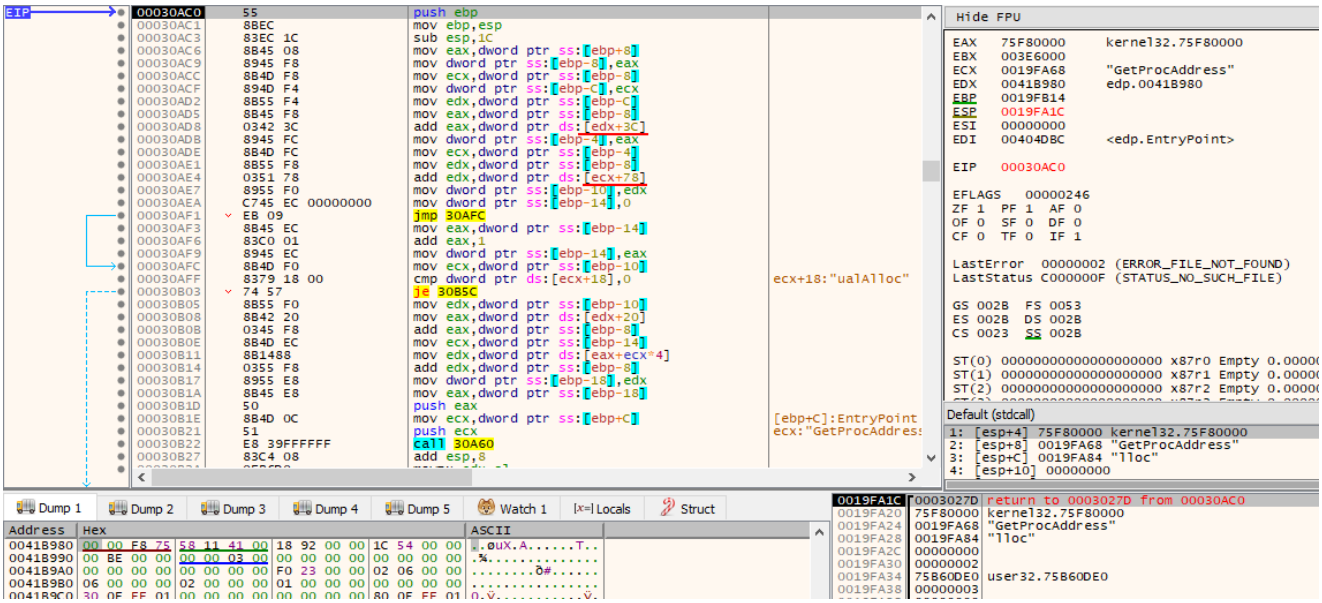
At this point, execution of the shellcode takes place. The following image demonstrates its initial instructions.



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.

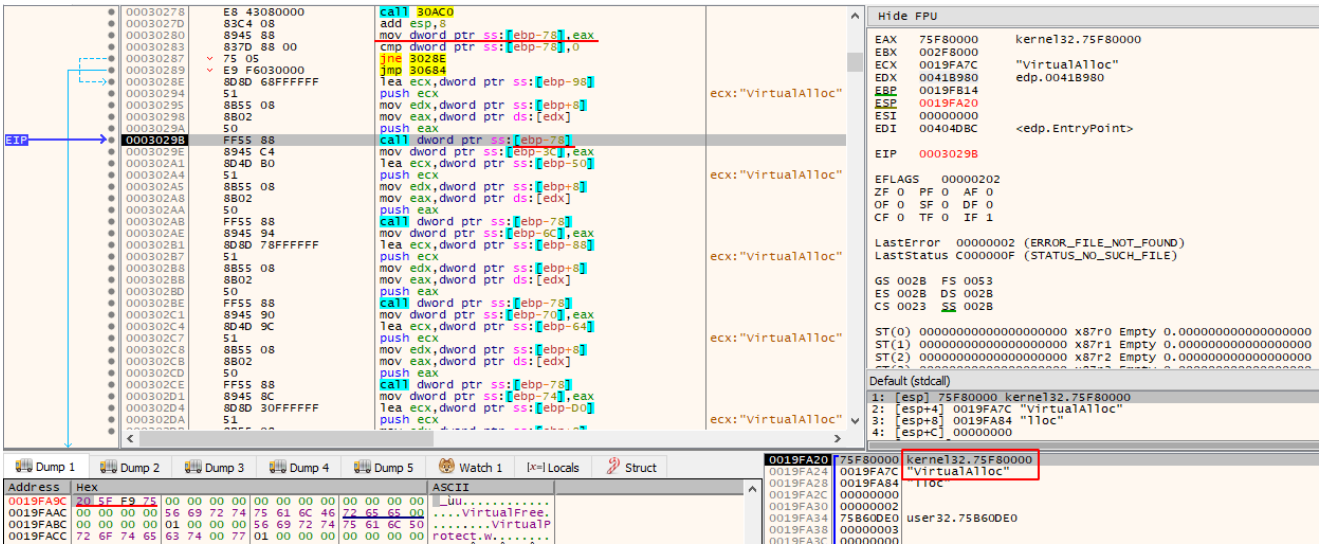


The following image demonstrates the initial instructions of this subroutine.

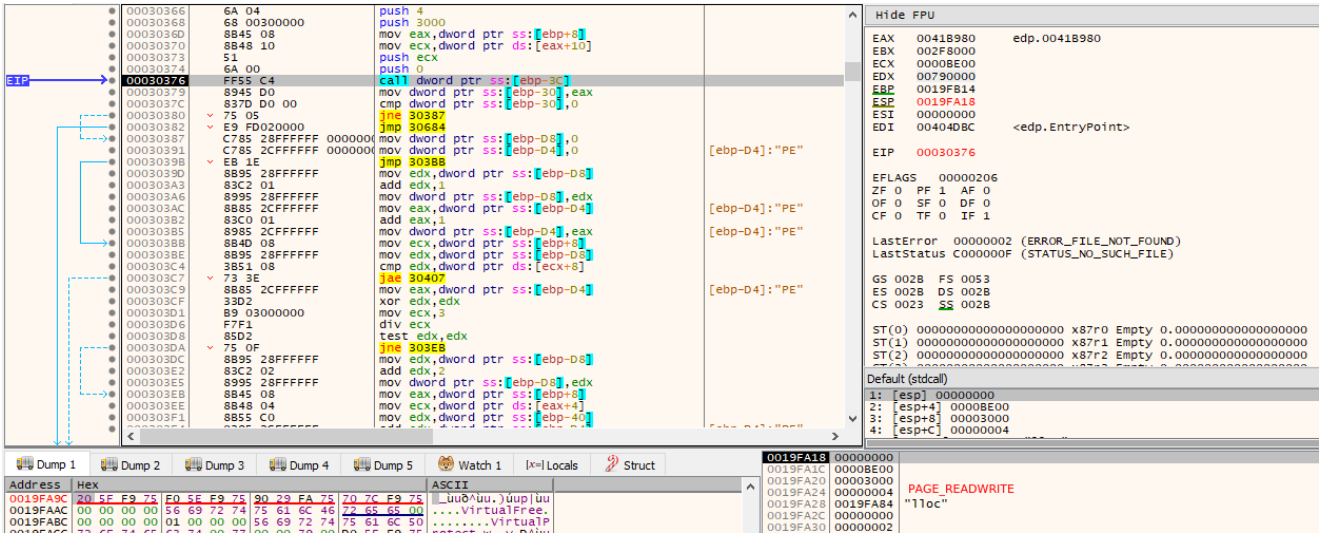


For anyone that has done enough malware reversing, or simply to anyone familiar with the PE file format, it becomes clear that this function is being used to manually iterate through kernel32.dll's exports in order to dynamically resolve, at runtime, the address of GetProcAddress() so that it can be 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-

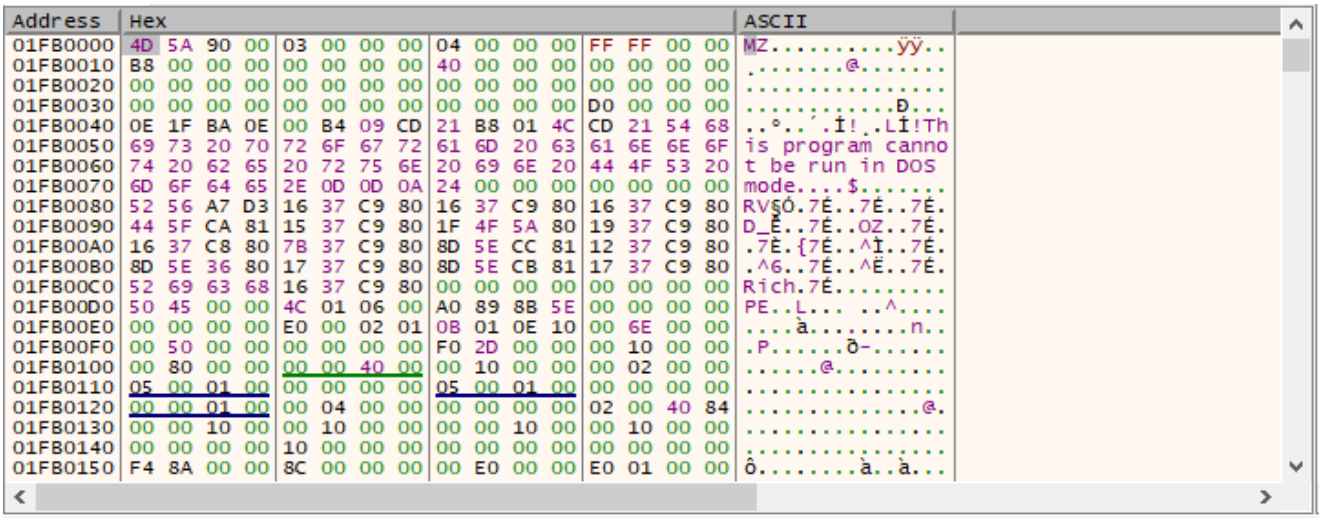
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress (at offset 0x78). After GetProcAddress() is resolved, it will then be used to dynamically resolve the rest of the APIs whose names were constructed previously one byte at a time on the stack.



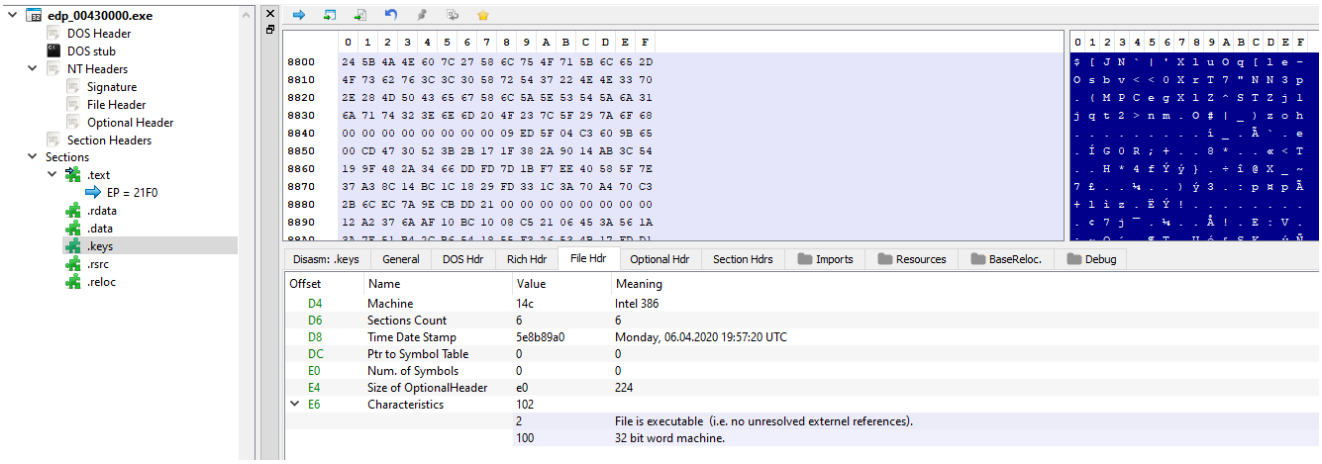
Then, VirtualAlloc() is called, but this time the memory access permissions do not include execution, allowing only reads and writes (PAGE_READWRITE).



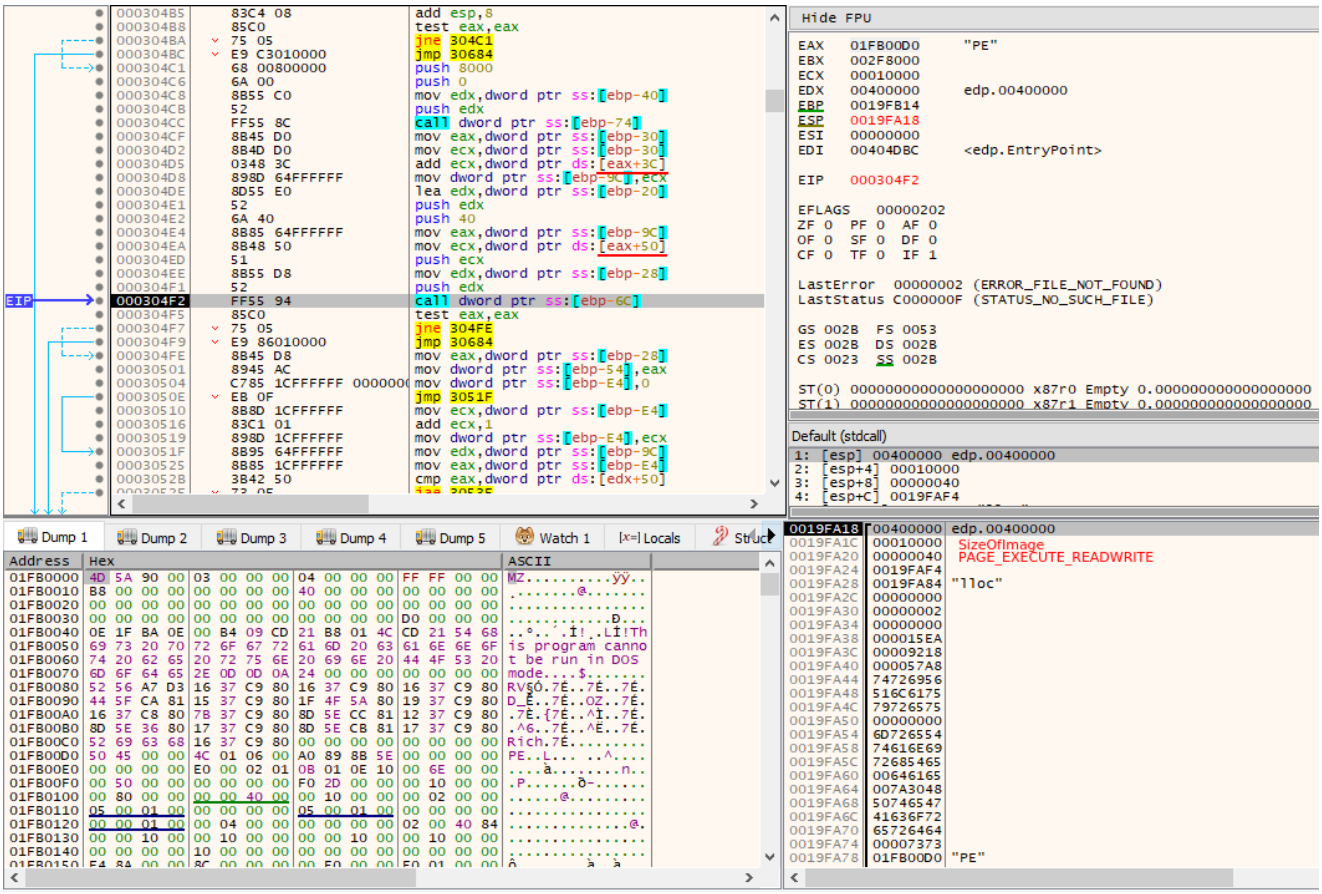
Eventually, a relatively long series of operations are performed, resulting in writes to the newly allocated memory region, where a full PE is decrypted at runtime into it. It is easily recognizable through the MS-DOS MZ header. This particular point is the best time to dump the memory region into disk, 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 program execution.



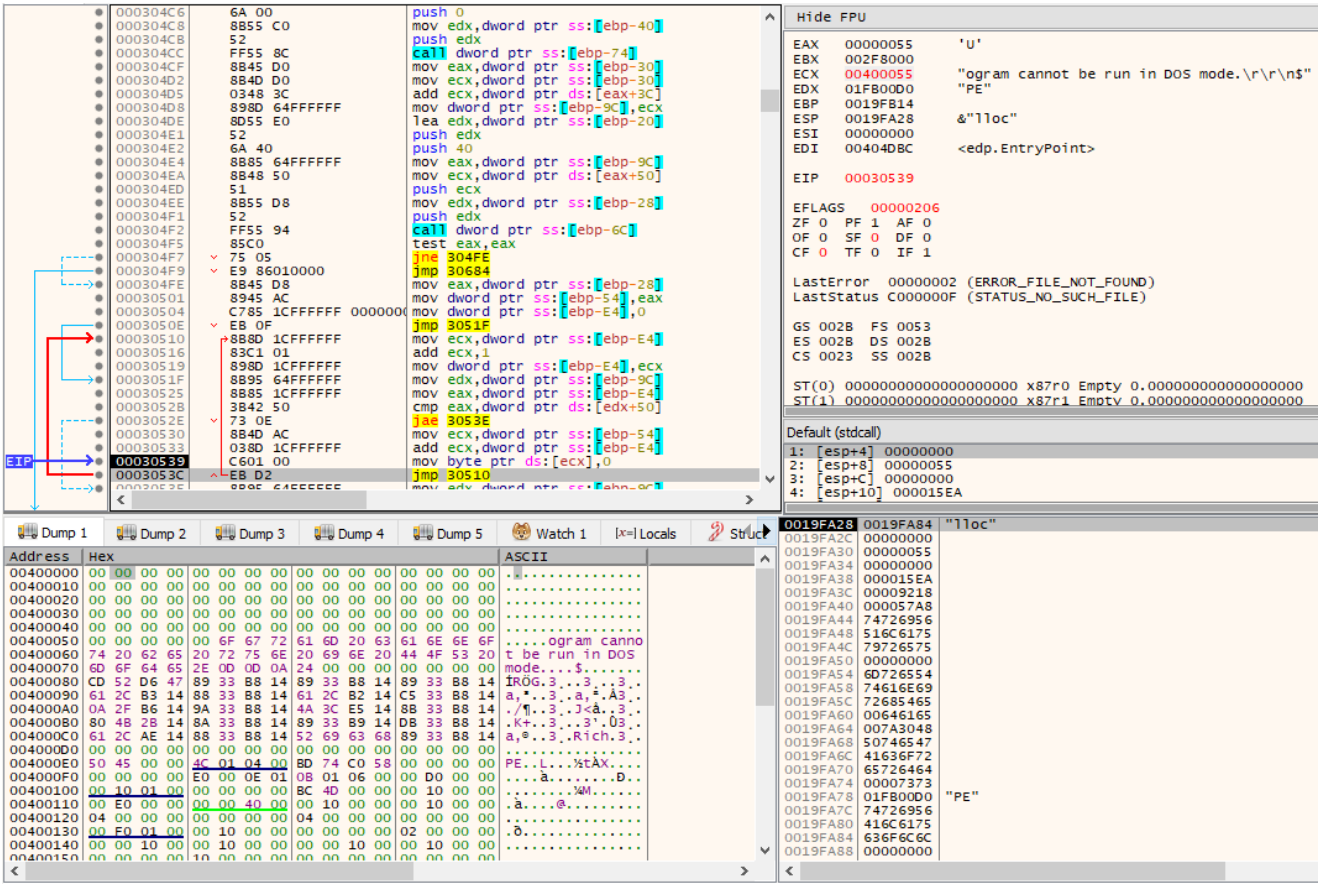
By dumping the PE to disk, it can be further analyzed through the very same basic static analysis steps in order to get a general idea of what this new binary might have (or do). As we can see, it contains an interesting .keys section and its Time Date Stamp (compilation date) is set to Monday, 06.04.2020 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.



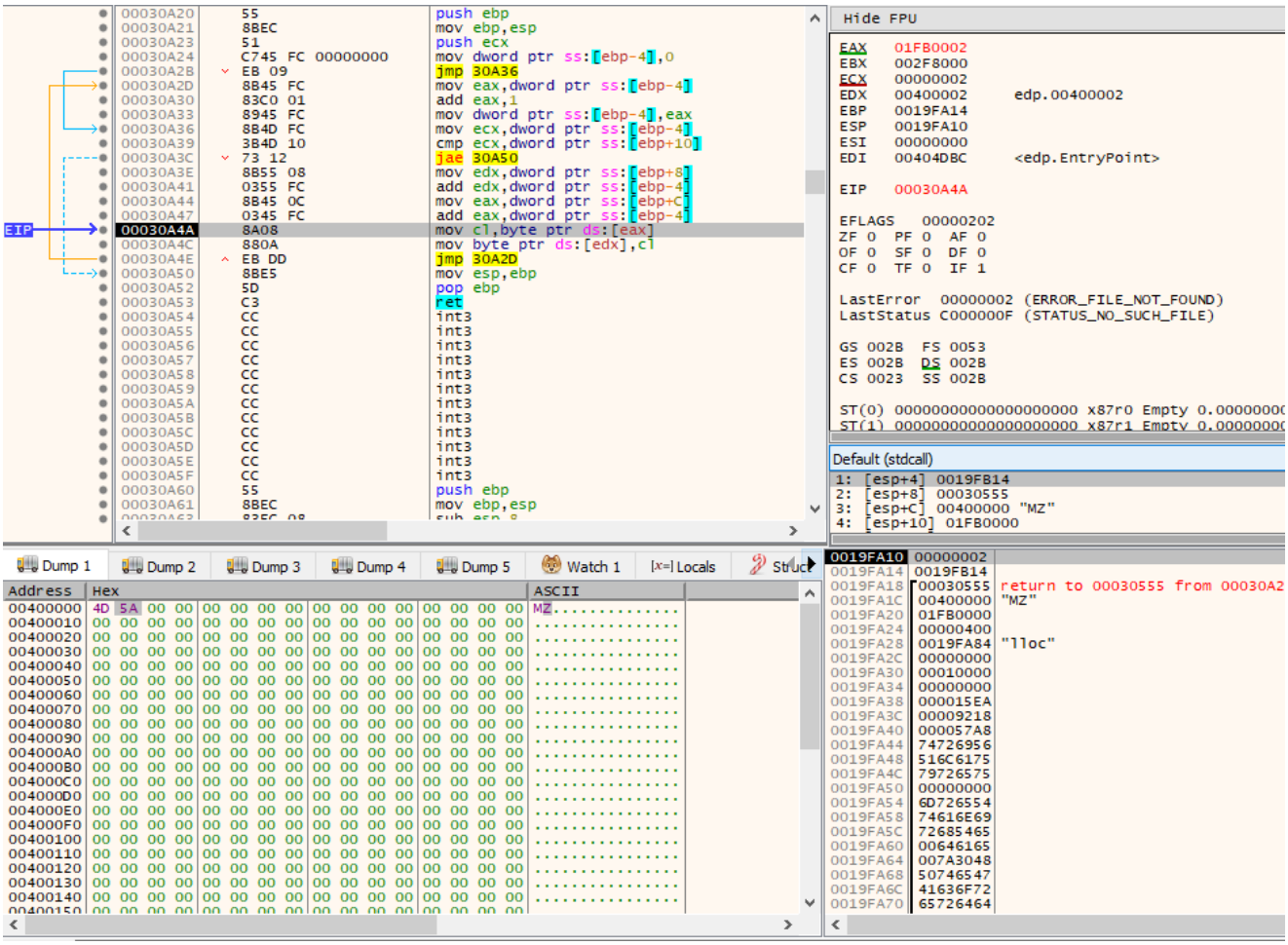
Proceeding execution, the SizeOfImage (the size of the image, in bytes, including all headers) of the newly decrypted PE is then obtained, via pNtHdr->OptionalHeader.SizeOfImage, as seen by the use of offset 0x3C to get the address of the PE header, and then by adding to that result 0x50. The SizeOfImage 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.



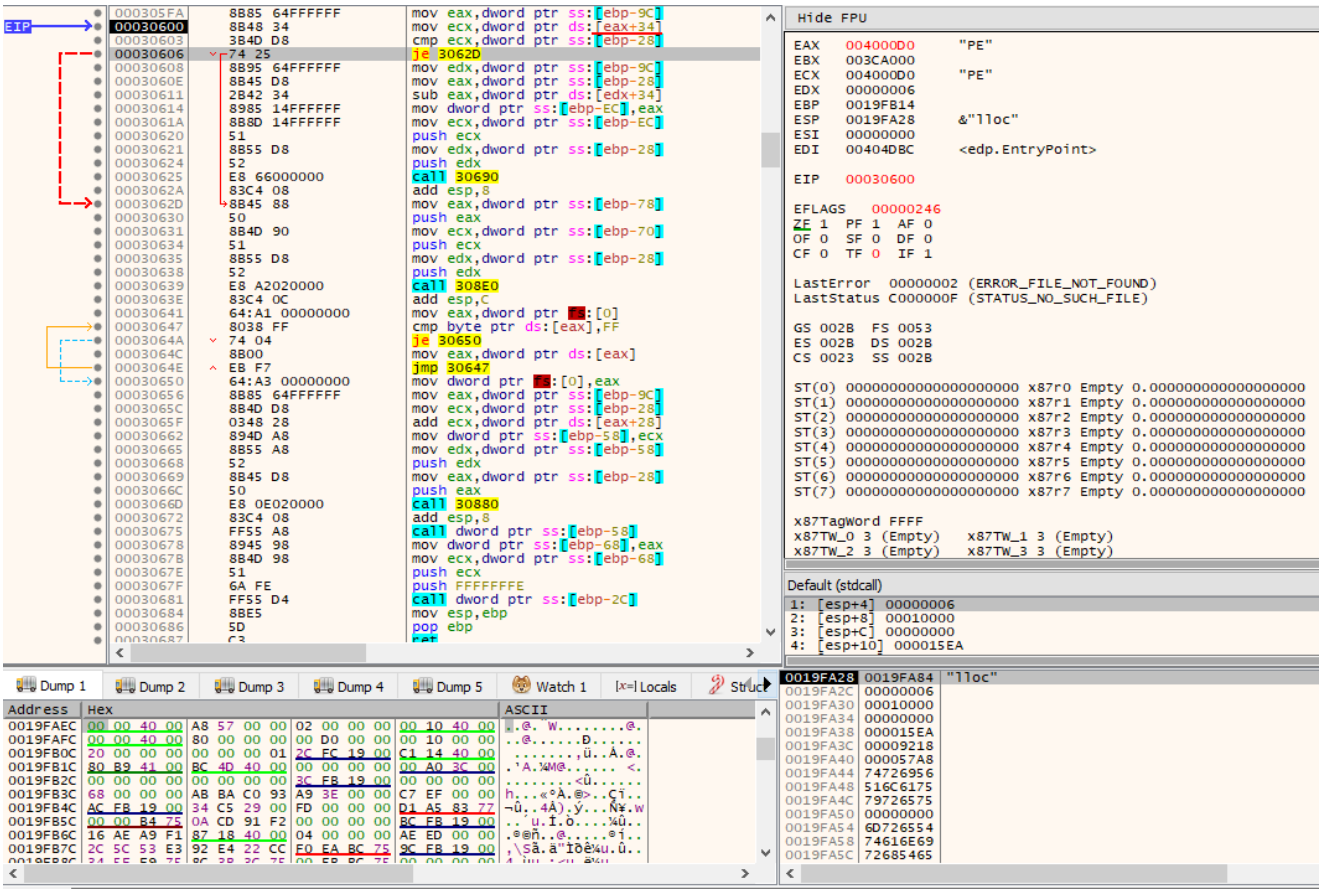
After the VirtualAlloc() call, the main binary's image (of the currently running process) is overwritten with 0's, in a loop that ends after executing SizeOfImage (of the new decrypted PE) times.



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.

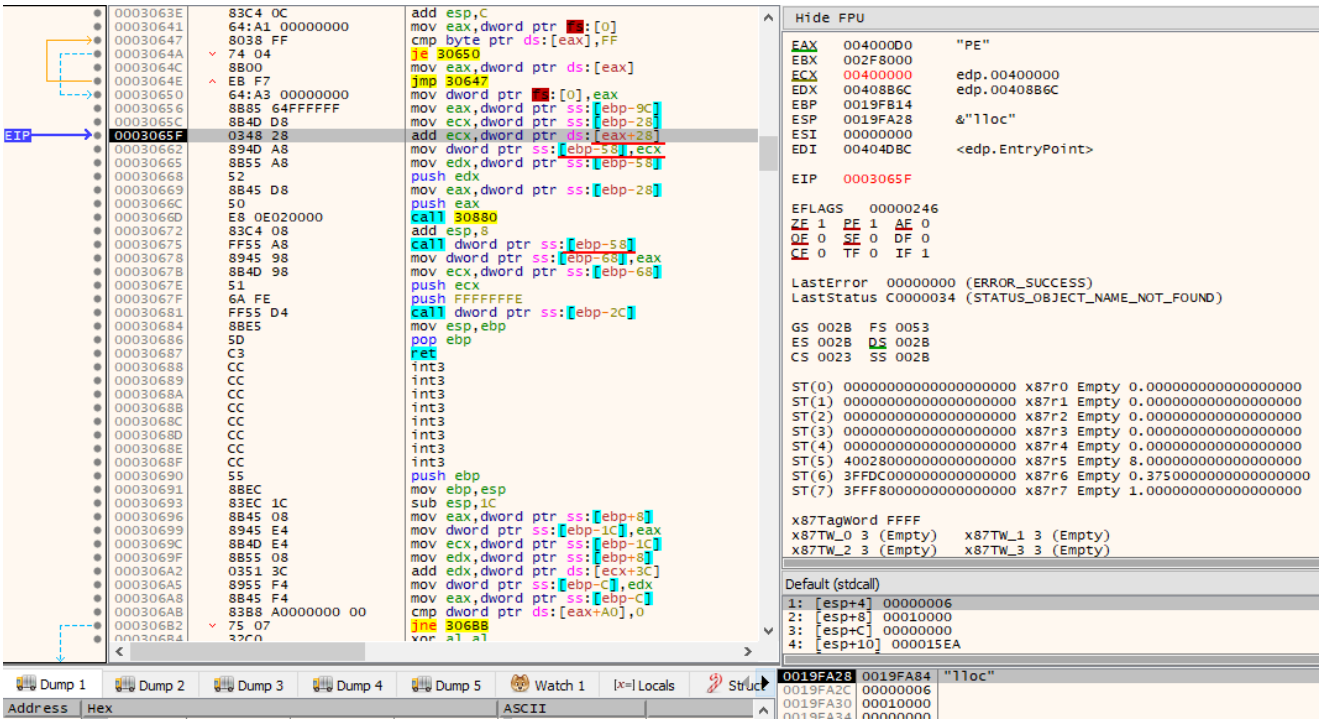


By comparing the base address where the new PE was placed against its ImageBase (preferred base address), via pNtHdr->OptionalHeader.ImageBase (as seen by offset 0x34), it can decide whether base relocations need to take place or not. In this case, base relocations do not need to be performed, but there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



The 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).

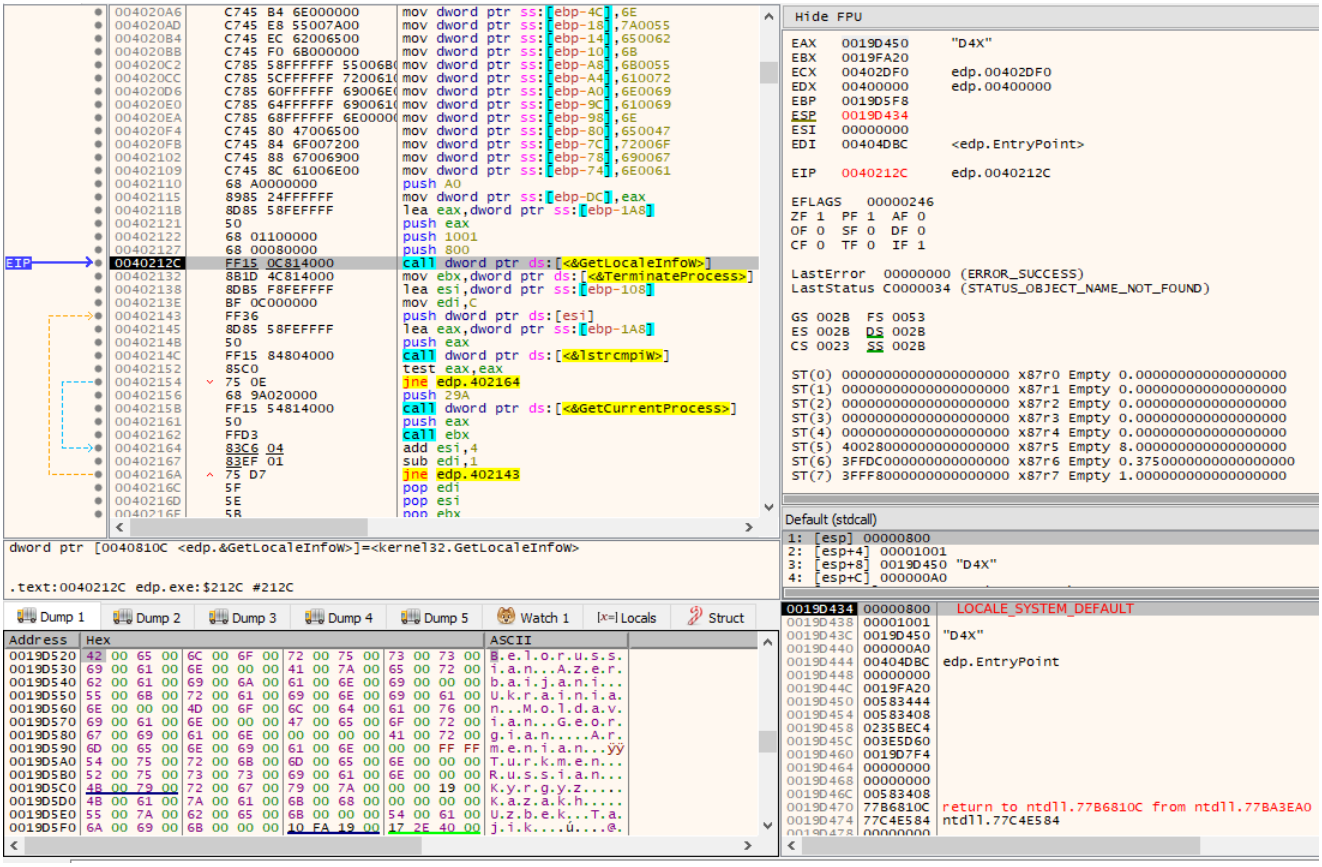
By accessing AddressOfEntryPoint, via pNtHdr->OptionalHeader.AddressOfEntryPoint (offset 0x28), the Original Entry Point (OEP) is then obtained and subsequently called, thus transfering execution to the newly unpacked executable, as it is now ready to be executed.



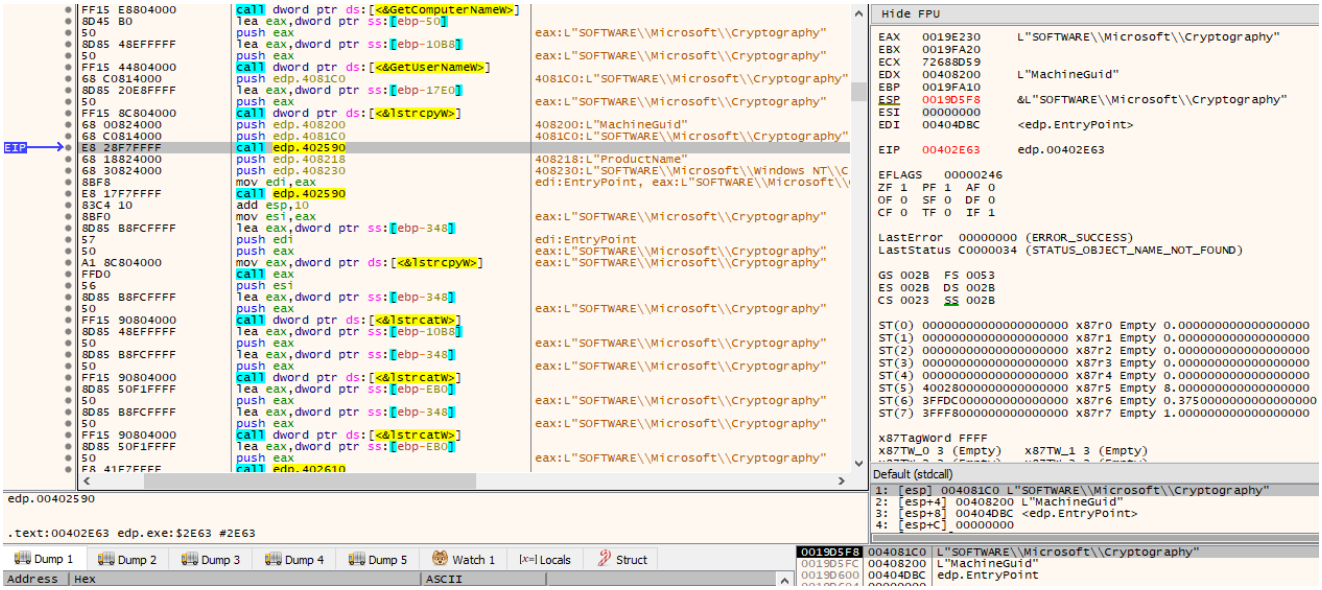
One of the very first things that is done after execution starts at the new PE's entry point is to call a subroutine that eventually calls GetLocaleInfoW() with LCID LOCALE_SYSTEM_DEFAULT (default locale for the operating system), in order to compare it against a possible set of unicode strings previously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

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

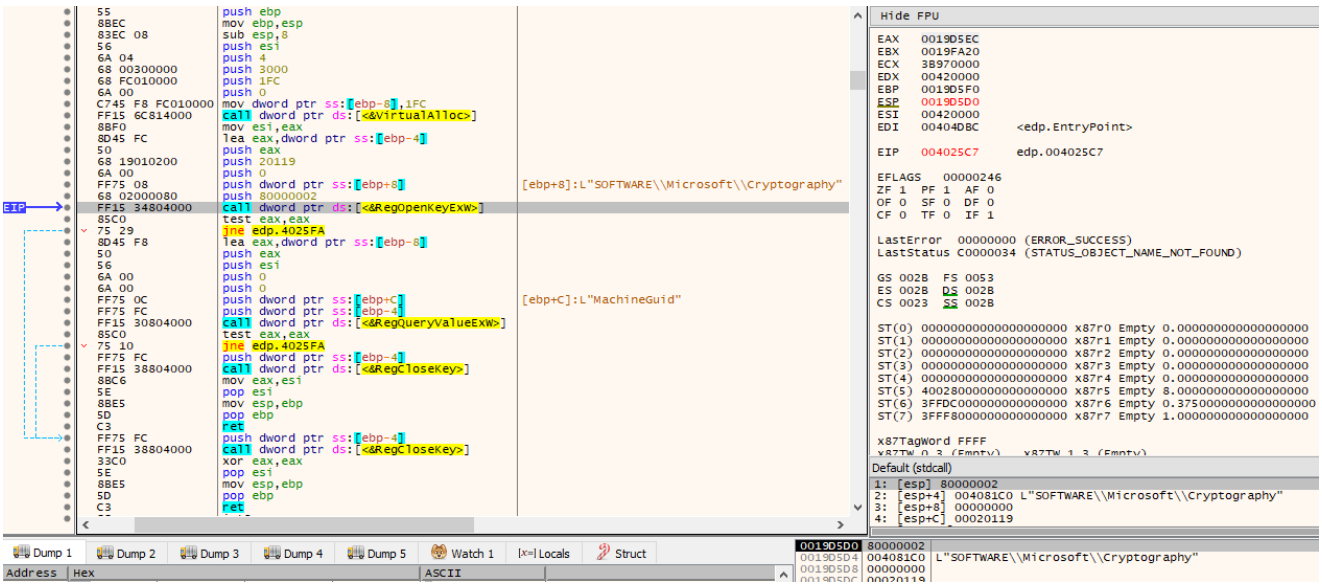
If the requested locale information matches any of those strings, as seen by the use of lstrcmpiW(), then the current process is terminated via TerminateProcess() with exit code 666.



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".

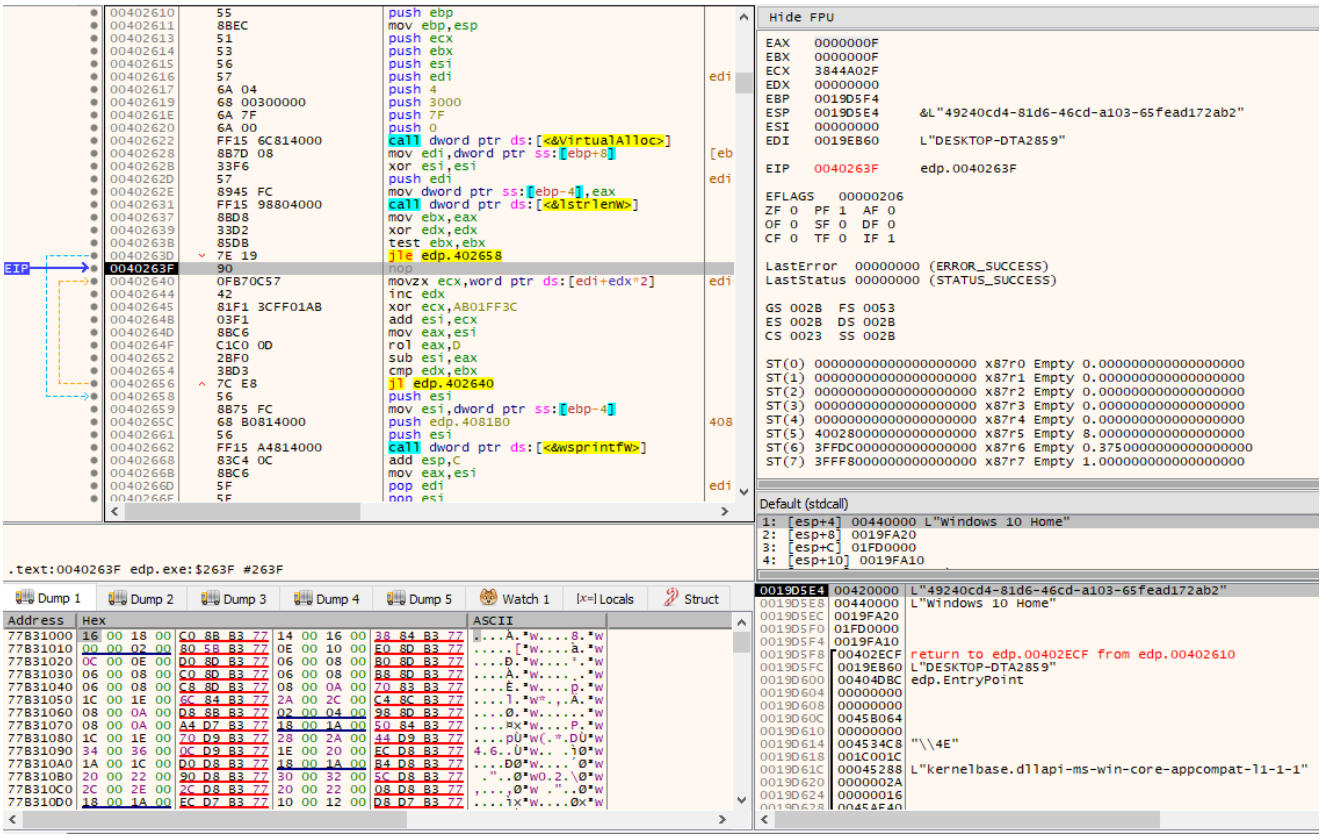


The function simply allocates a page via VirtualAlloc(), opens the provided subkey via RegOpenKeyExW() from the HKEY_LOCAL_MACHINE (HKLM) registry hive and KEY_READ access rights, and then retrieves the data for the provided value name associated with the opened registry key via RegQueryValueExW(). The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

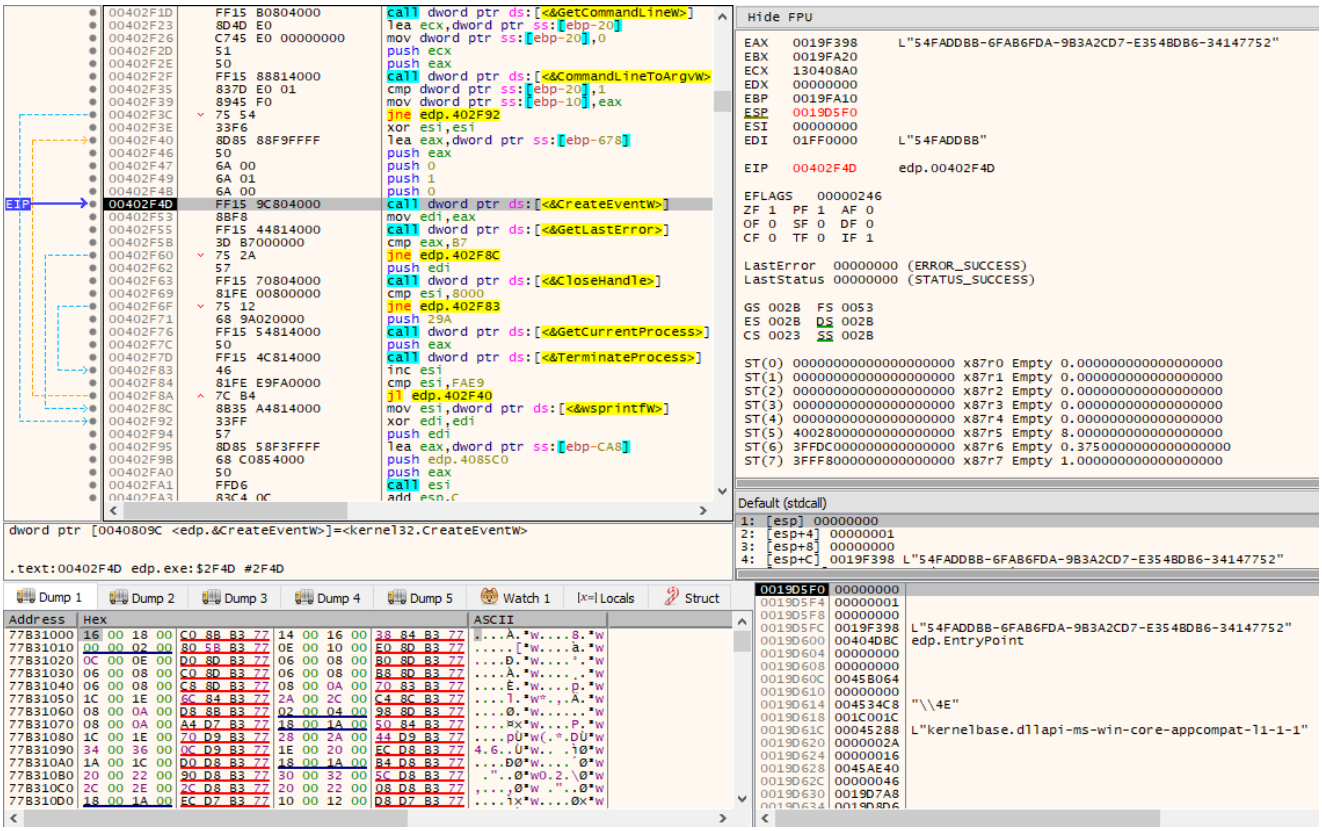


For each of the data obtained via the calls to the APIs (GetComputerNameW() and GetUserNameW()) and the function responsible for retrieving the data associated with the opened registry keys, it will perform a series of operations on them. Specifically, for each

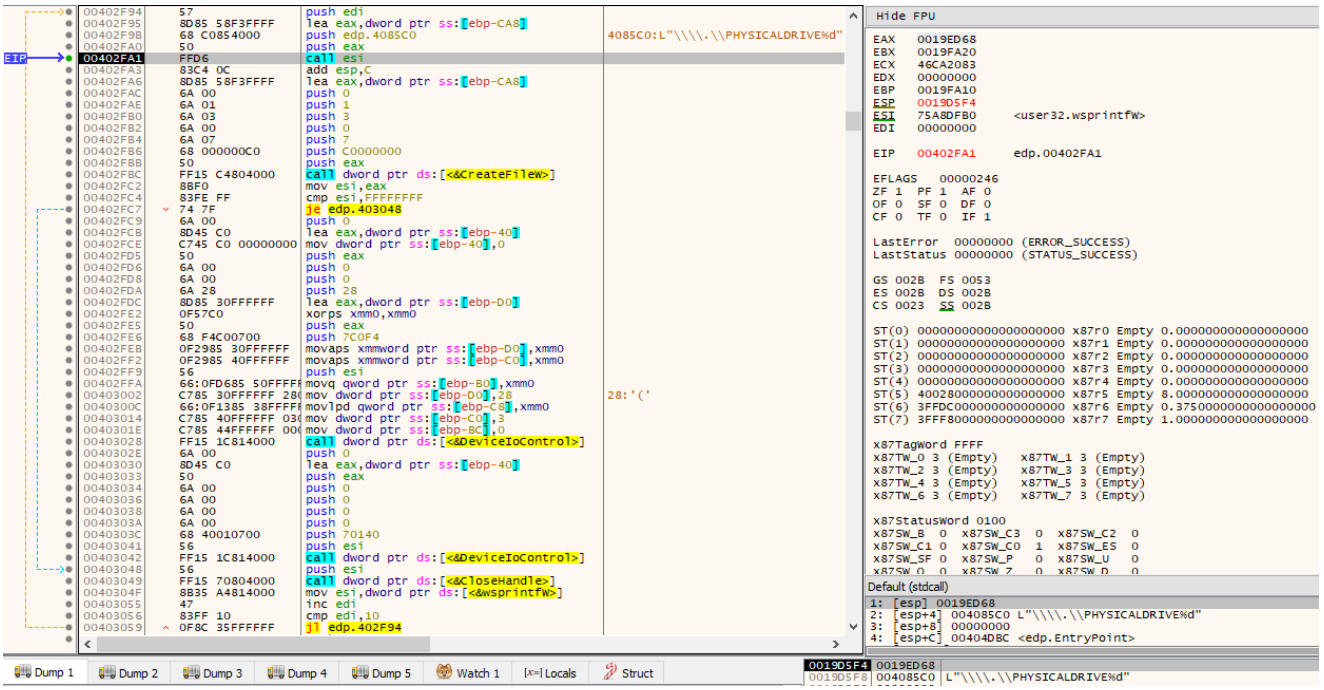
character, it will XOR it the value 0xAB01FF3C, add the previous value to the next one, rotate it left 13 bits and subtract the result of the rotate operation with the value before the rotate. This is done so that unique IDs result from the operations, where they are later concatenated.



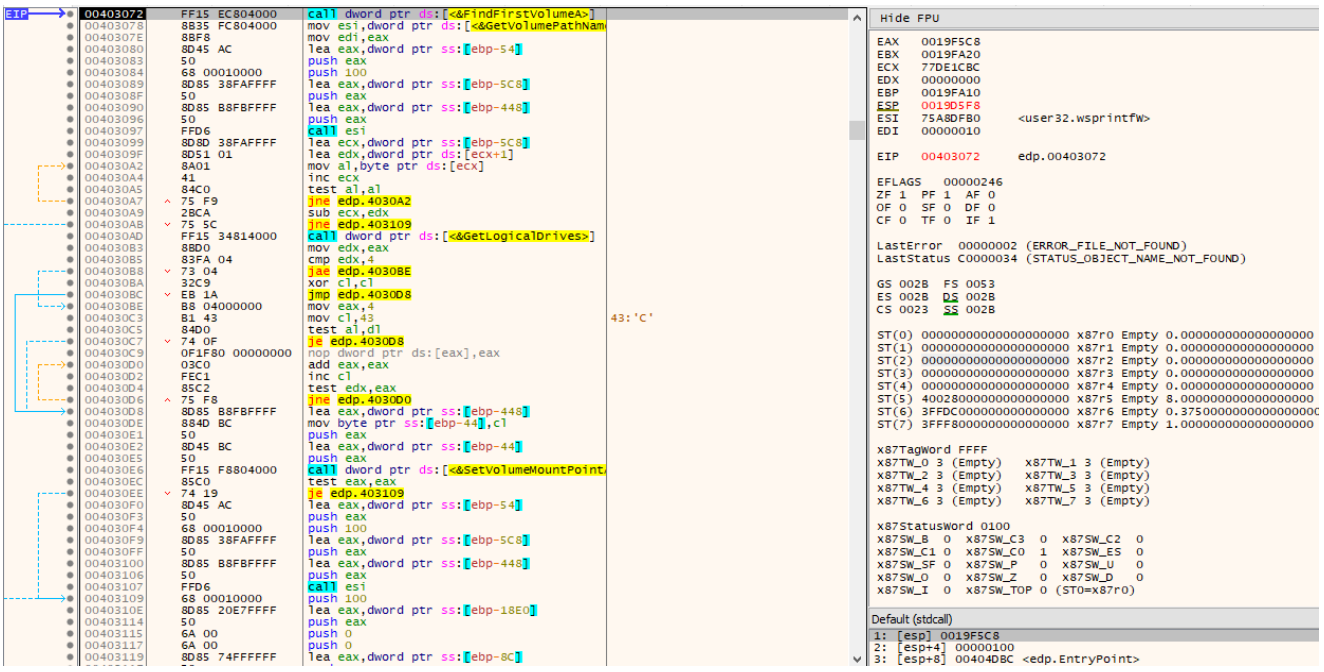
The result of the previous operations (unique IDs) and their concatenation is so that it will be used as lpName (the name of the event object) passed to CreateEventW(). But first, it checks if argc (argument count) is 1, if it's not, CreateEventW() is skipped entirely. However, if it is 1, a loop is 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.



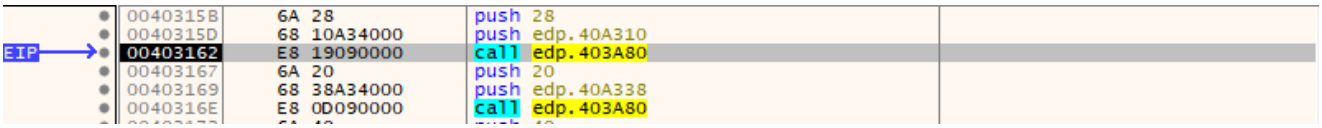
Then, it will enter another loop, executed 17 times, where it tries to open \\.\PHYSICALDRIVE%d (a physical hard drive) via CreateFileW(), where %d is incremented for each iteration in the loop, starting from 0. If the return value from CreateFileW() differs from 0xFFFFFFFF, it will then call DeviceIoControl() 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.



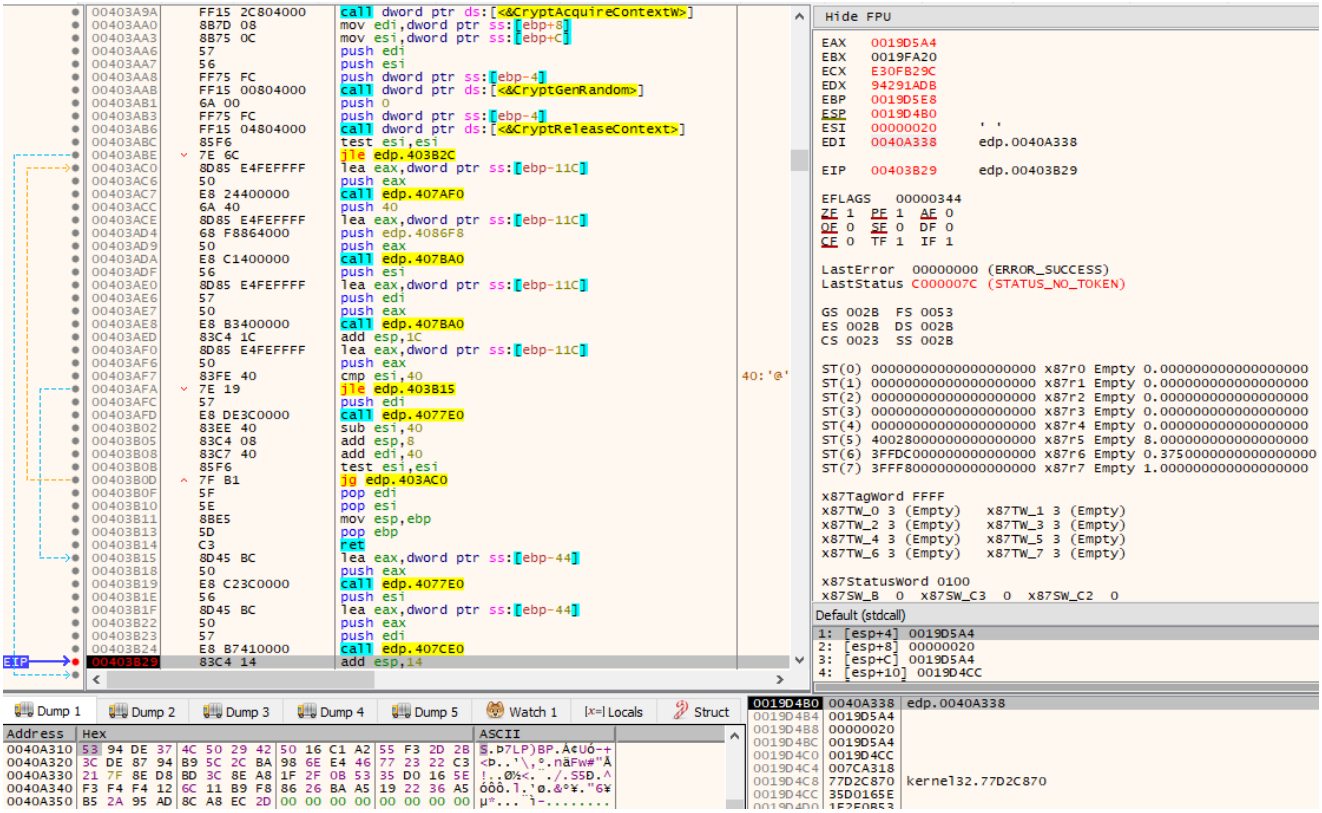
For every existing volume without an associated drive letter, it will then attempt to associate it an unused drive letter. This is done by scanning and iterating through existing volumes on the system by making use of the FindFirstVolumeA()/FindNextVolumeA() combination and determining whether a drive 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.



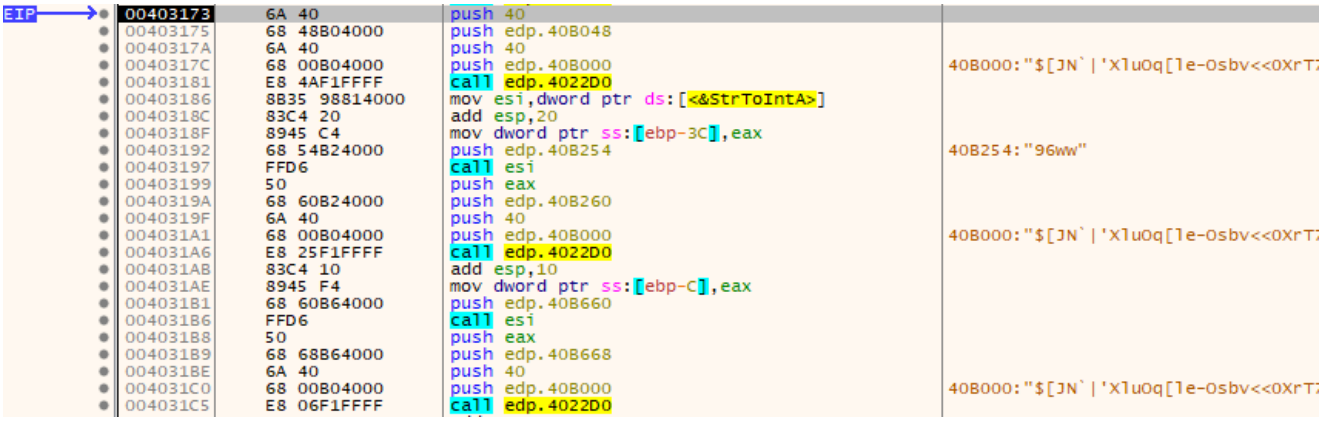
At this point, the same routine will be called twice, one after the other, with different arguments.



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.

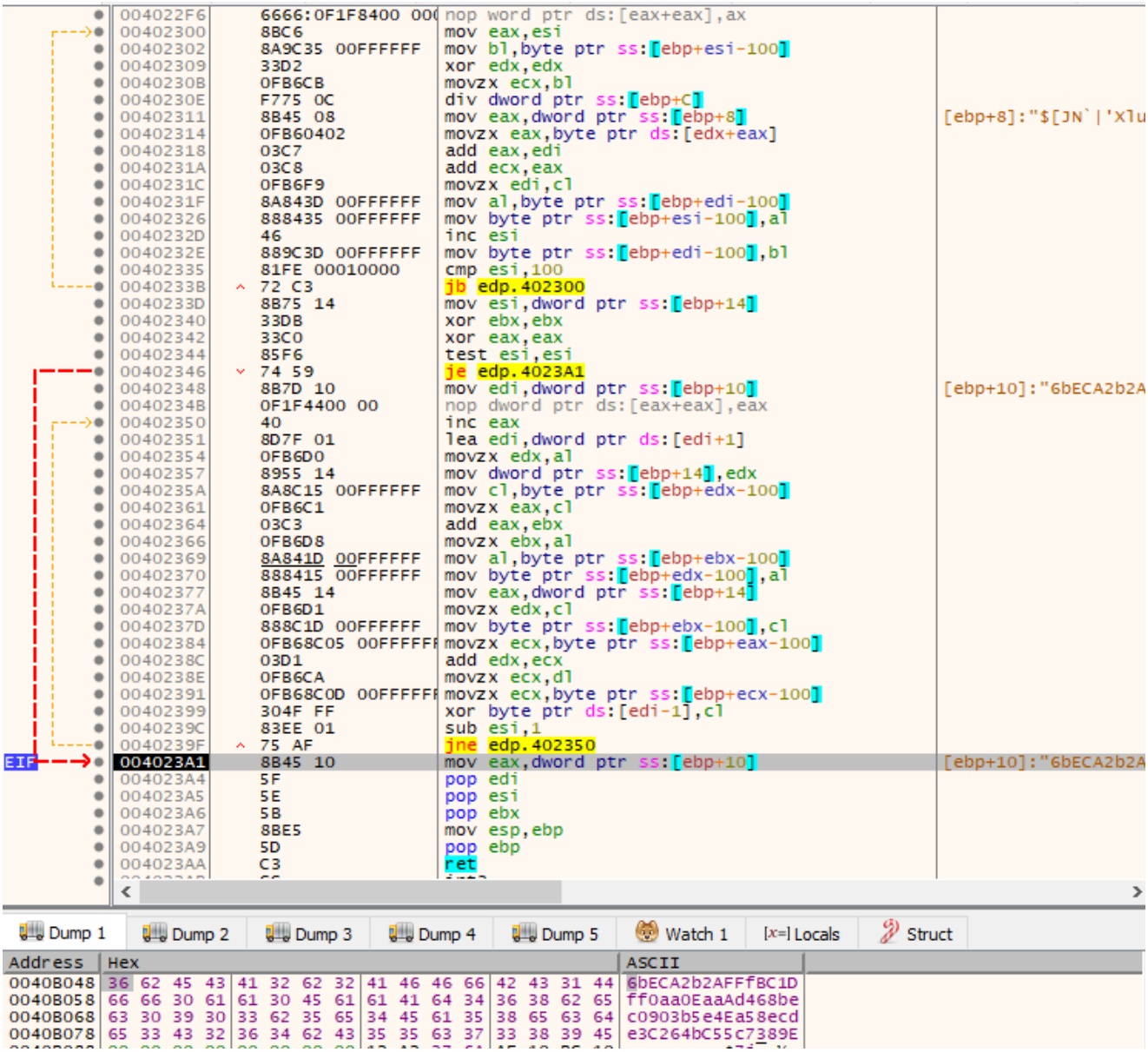


Another routine will be called, this time called thrice. One of the arguments to the function that is always passed is a pointer to a suspiciously looking string.



)

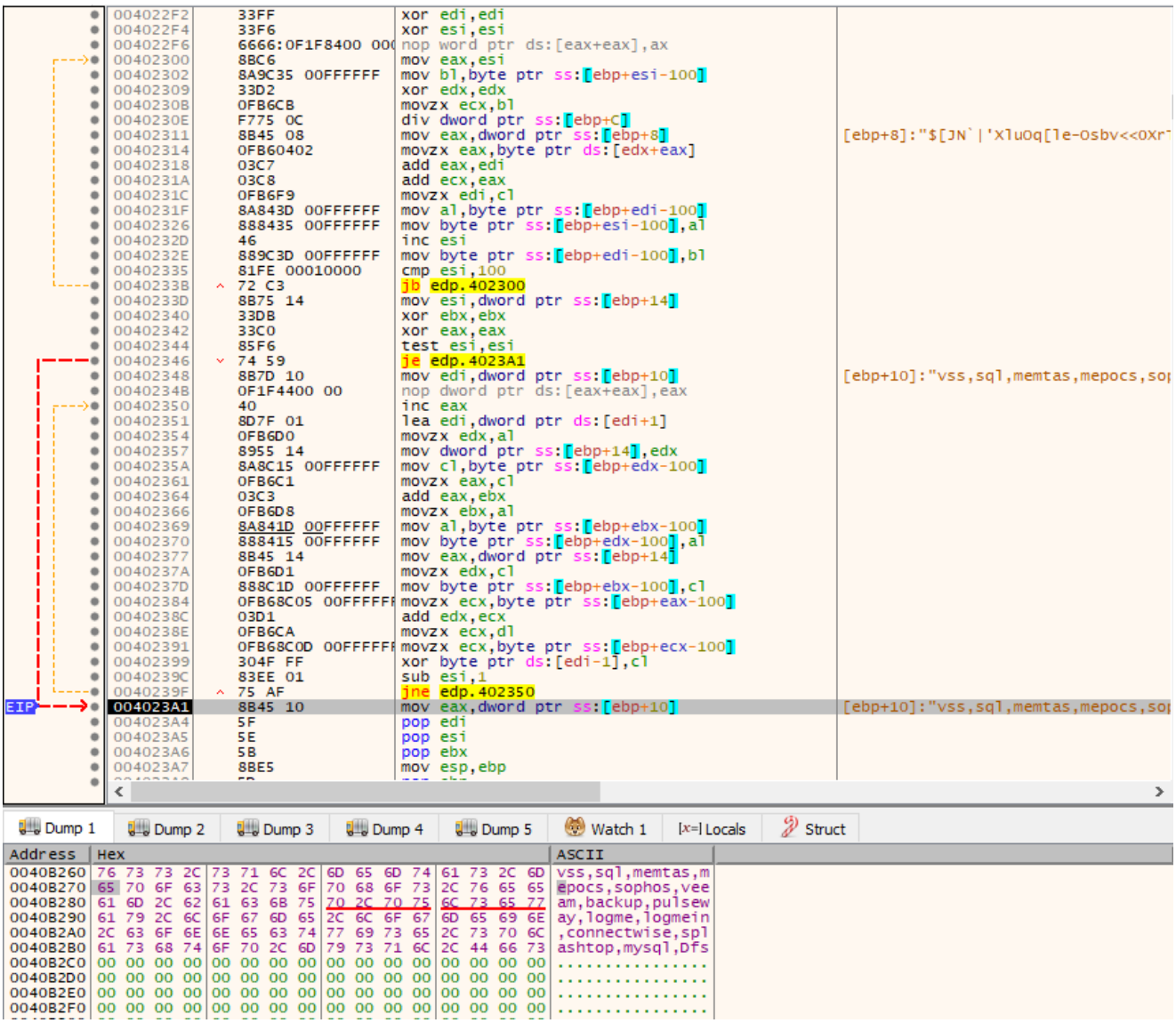
It turns out that this function is responsible for decrypting several data stored in the binary's .keys section. The first time the routine is called, it decrypts the Tor client chat ID used to communicate with the perpetrators.



The 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

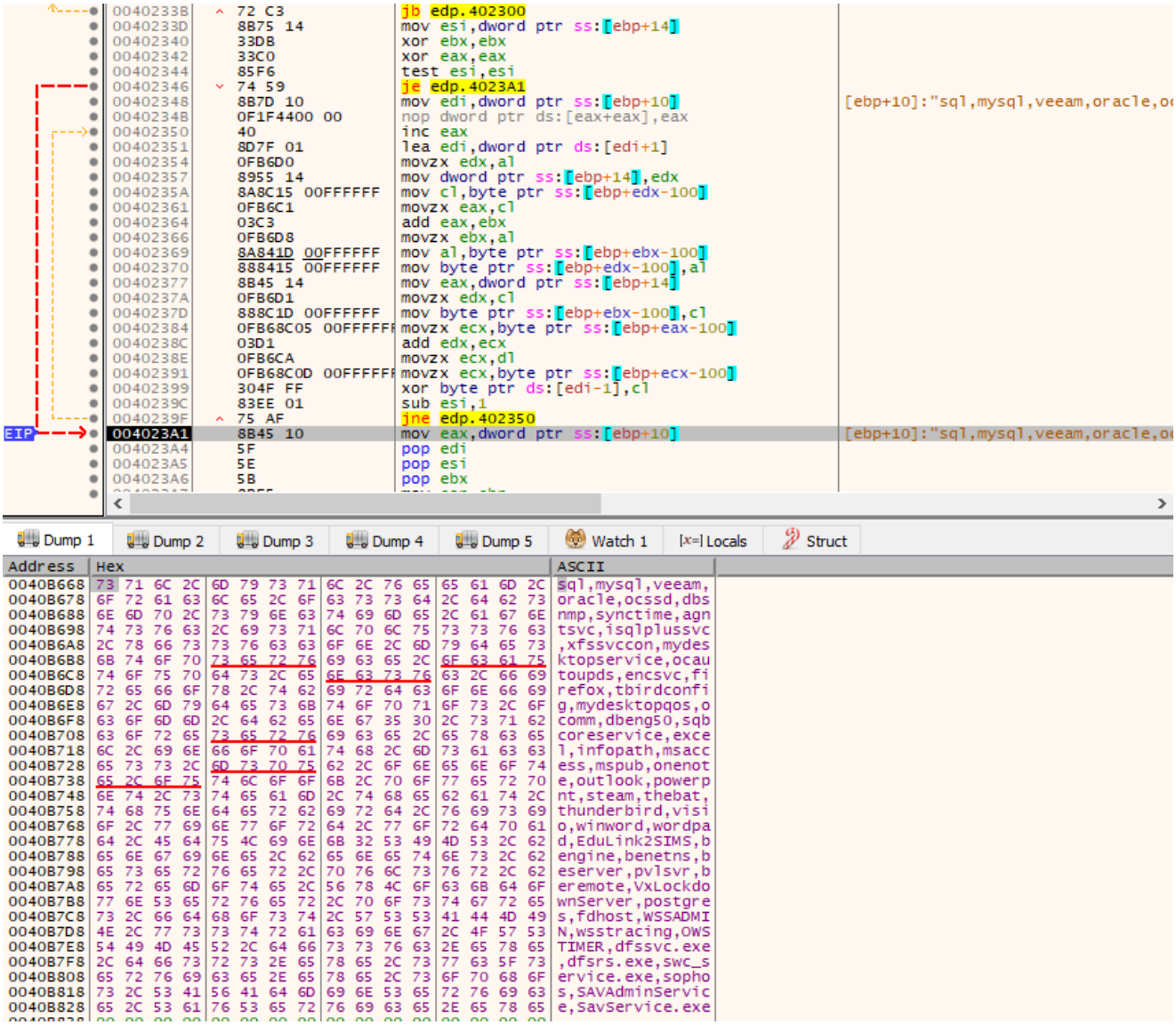


The third time it is called, it is used to decrypt another series of strings, which will later be used as reference for substrings to look for in order to determine which processes to terminate, as we will also see. The strings are:

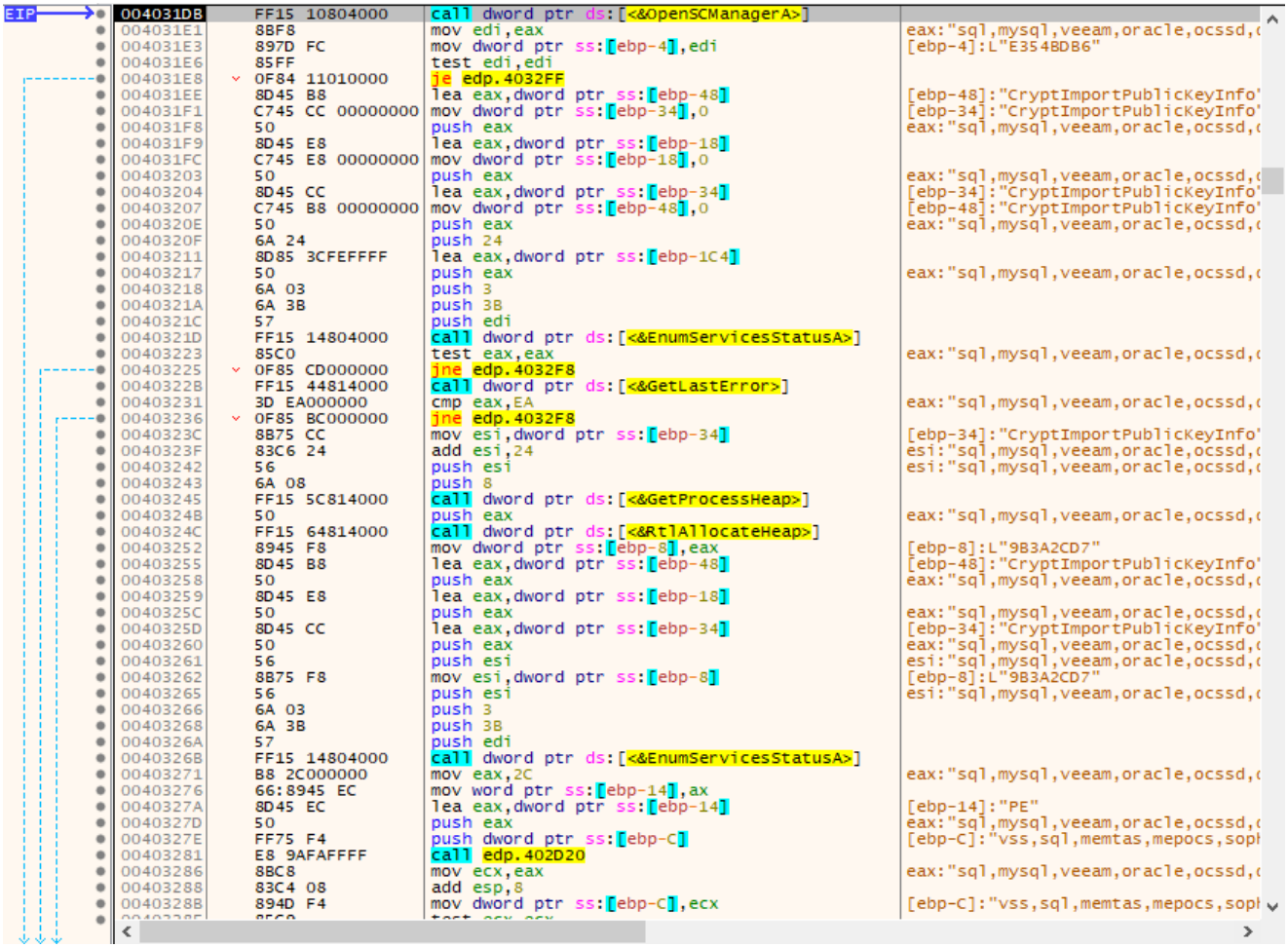
- sql
- mysql
- veeam
- oracle

- ocssd
- dbsnmp
- synctime
- agntsvc
- isqlplussvc
- xfssvccon
- 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

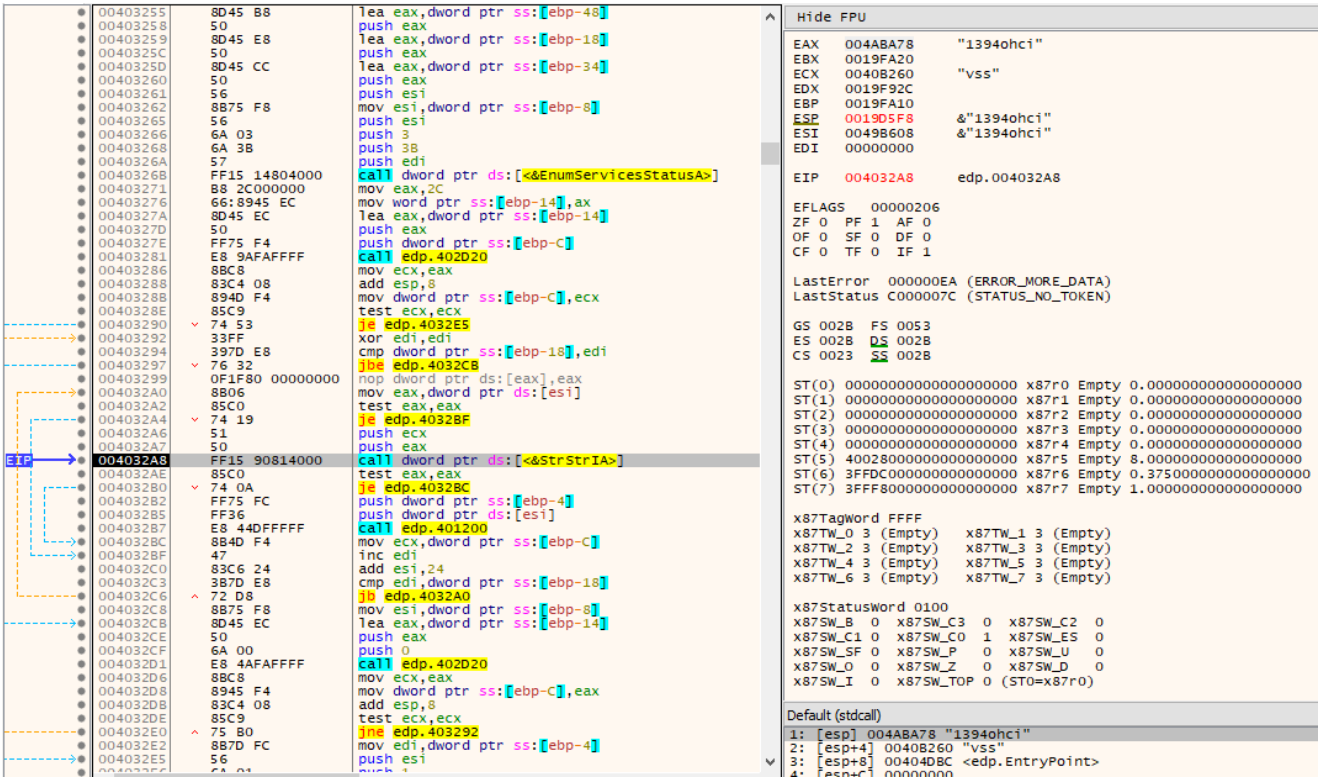


After some data has been decrypted, as seen in the previous steps, it will now attempt to establish a connection to the service control manager (via OpenSCManagerA()) of the local computer (lpMachineName is set to NULL) and open the SERVICES_ACTIVE_DATABASE database (lpDatabaseName set to NULL) with 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  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]:L"E354BDB6"
     004031E6   85FF            test edi,edi
     004031E8   0F84 11010000   je 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   jne edp.4032F8
     0040322B   FF15 44814000   call dword ptr ds:[<&GetLastError>]
     00403231   3D EA000000     cmp eax,EA                           eax:"sql,mysql,veeam,oracle,ocssd,
     00403236   0F85 BC000000   jne 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>]
     0040324B   50              push eax                             eax:"sql,mysql,veeam,oracle,ocssd,
     0040324C   FF15 64814000   call dword ptr ds:[<&RtlAllocateHeap>]
     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,sop
     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,sop
```

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.

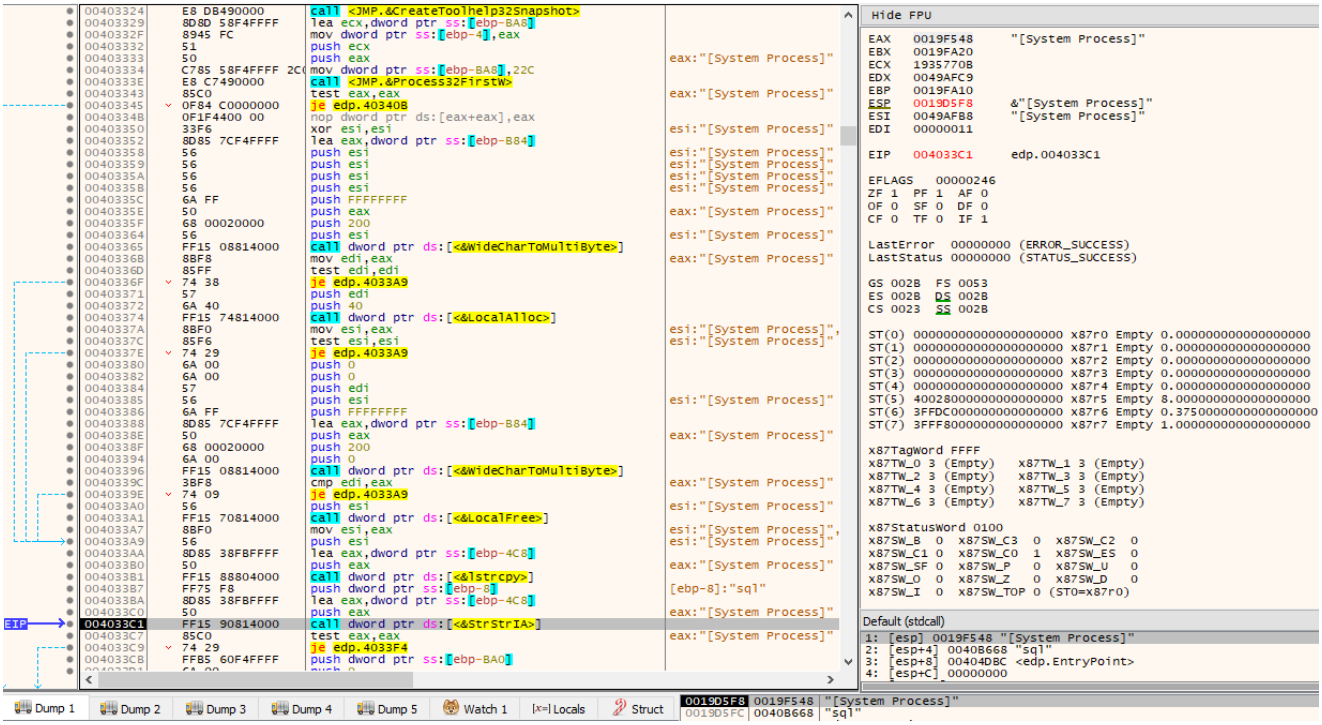


This subroutine is responsible for calling GetTickCount(), opening the existing service via OpenServiceA() with dwDesiredAccess of SERVICE_STOP | SERVICE_QUERY_STATUS | SERVICE_ENUMERATE_DEPENDENTS, look up the status of the service via QueryServiceStatusEx() and:

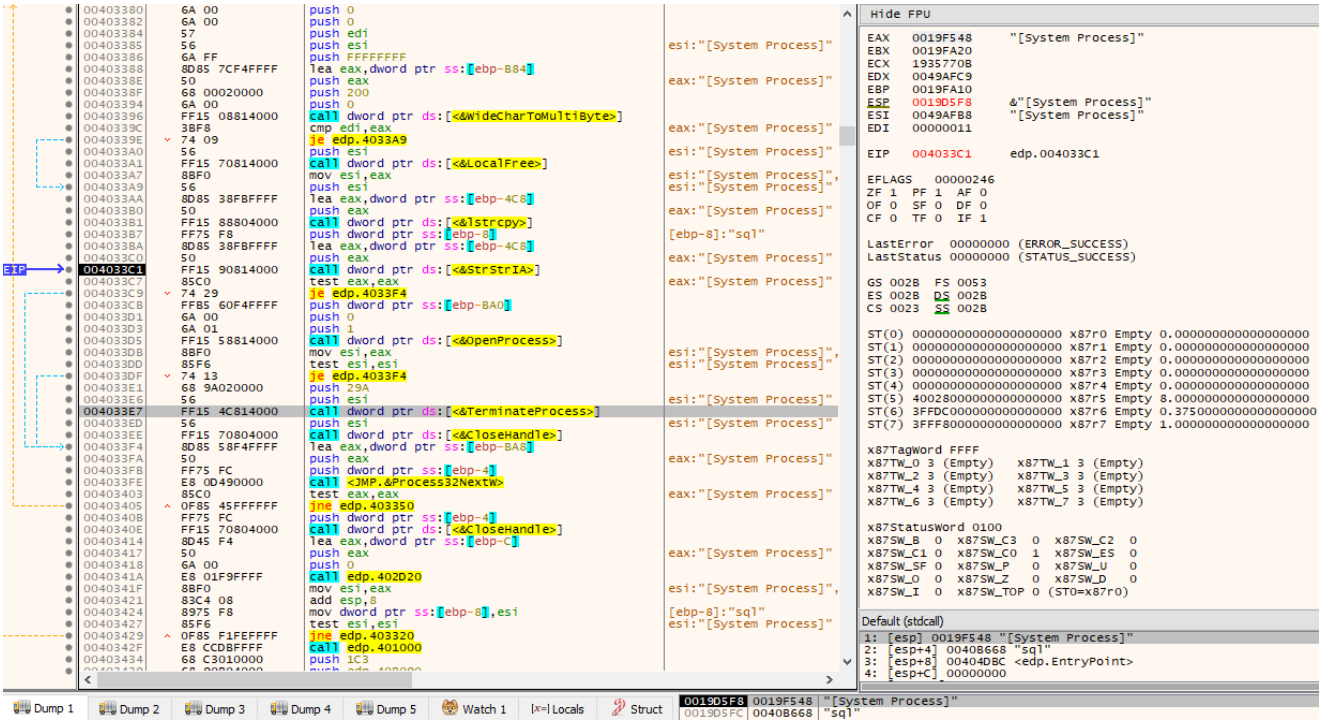
- If its dwCurrentState is SERVICE_STOPPED, it closes the service handle via CloseServiceHandle() and returns.
- If its dwCurrentState is SERVICE_STOP_PENDING, then it will enter a loop where it sleeps via Sleep() for either 10000 or 1000 milliseconds, calls QueryServiceStatusEx() again and exists the loop if dwCurrtentStats 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.40142B
004012FD    FF15 44814000    call dword ptr ds:[<&GetLastError>]
00401303    3D EA000000      cmp eax,EA
00401308    0F85 1D010000    jne edp.40142B
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.40142B
0040132B    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    75 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.40142B
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    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 CreateToolHelp32Spanshot() with dwFlags TH32CS_SNAPALL (and th32ProcessID 0), then processes are iterated via the Process32FirstW()/Process32NextW() combination.

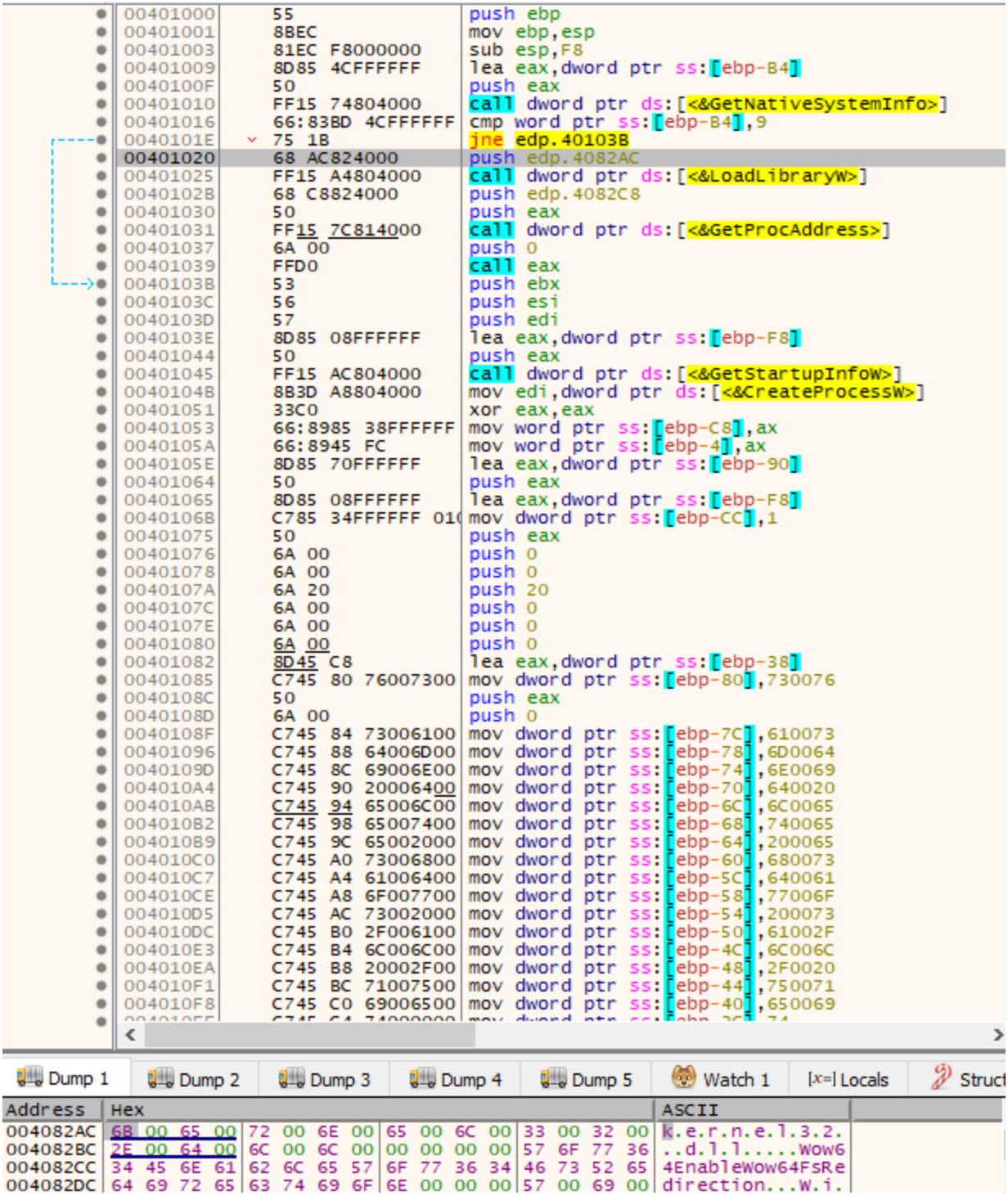


If the process name contains any substring as indicated by the decrypted data, then the process is opened via OpenProcess() with dwDesiredAccess of PROCESS_TERMINATE and terminated via TerminateProcess() with exit code 666.



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

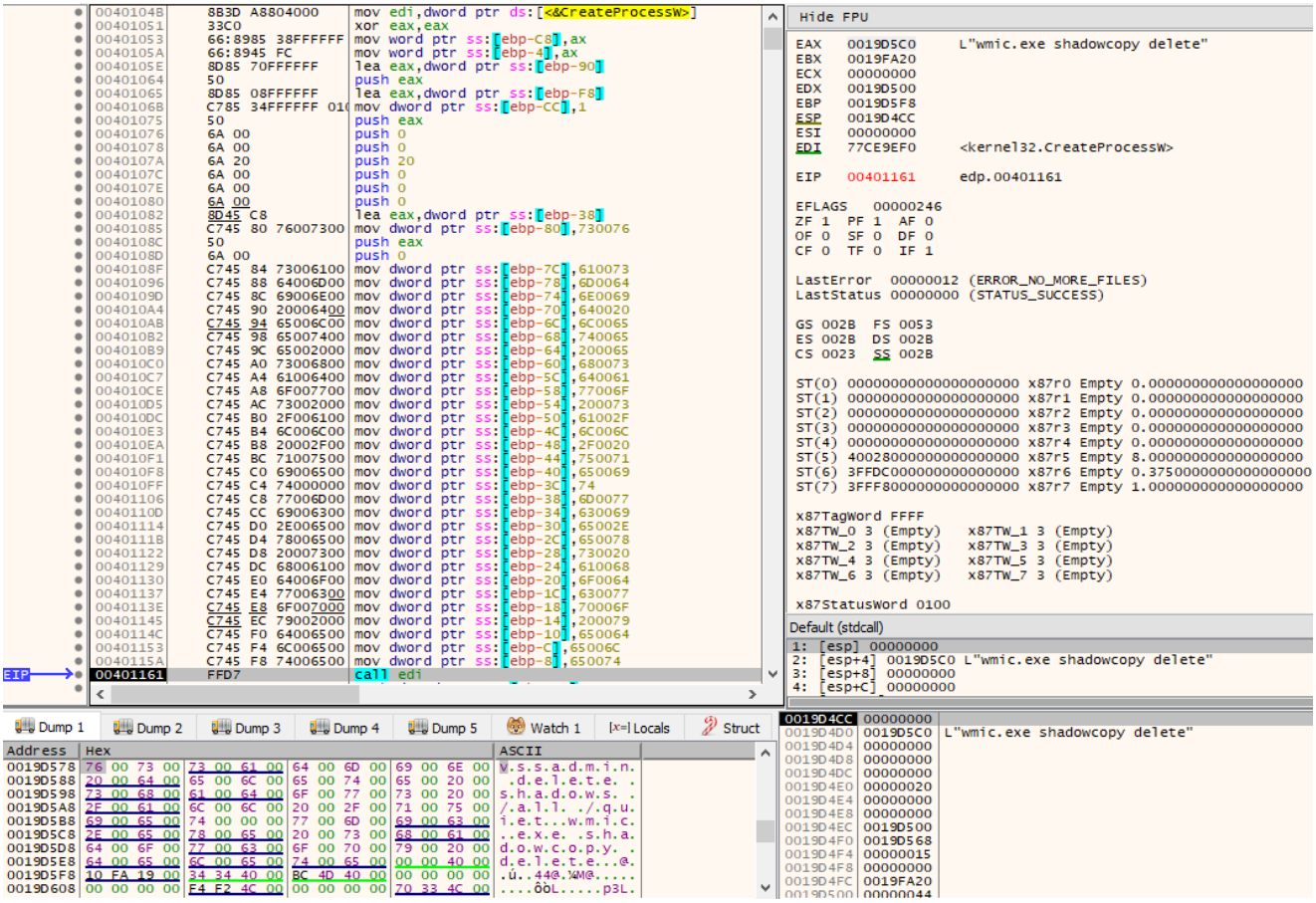
and the address of Wow64EnableWow64FsRedirection() is obtained via a call to GetProcAddress(). This WinAPI is then called with Wow64FsEnableRedirection set to FALSE, thus disabling WOW64 system folder redirection.



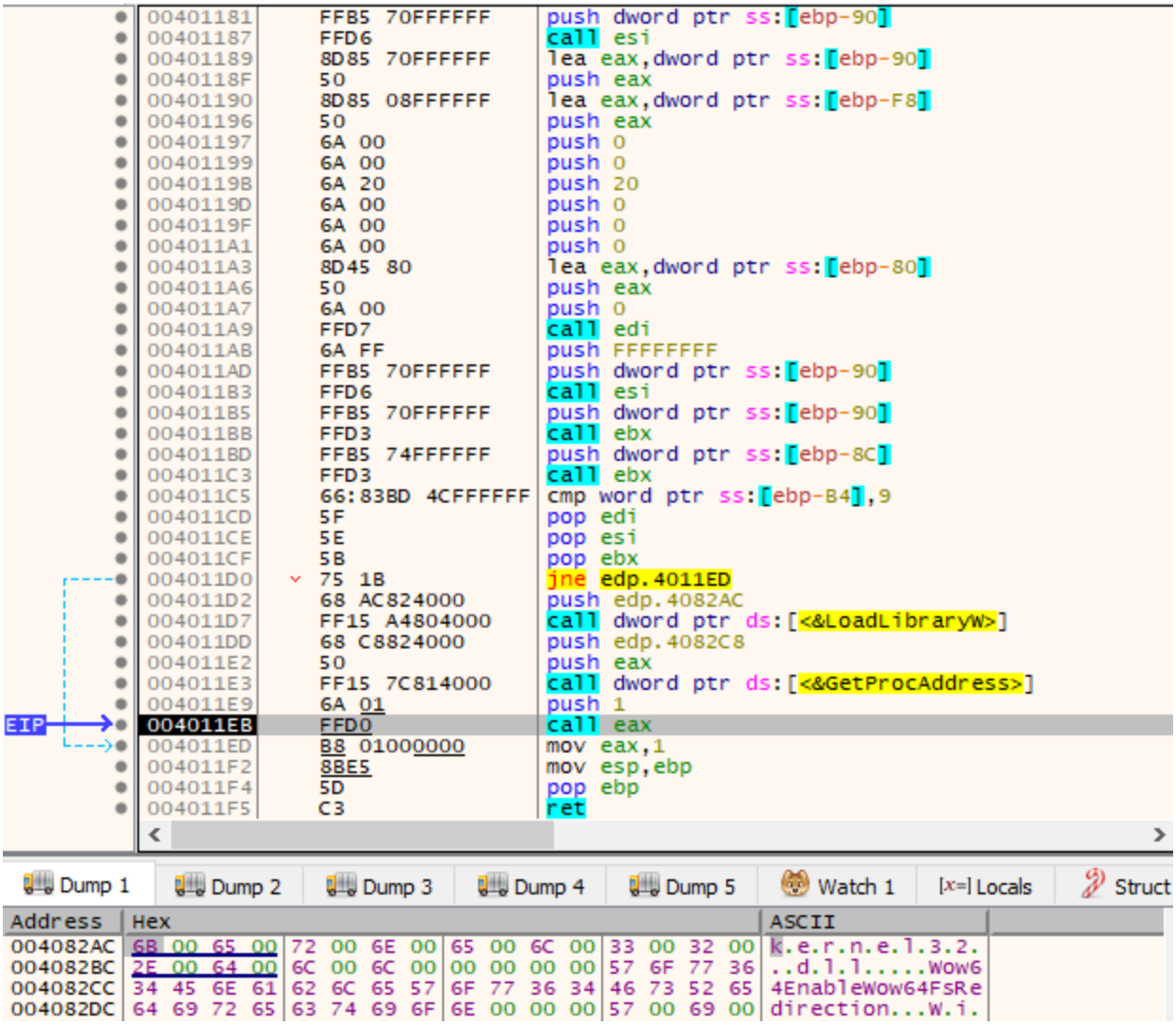
When redirection is disabled, two unicode strings are built on the stack by a series of mov instructions. These unicode strings will be used as lpCommandLine for subsequent calls to CreateProcessW(). The executed command lines are:

- wmic.exe shadowcopy delete

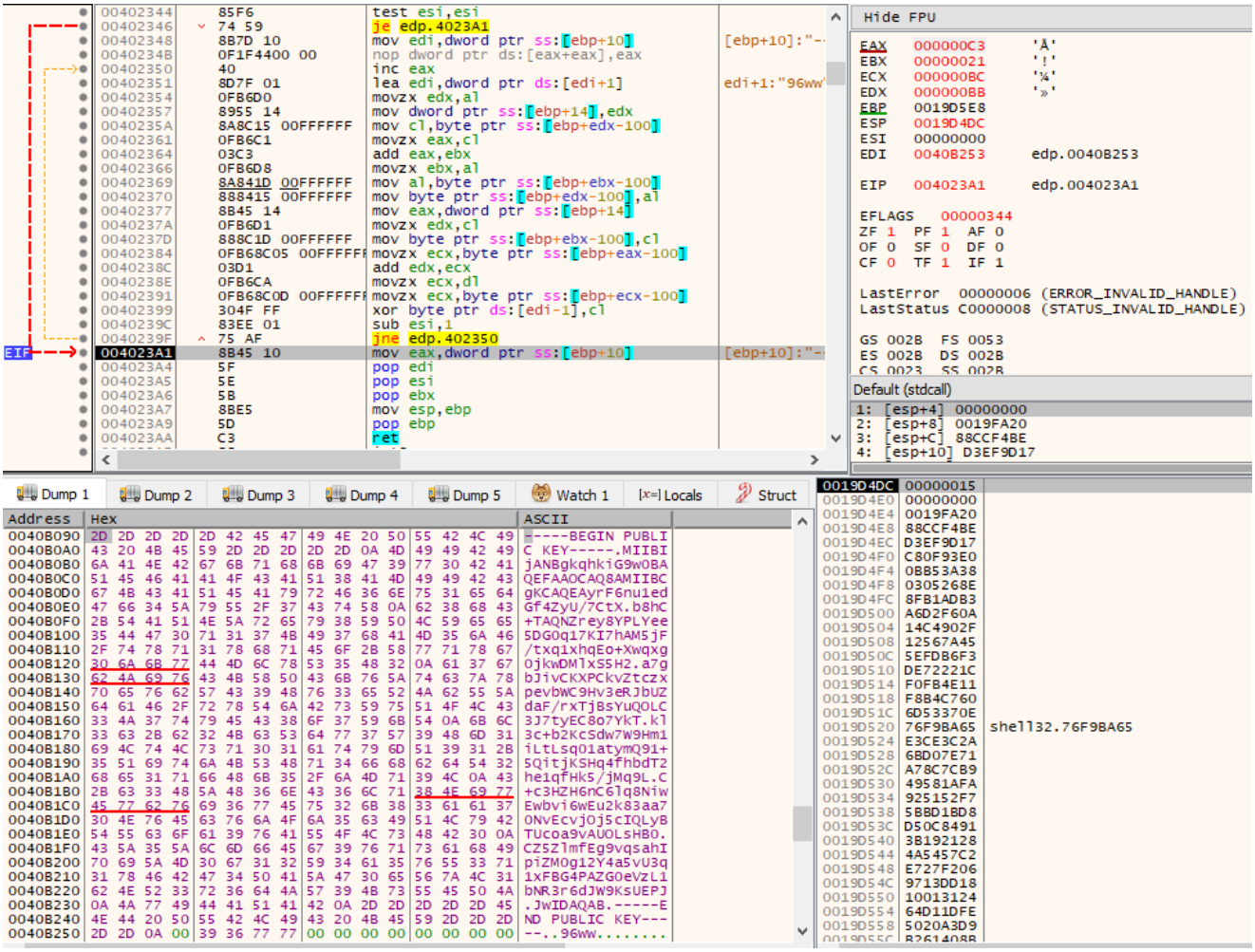
- vssadmin delete shadows /all /quiet



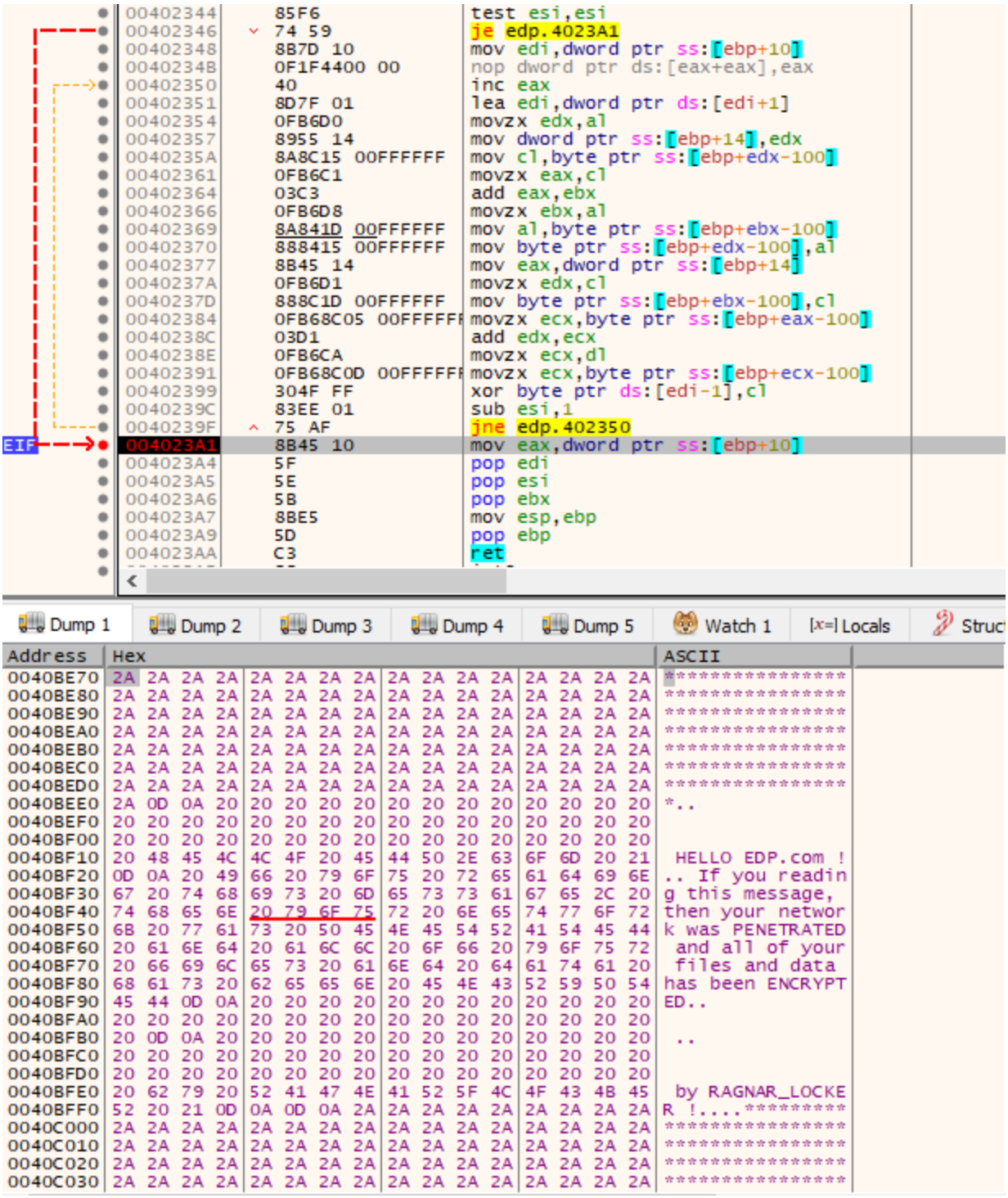
Right after shadow copy deletion, LoadLibraryW(L"kernel32.dll") is called once again and Wow64EnableWow64FsRedirection() is obtained via GetProcAddress(), this time in order to be called with Wow64FsEnableRedirection set to TRUE, thus enabling WOW64 system folder redirection. The routine then returns.



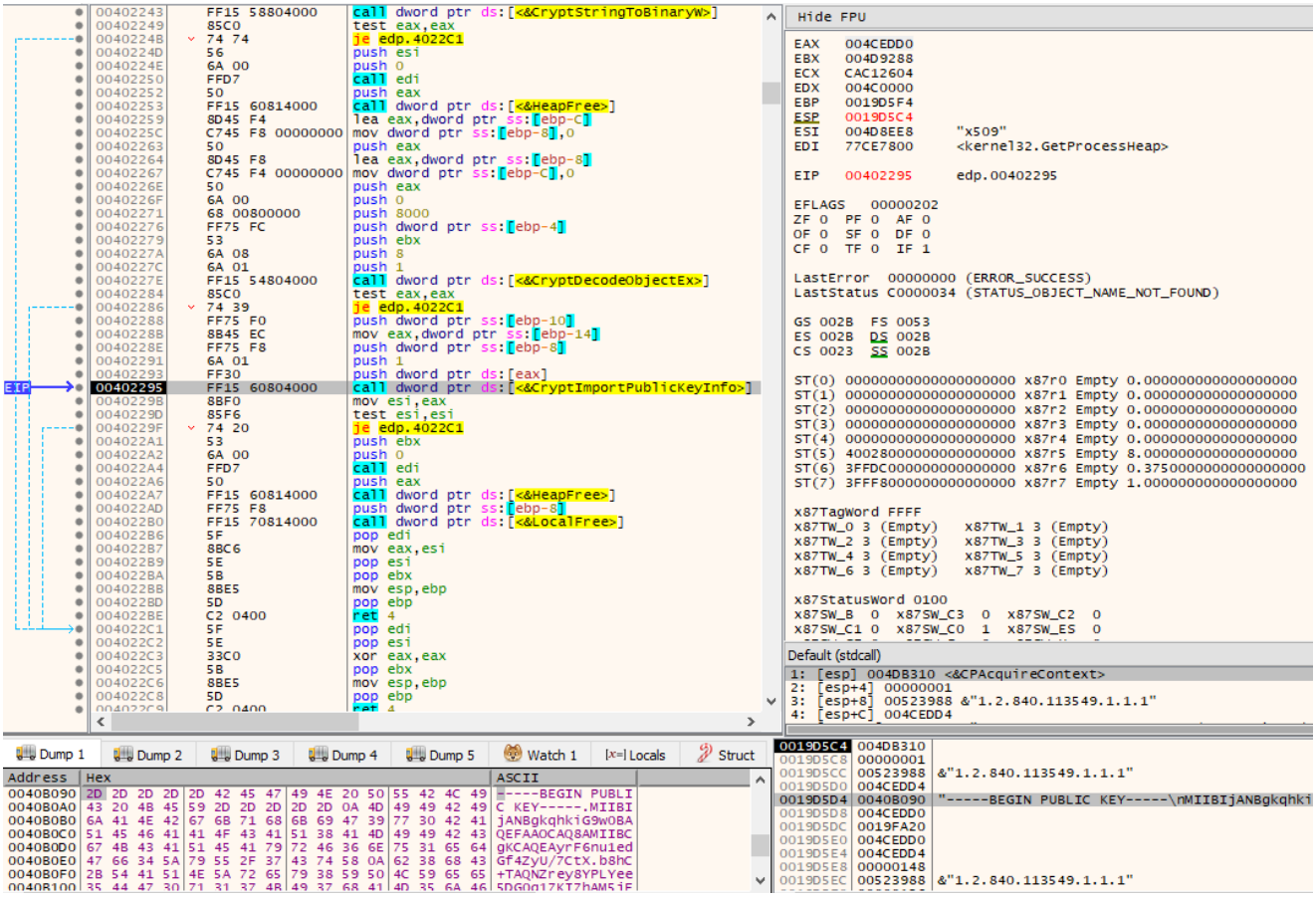
It is now time for some more data decryption from the .keys section. This time, the data that is decrypted is a 2048-bit RSA public key. We will see how it will be used later.



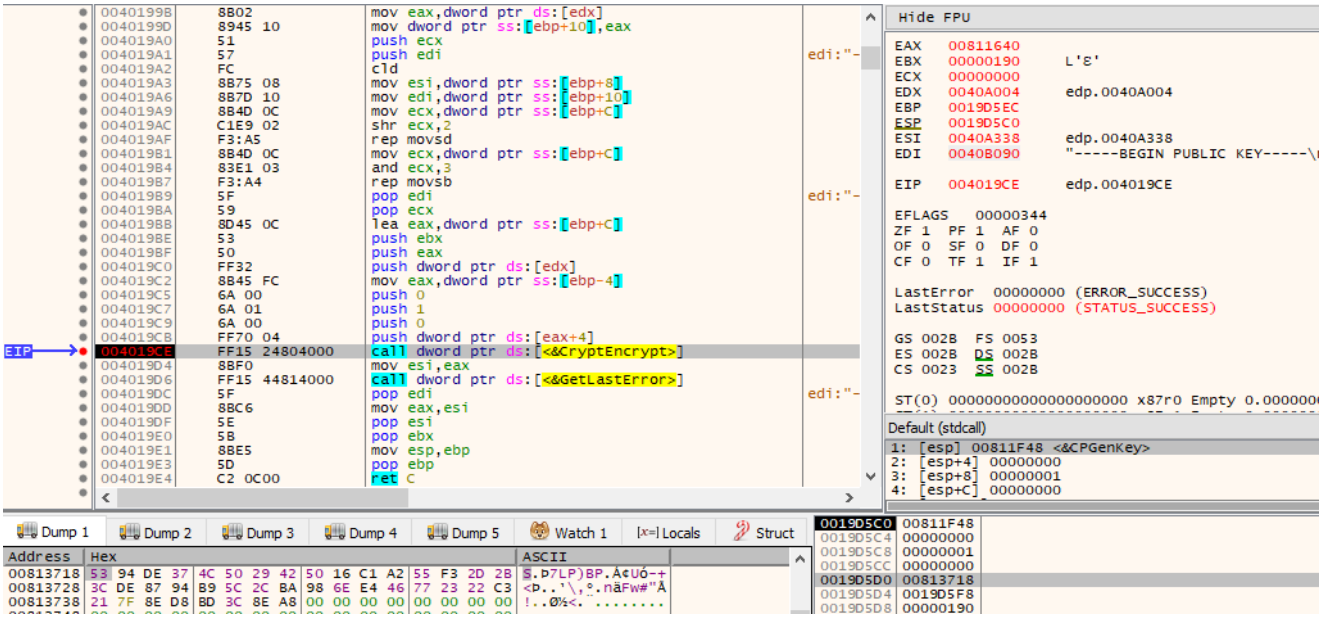
Other data that is decrypted, by another call to the routine, is the final ransom note. Please refer to the following image.



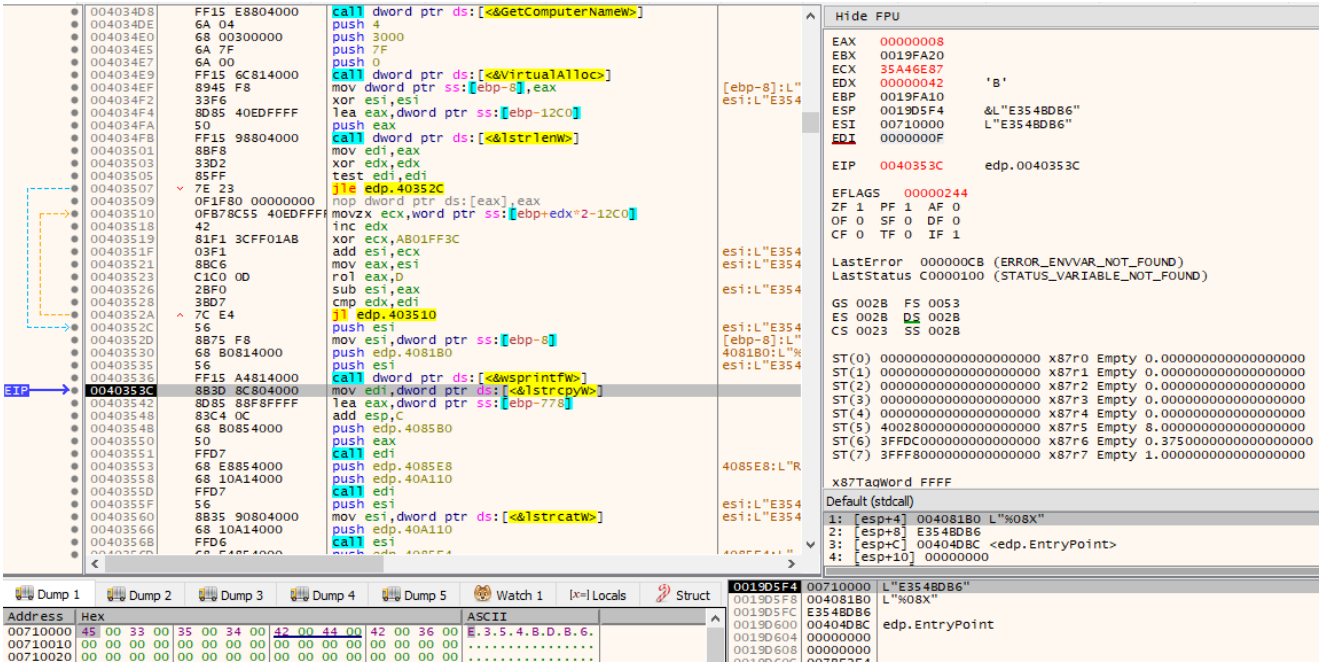
The 2048-bit RSA public key is then converted and its public key information imported via CryptImportPublicKeyInfo() into the provider.



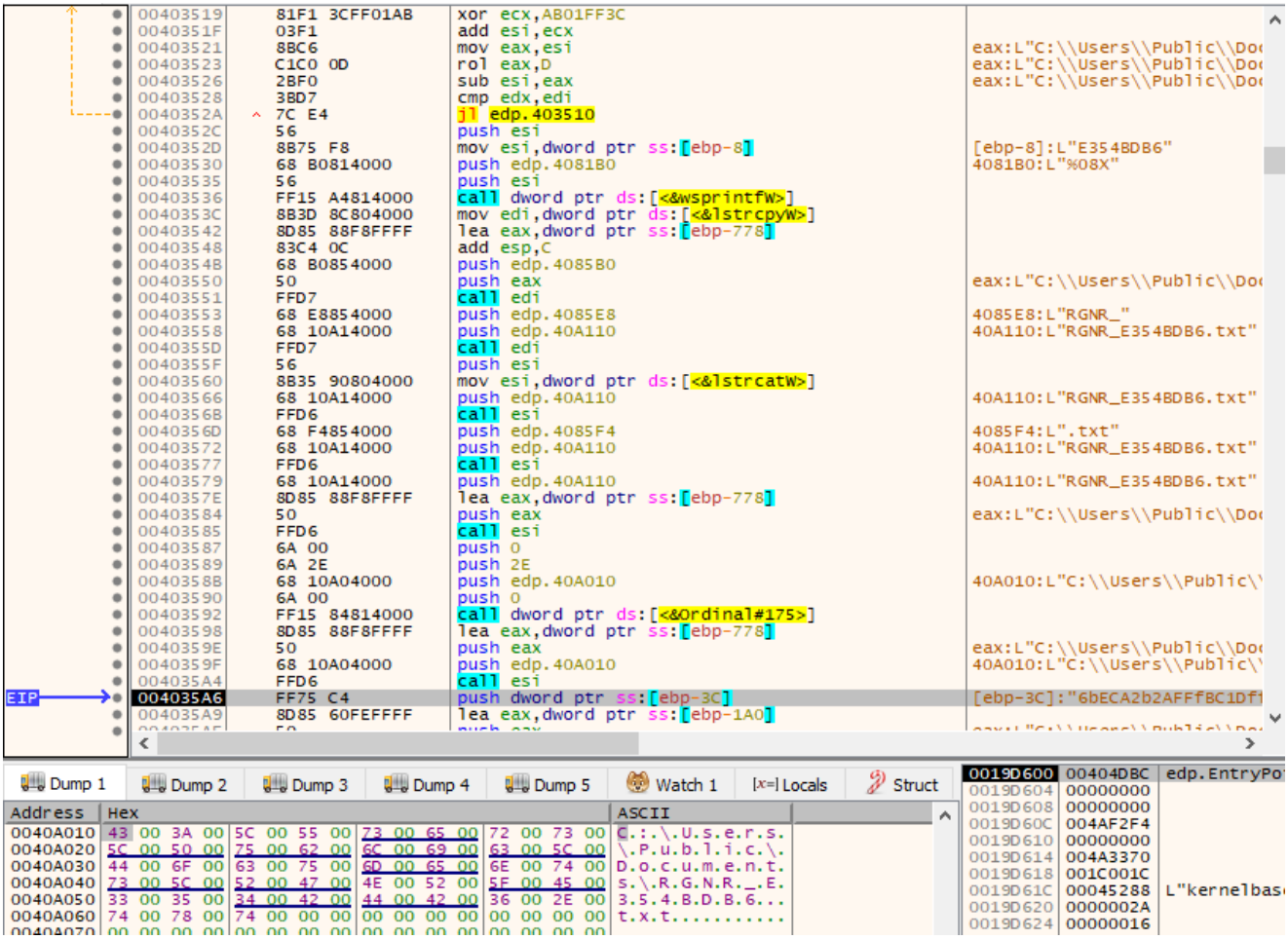
By calling a subroutine twice that calls CryptEncrypt(), the previous two cryptographically random data that were generated by both calls to CryptGenRandom(), which were subsequently modified through a series of operations, will be encrypted with the 2048-bit public key.



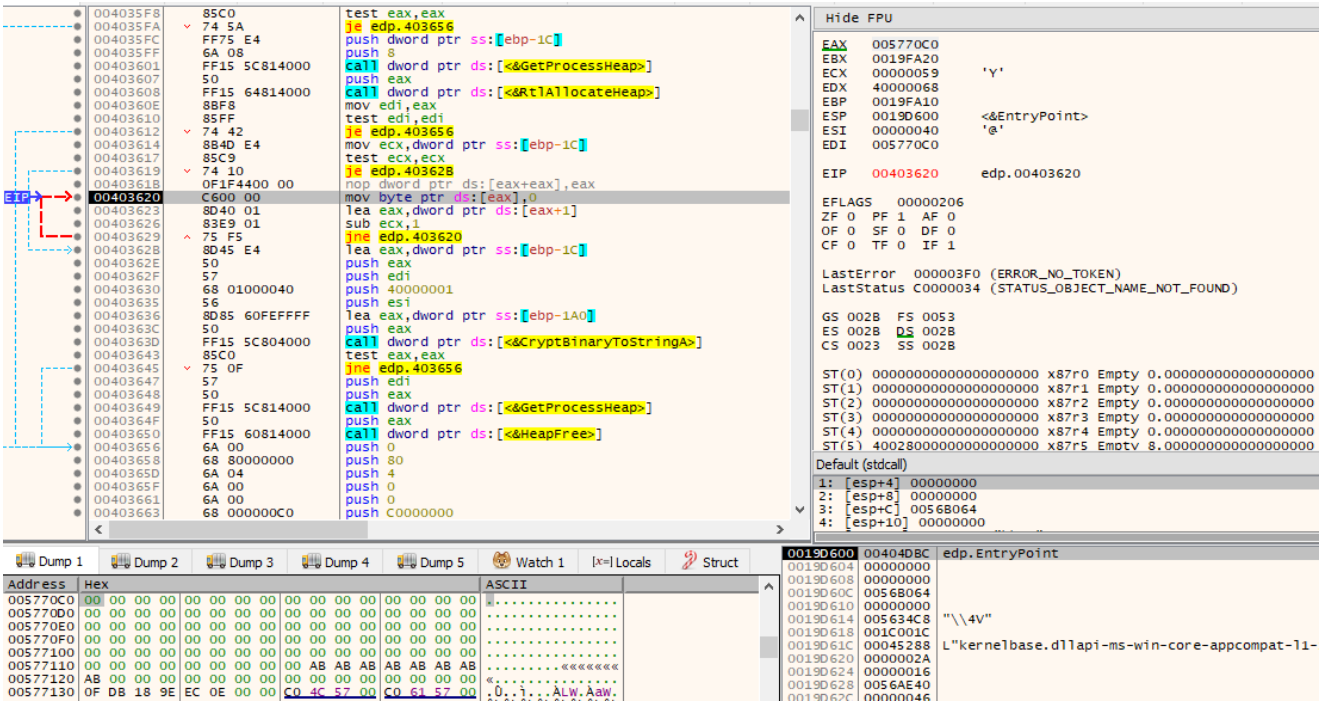
Via a call to GetComputerNameW(), and through the same series of operations that were used to generate unique IDs for the CreateEventW() even object name (lpName), an hex encoded ID is generated.



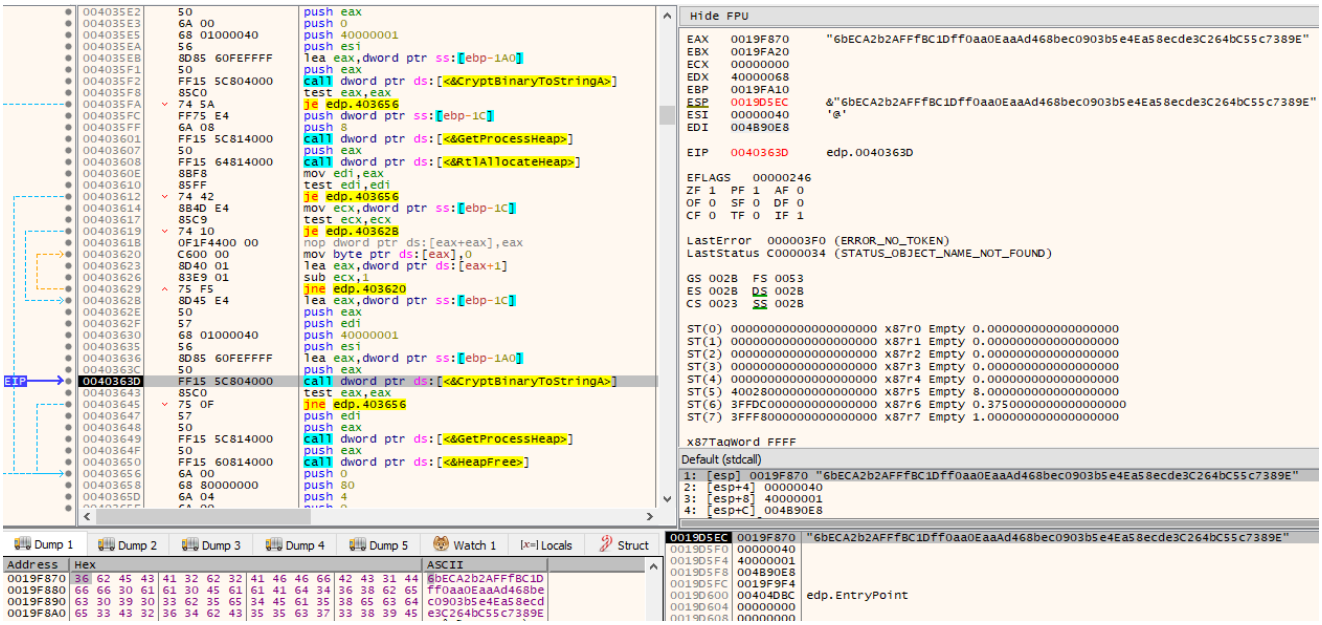
Through concatenation, by making use of lstrcatW(), and a call to SHGetSpecialFolderPathW() with csidl CSIDL_COMMON_DOCUMENTS, the path C:\Users\Public\Documents\RGNR_E354BDB6.txt is built.



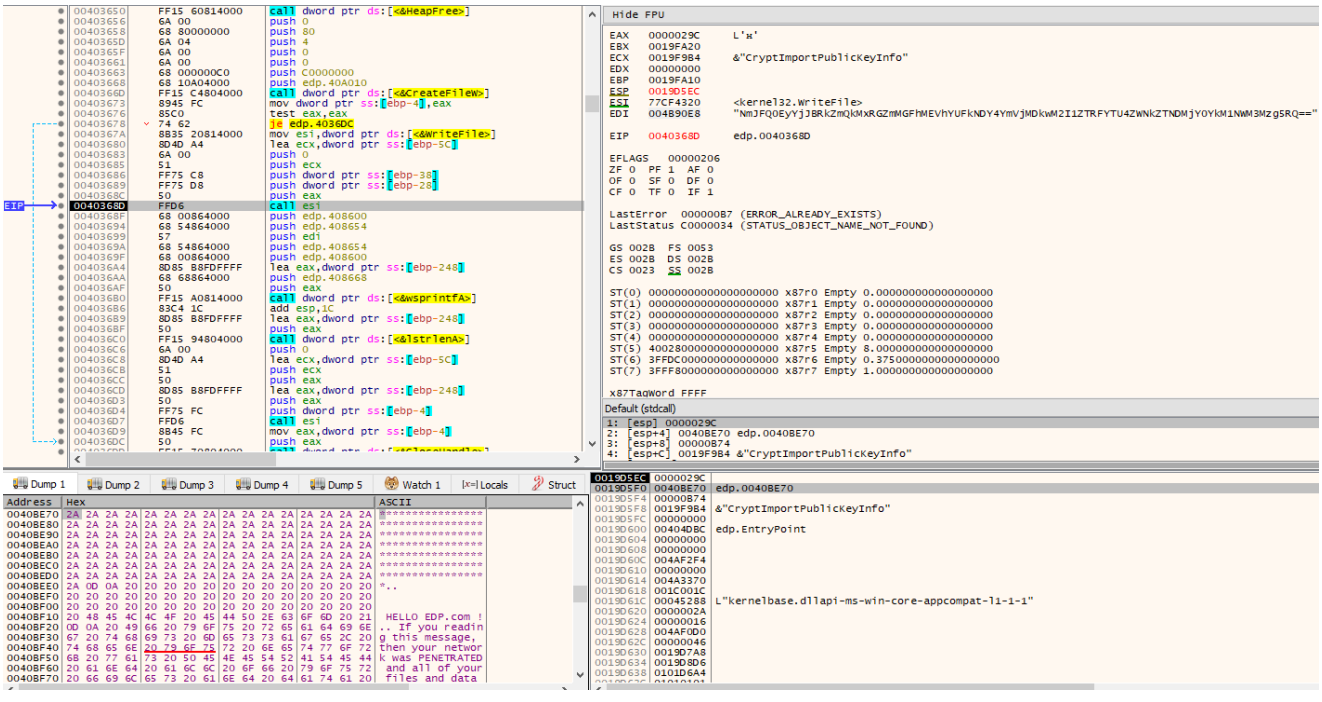
Along the way, a block of heap memory allocated via RtlAllocateHeap() is called with HEAP_ZERO_MEMORY as Flags, which initializes it with 0's. For some reason, this memory area will be, again, zeroed out after the call to RtlAllocateHeap().



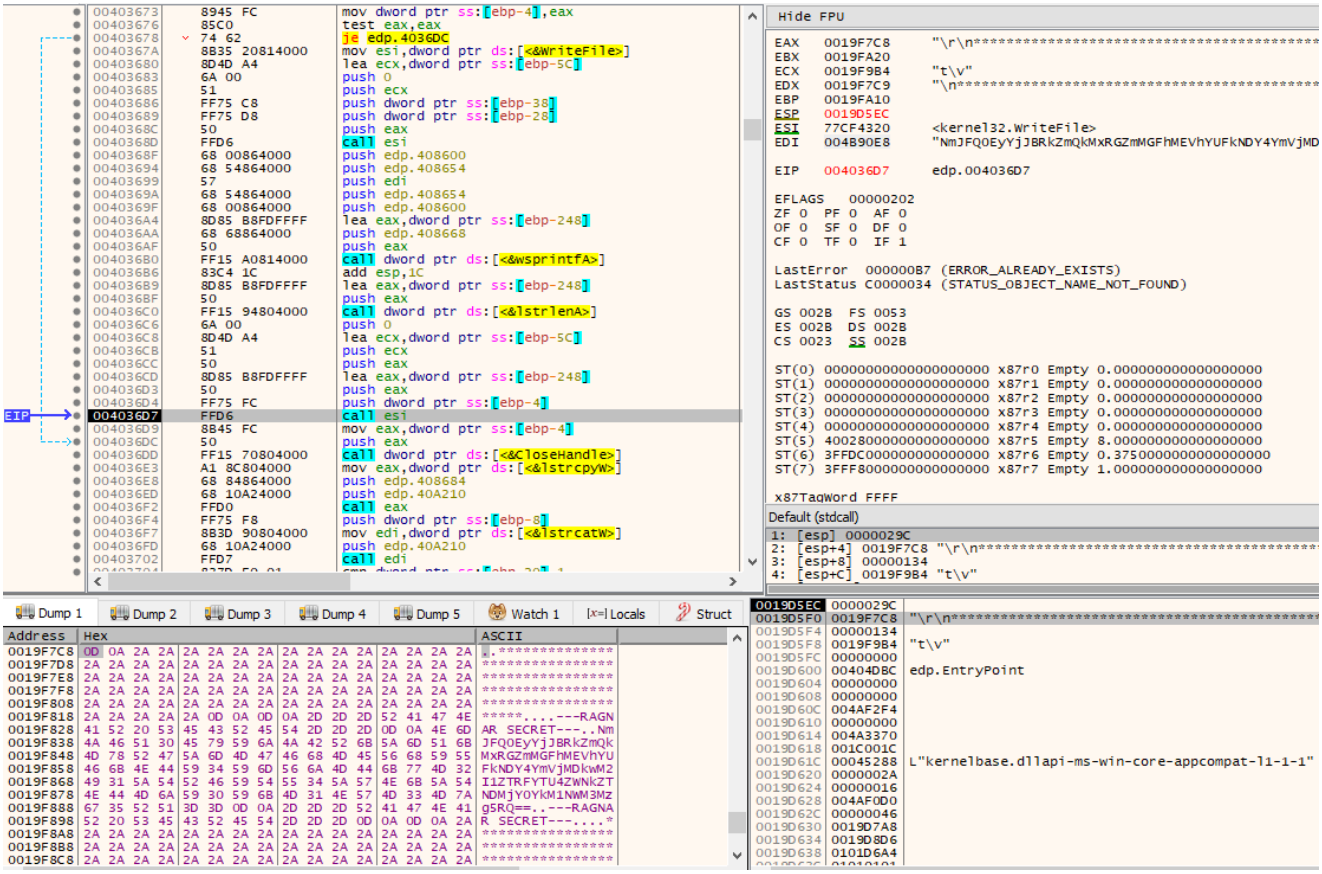
The Tor client chat ID previously decrypted is then converted to Base64, by making a call to CryptBinaryToStringA(), as seen by the use of dwFlags set to CRYPT_STRING_BASE64.



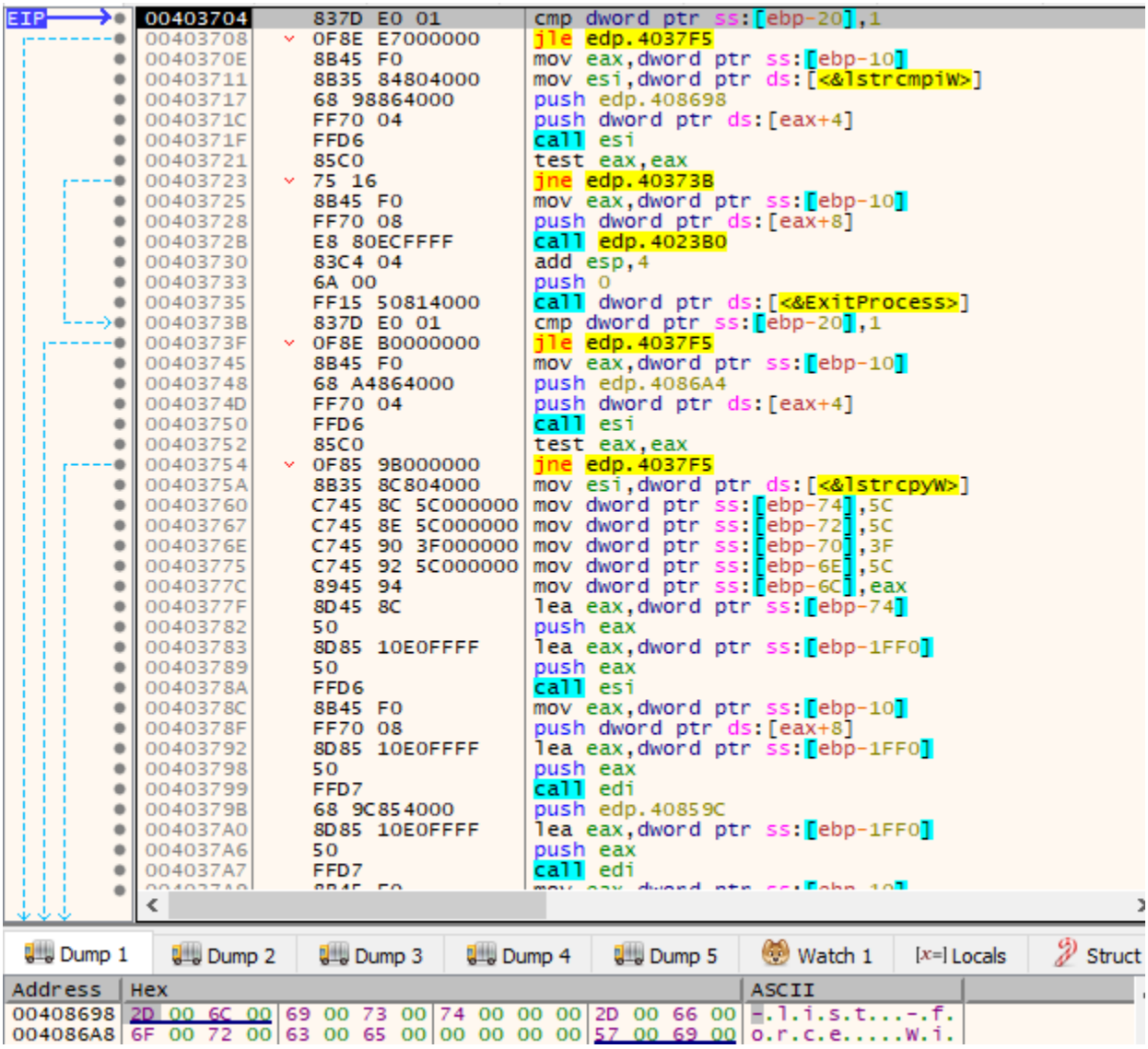
The previously decrypted specifically targeted ransom note that will be left in the attacked systems is then written via WriteFile() to the C:\Users\Public\Documents\RGNR_E354BDB6.txt path that had been built moments prior, by first opening it via CreateFileW() with dwDesiredAccess of GENERIC_READ |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



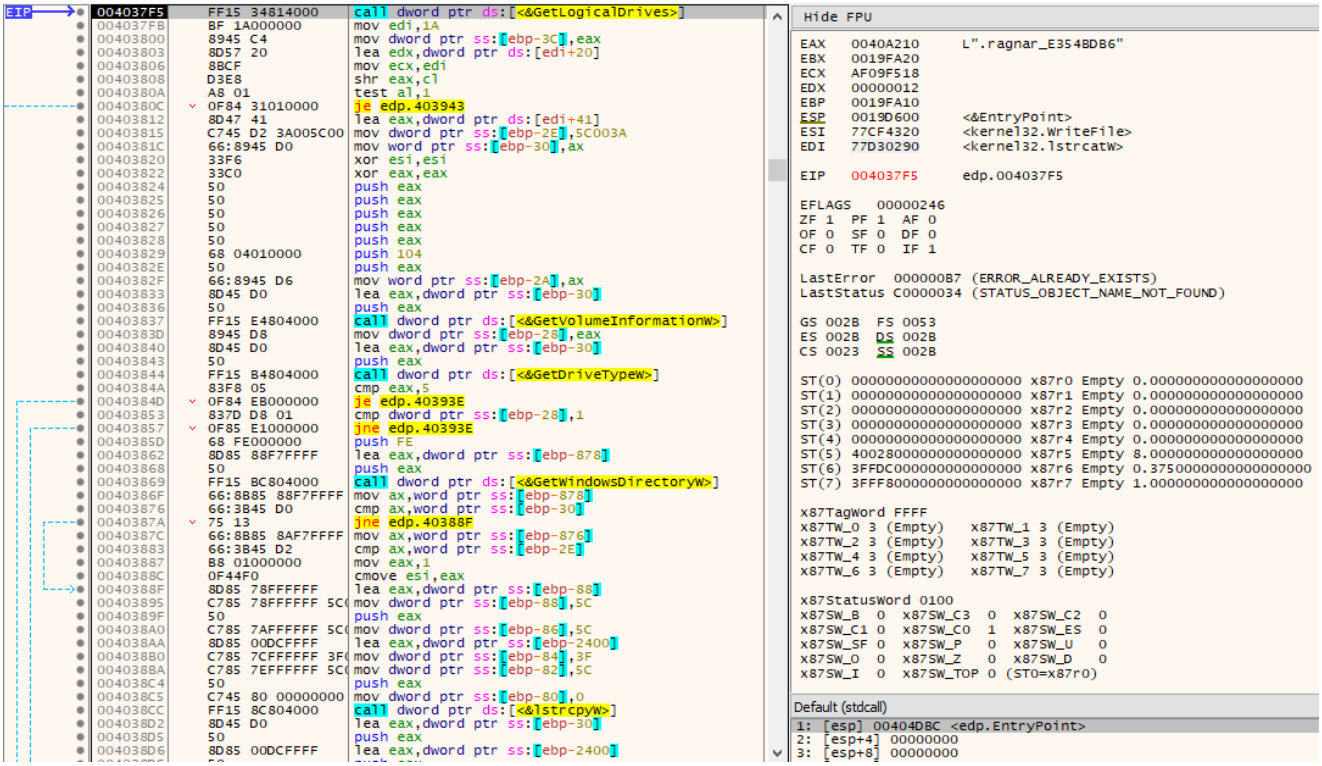
Then, through concatenation, the "RAGNAR SECRET" will be appended to the file, which is simply the Base64 encoded version of the Tor client chat ID.



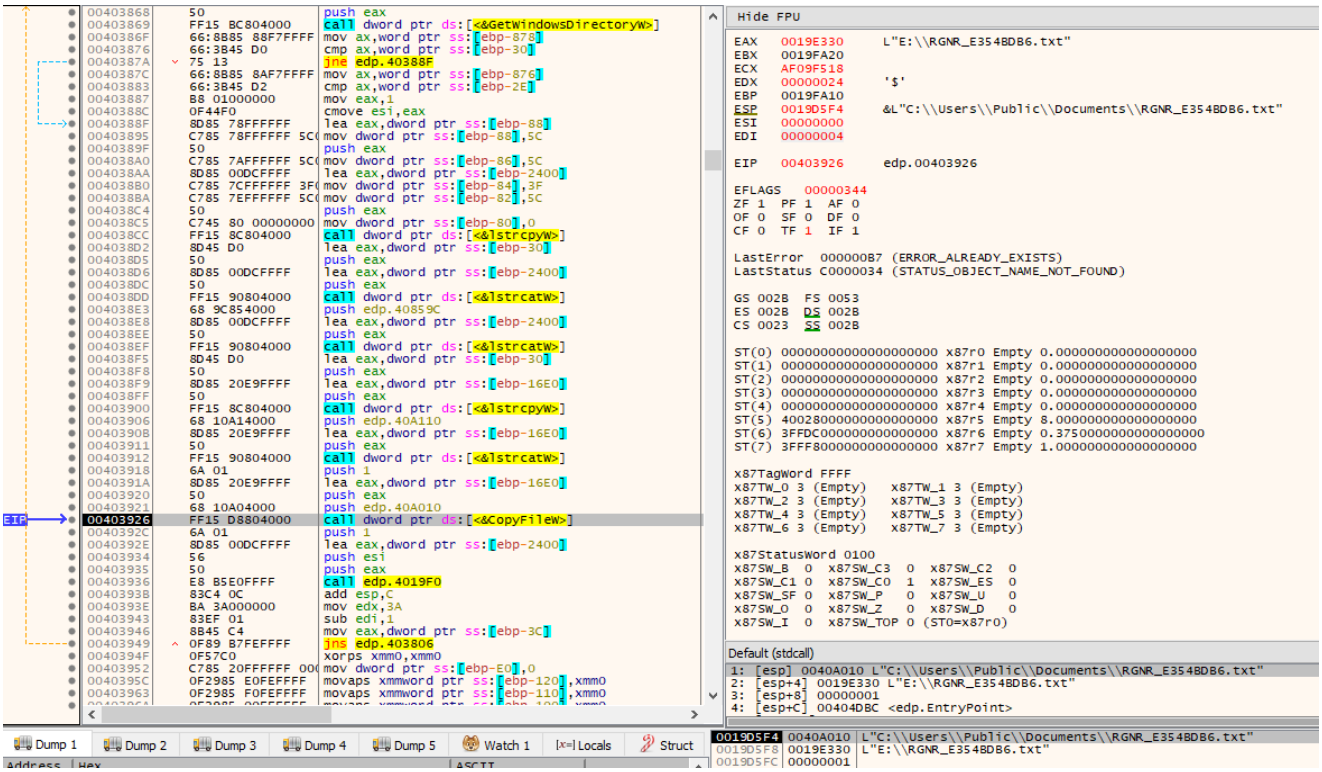
After the file with the ransom note has been written to, the ransomware will check if argc (argument count) is higher than 1. The ransomware can be executed with "-list" or "-force" command line options. These are simply used to determine how the paths that will be used as base to start file encryption 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.



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.

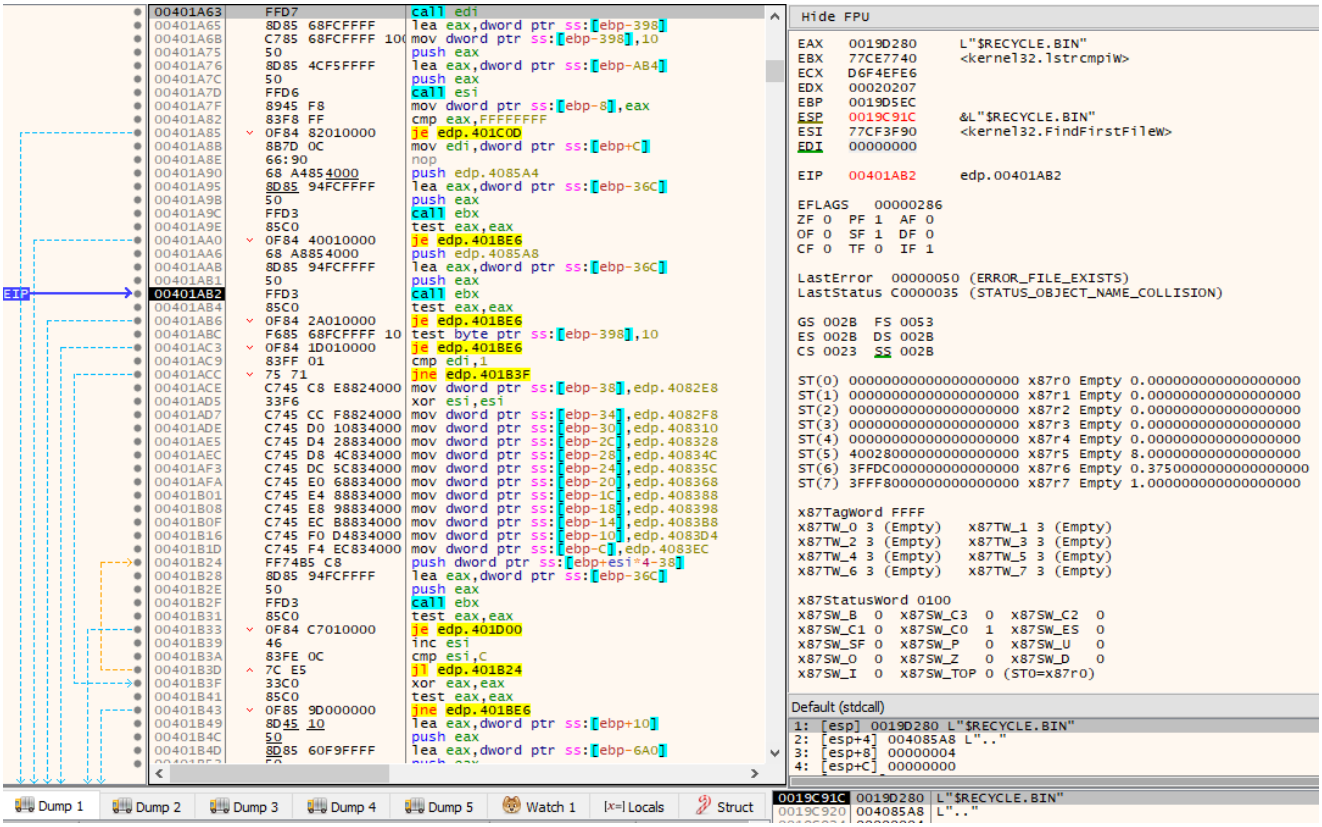


The file containing the ransom note is then copied into this newly obtained path.



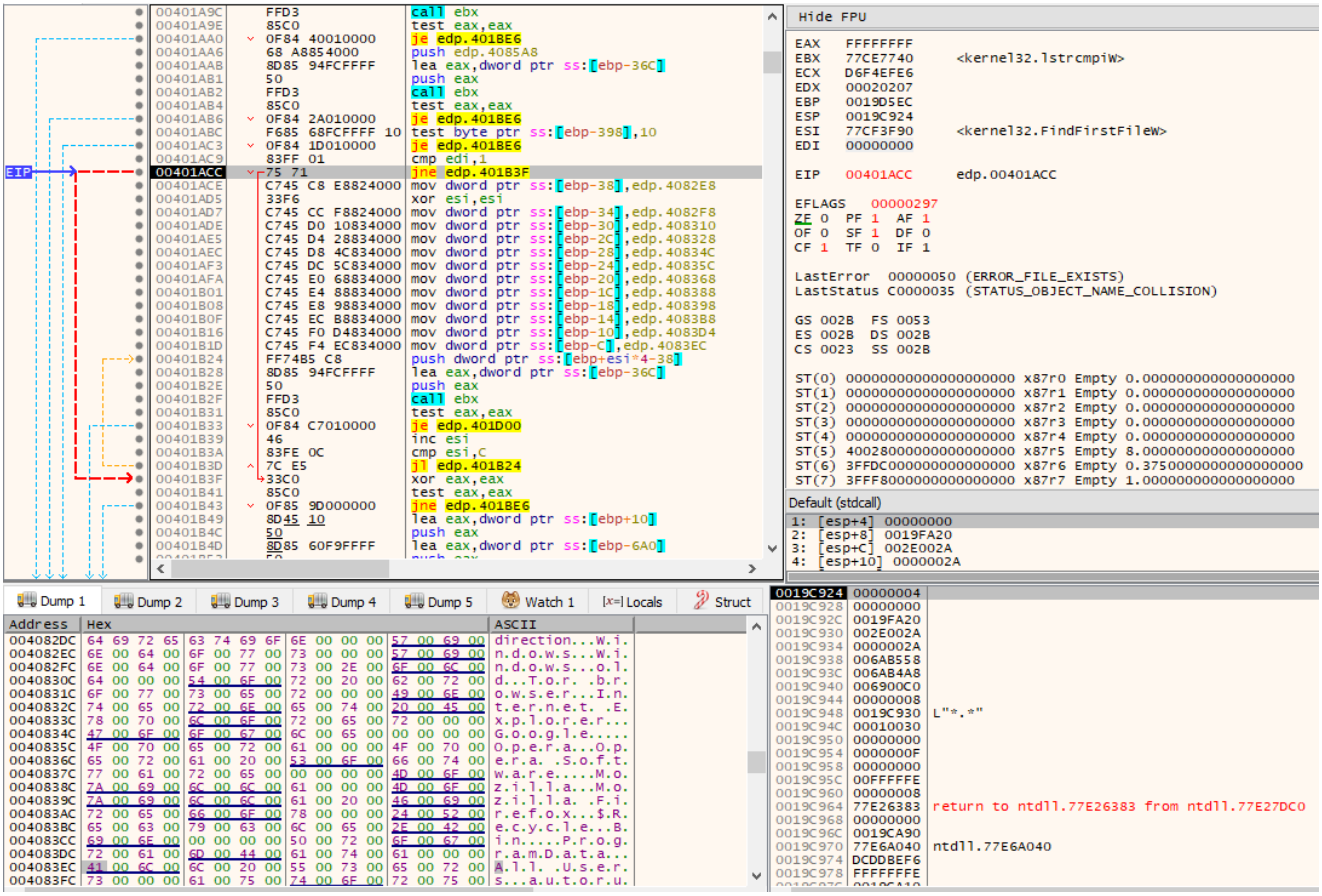
After the file is copied, a subroutine will be called with this new path as argument. One of the other arguments to this subroutine is the integer being treated as a flag to indicate whether the drive letter of the current path matches the drive letter being used where the WindowsDirectory is located. 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.

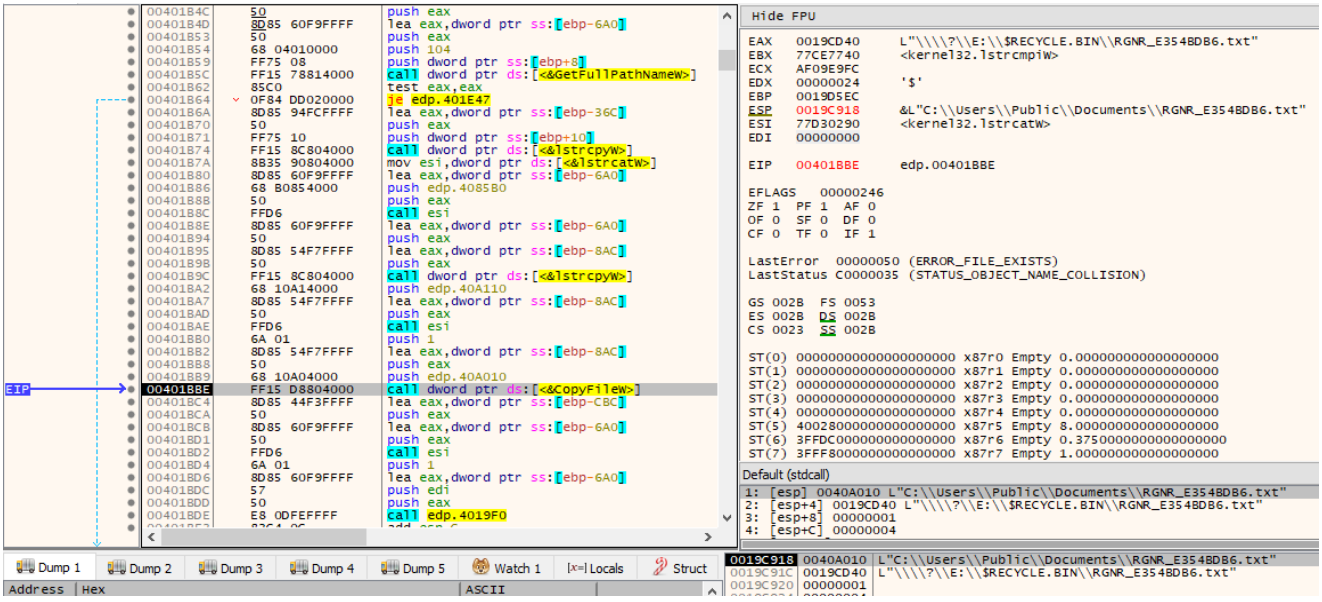


The checks that take place when the flag passed as argument is set occur so that certain directories are skipped and nothing will be done on them. The directories that are compared against the currently obtained directory are:

- 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:

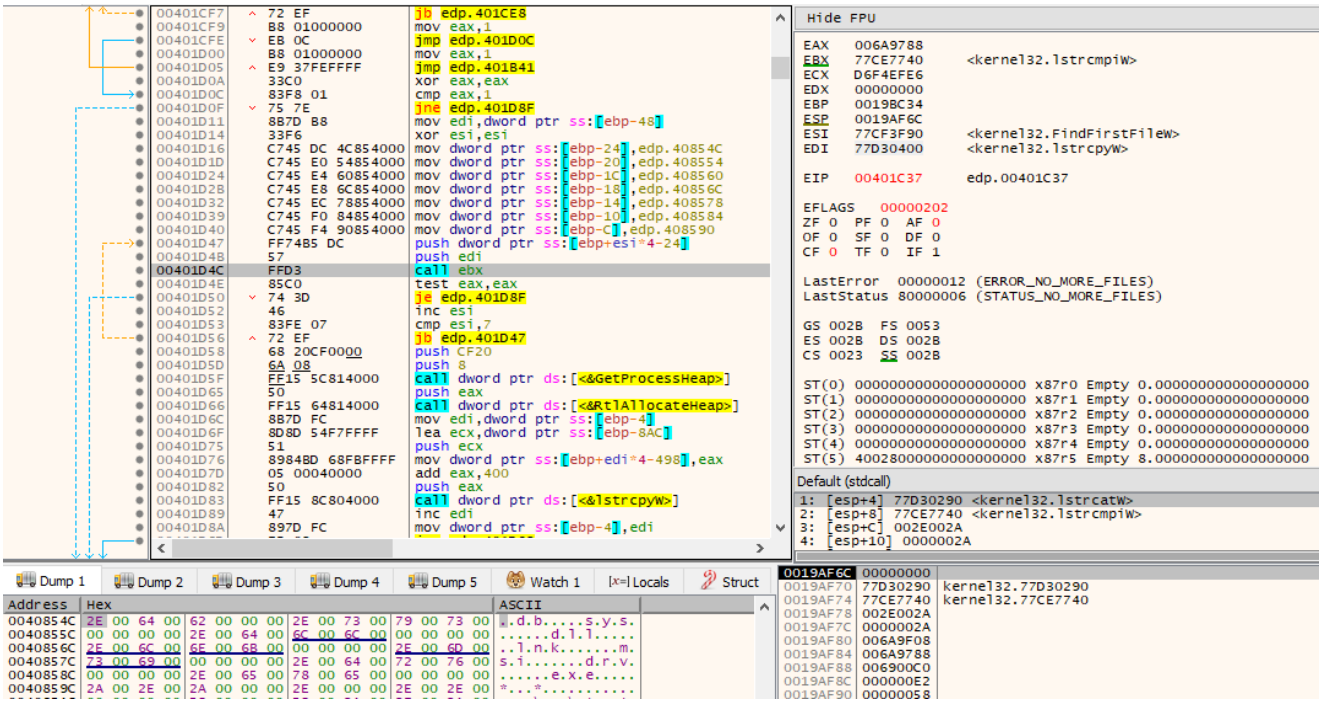
- The ransom note filename
- autorun.inf
- boot.ini
- bootfont.bin
- bootsect.bak
- bootmgr
- bootmgr.efi
- bootmgfw.efi
- desktop.ini
- iconcache.db
- ntldr
- ntuser.dat
- ntuser.dat.log
- ntuser.ini
- thumbs.db

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

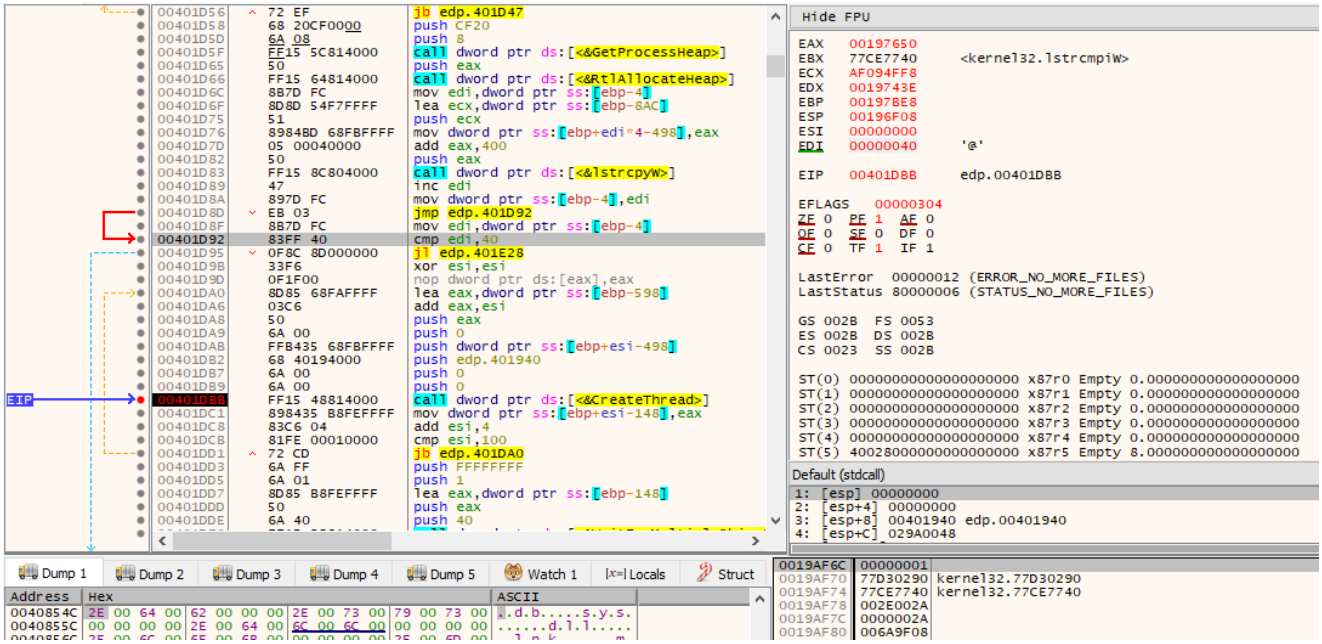
If the currently found file's name does not match the above list of filenames, then extension checks will also be performed. Specifically, the current file's extension is checked against:

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

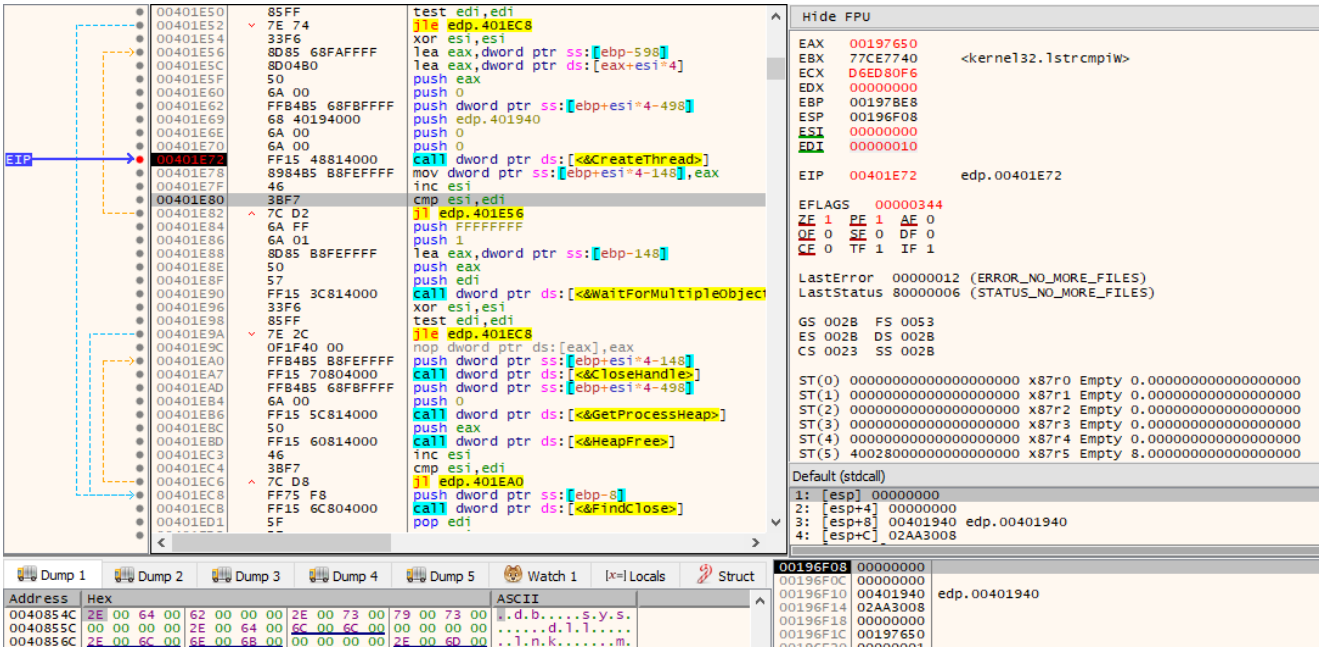
If the extension of the current file's name matches any of the above list of extensions, then nothing is done with it and it is skipped. If it doesn't match, however, the pointer to the file's name will be added into a stack array.



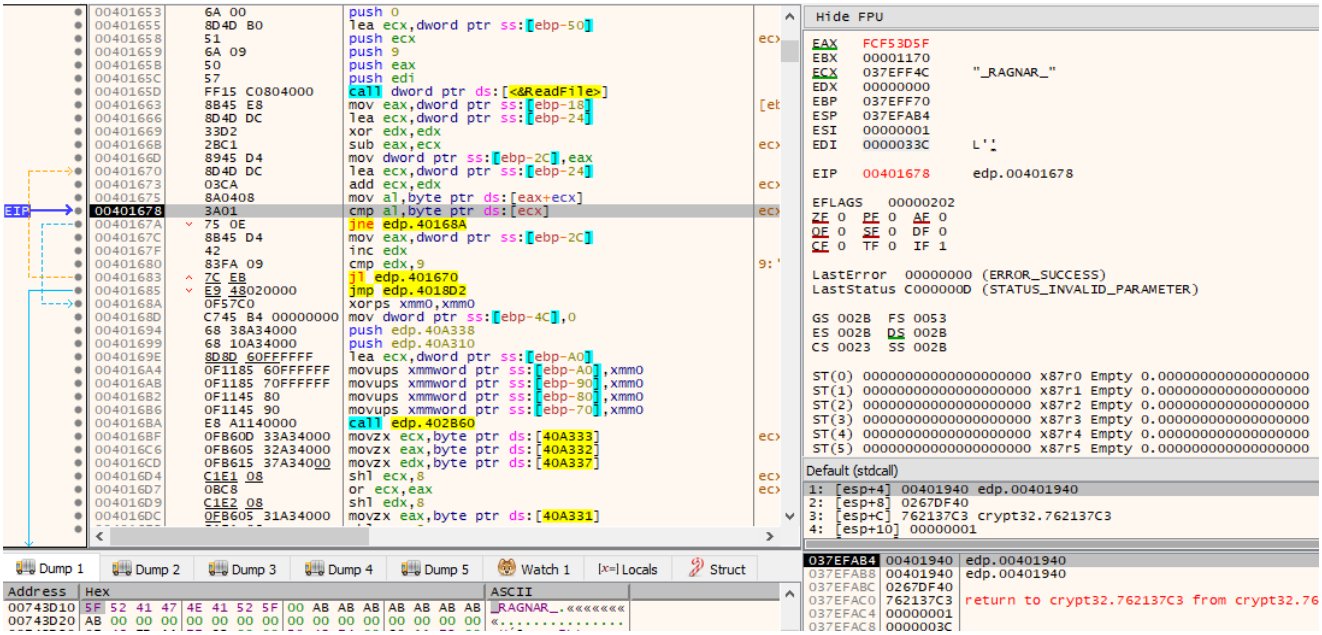
If 64 files in the current directory under examination have been added into the stack array (thus passing all of the above checks), then 64 threads will be created via CreateThread(). Each of the 64 pointers in the stack array are passed as lpParameter and the routine that handles file encryption is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



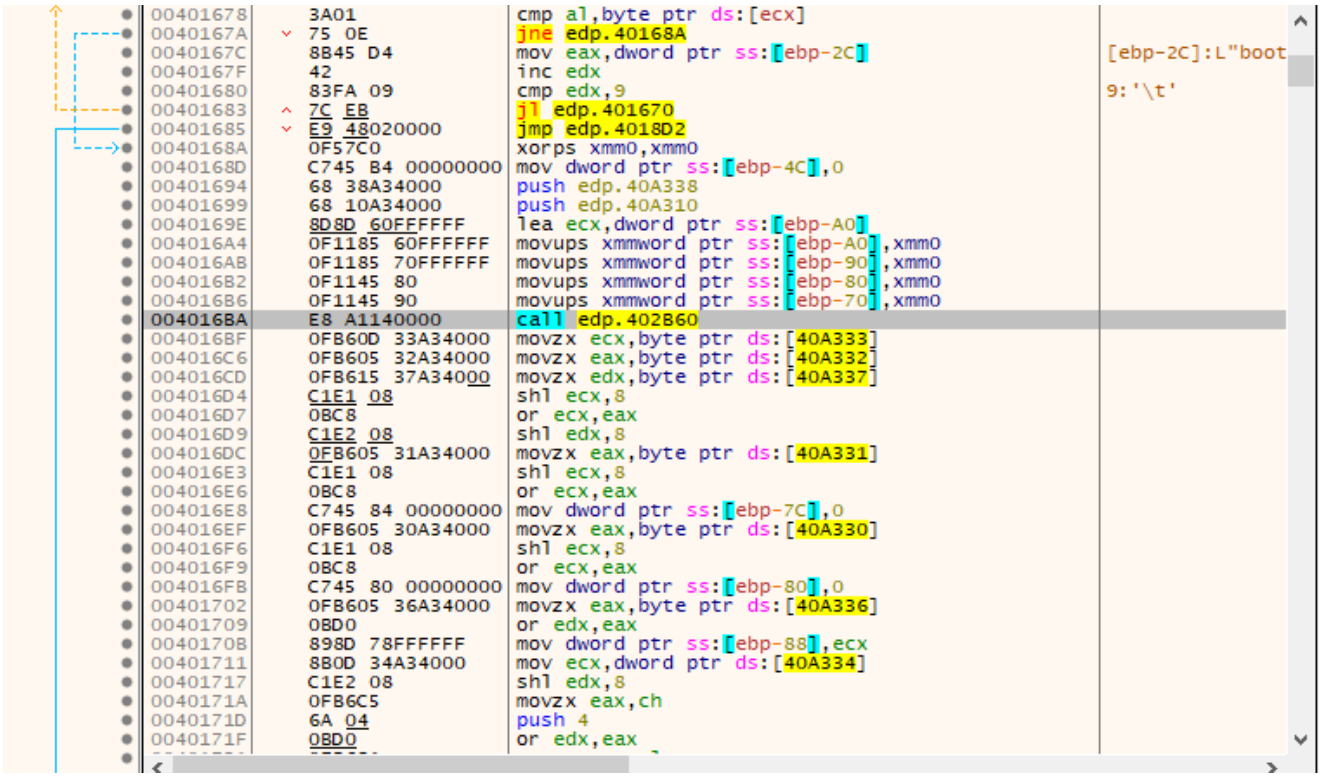
If, at the end of examination of the directory, the number of files in the stack array are less than 64, then a thread will be created via CreateThread for each of the files. Each of the pointers in the stack array are passed as lpParameter and the routine that handles file encryption is also passed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



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.



If the marker is not found, a routine is called which performs a series of operations on both cryptographycally random bytes resulted from the calls to CryptGenRandom() (although immediatly 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.



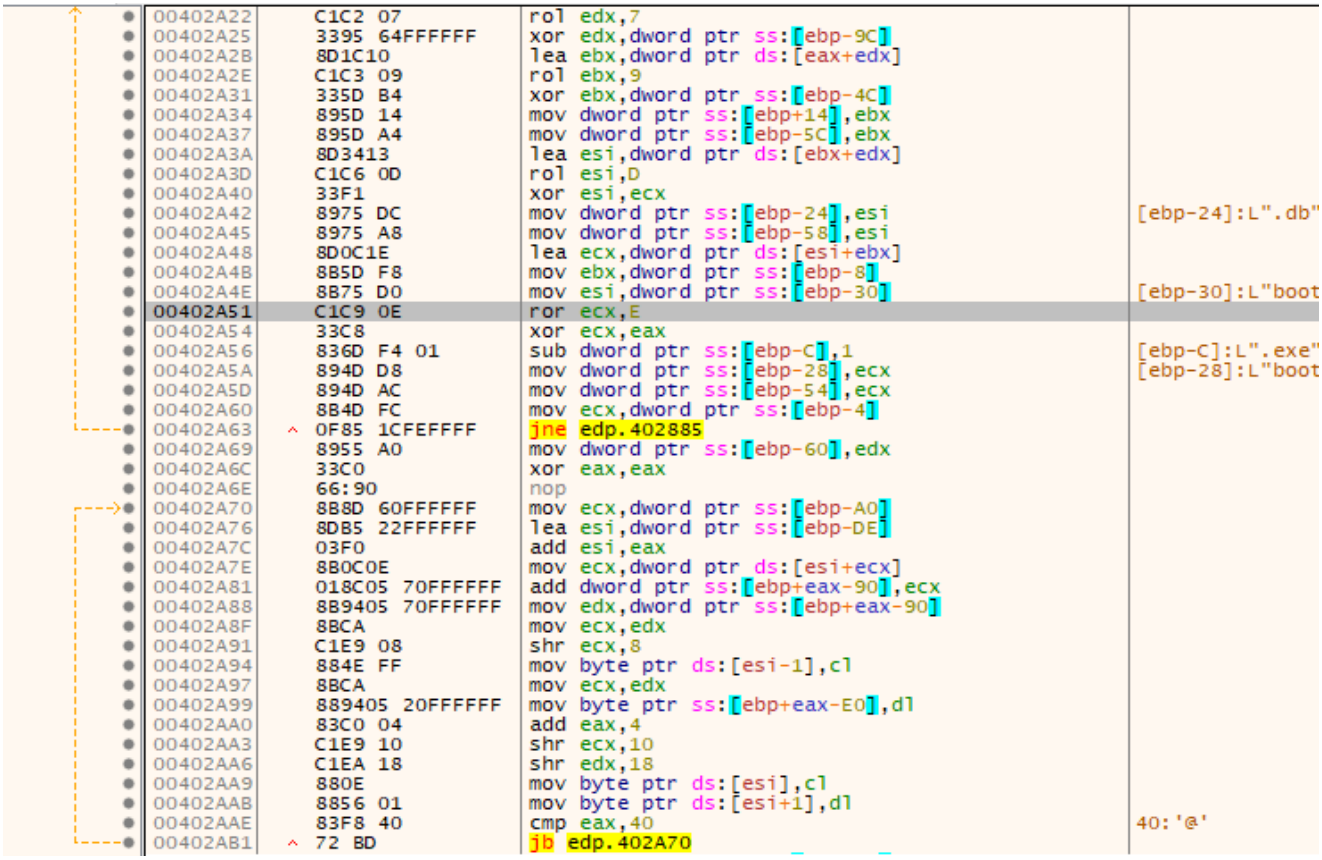
A teaser on these operations is demonstrated in the following image.

```
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:[<&ReadFile>]
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:[<&LockFile>]
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:[<&WriteFile>]
0040181D    6A 00           push 0

Hide FPU

EAX   00000000
EBX   77CE7740    <kernel32.lstrcmpiW>
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) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 40028000000000000000 x87r5 Empty 8.000000000000000000

Default (stdcall)
1: [esp+4] 77D30290 <kernel32.lstrcatW>
2: [esp+8] 77CE7740 <kernel32.lstrcmpiW>
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.

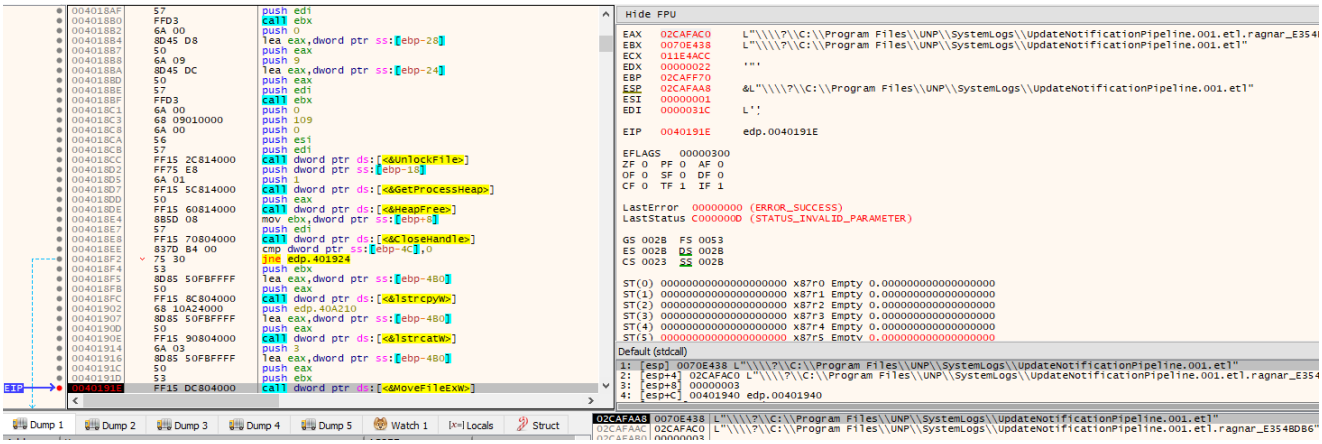


```
00402A22    C1C2 07         rol edx,7
00402A25    3395 64FFFFFF   xor edx,dword ptr ss:[ebp-9C]
00402A2B    8D1C10          lea ebx,dword ptr ds:[eax+edx]
00402A2E    C1C3 09         rol ebx,9
00402A31    335D B4         xor ebx,dword ptr ss:[ebp-4C]
00402A34    895D 14         mov dword ptr ss:[ebp+14],ebx
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:'@'
00402AB1  ^ 72 BD           jb edp.402A70
```

When file encryption is complete, both cryptographycally 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:[<&lstrcpyW>]
```

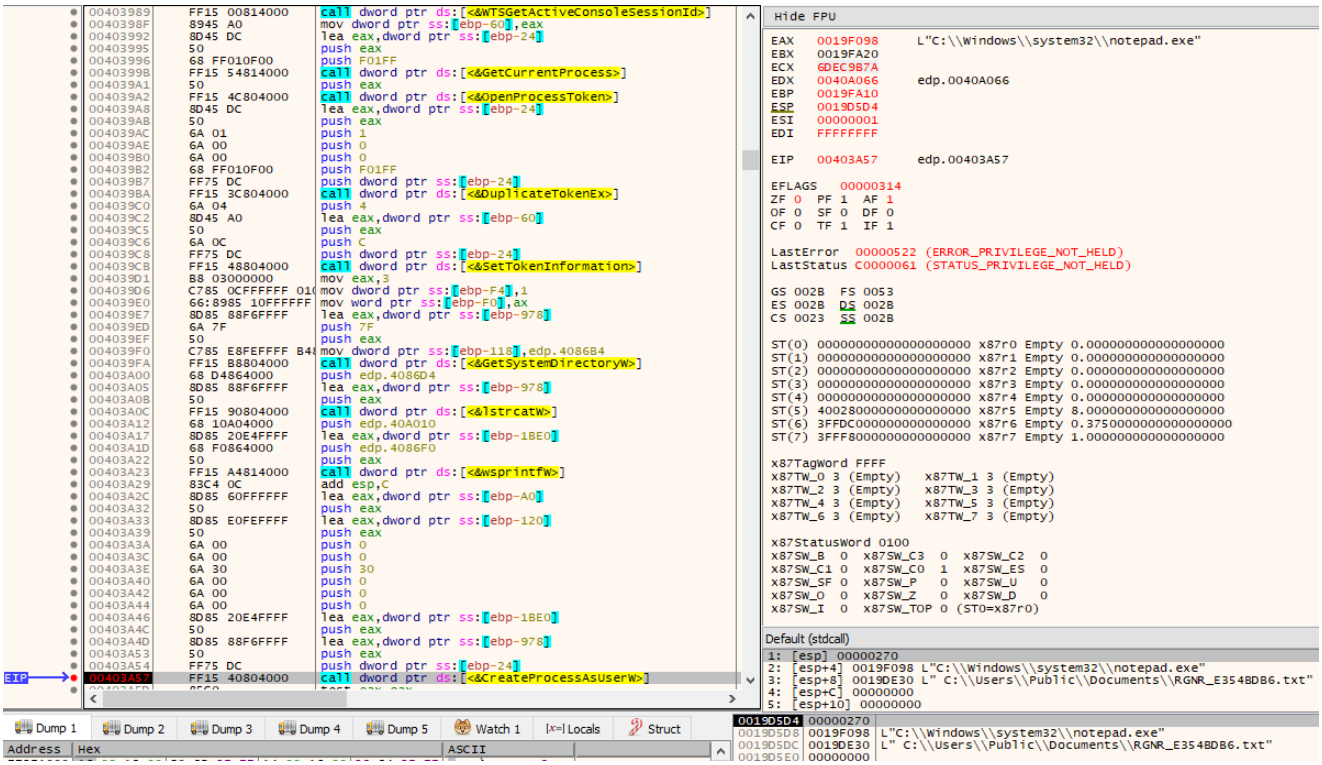
After everything is written into the new file as part of the encryption process, the file will be moved via MoveFile(), essentially adding it a new extension: .ragnar_E354BDB6



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_E354BDB6 - 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 fileservers 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) : [redacted].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 publicated 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 : [redacted].onion/client/?6bECA2b2AFFfBC1Dff0aa0EaaAd468bec0903b5e4Ea58ecde3C264bC55c7389E
c) For visit our NEWS PORTAL with your data, open this website : [redacted].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 immediatly terminates.

In theory, the perpetrators can possess file decryption capabilities, as the cryptographycally 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/