A coin miner with a “Heaven’s Gate”

blog.malwarebytes.com/threat-analysis/2018/01/a-coin-miner-with-a-heavens-gate/amp/

hasherezade

hasherezade
4 years ago

You might call the last two years the years of ransomware. Ransomware was, without a doubt, the most popular type of malware. But at the end of last year, we started observing that ransomware was losing its popularity to coin miners. It is very much possible that this trend will grow as 2018 progresses.

From the point of view of the victim, this is a huge relief, because miners are not as much of a threat as ransomware. They slow down the system, yes, but once you get rid of them you can continue using your computer as before. No data is stolen, or lost as in the case with a ransomware infection.

From the point of view of a malware researcher, miners are so far disappointing. They don’t give enough interesting material for a deeper analysis, mostly because they are based on well-known open source components with little or no obfuscation.

However, from time to time, we find coin miners incorporating interesting tricks. In one recent sample, we observed a technique called “Heaven’s Gate” that allows the malware to make injections to 64-bit processes from 32-bit loaders. This trick is not new—its introduction is dated to 2009—but it’s curious to see it implemented in this new sample captured in wild.

Those who are beginners in malware analysis can read on for a guide about what Heaven’s Gate is and how to approach analyzing it.

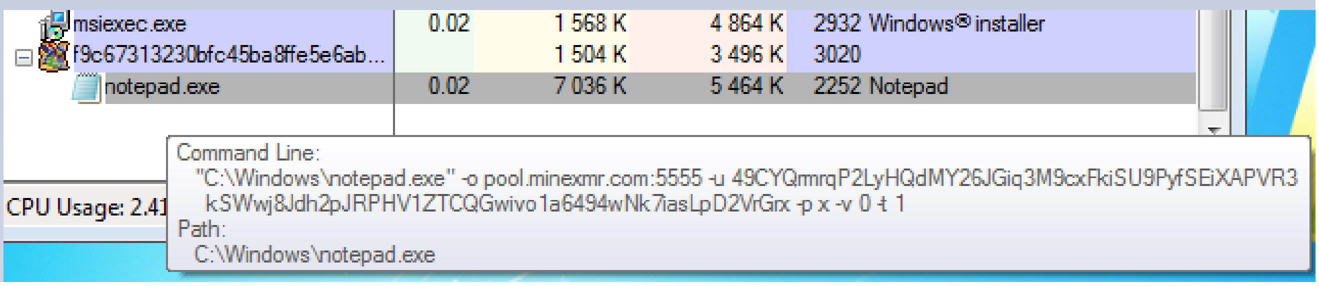
Analyzed samples

7b3491e0028d443f11989efaeb0fbec2 – dropper #1

This sample was found in the continuation of the Ngay campaign (more about it here). A background check on similar samples lead me to the article of @_qaz_qaz, who described an earlier campaign with a similar sample. However, his analysis skipped details on the Heaven’s Gate technique.

Behavioral analysis

To observe the mentioned injection, we must run the sample on a 64-bit system. We can see that it runs an instance of notepad, with parameters typical for mining cryptocurrency:

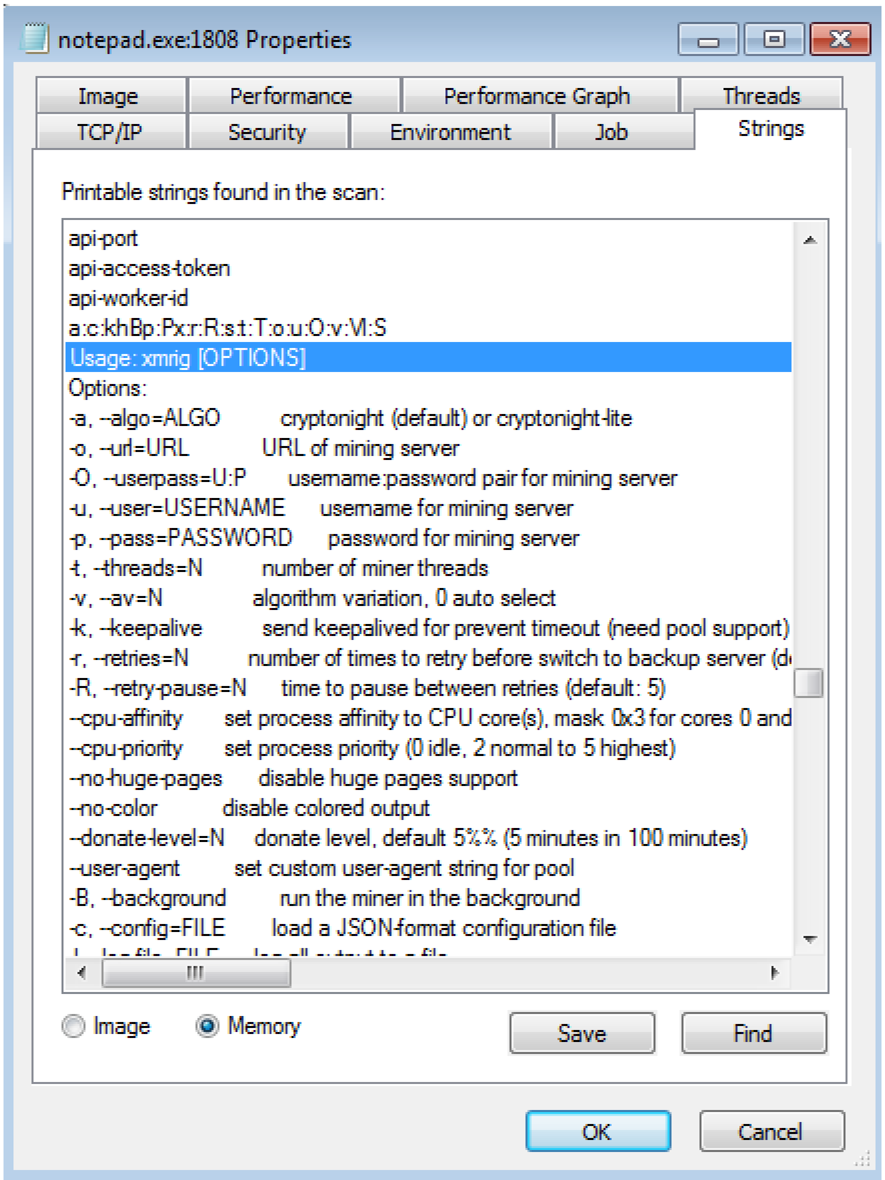


```
msiexec.exe                    0.02    1 568 K    4 864 K    2932 Windows® installer
f9c67313230bfc45ba8ffe5e6ab...         1 504 K    3 496 K    3020
    notepad.exe                0.02    7 036 K    5 464 K    2252 Notepad

CPU Usage: 2.41

Command Line:
  "C:\Windows\notepad.exe" -o pool.minexmr.com:5555 -u 49CYQmrqP2LyHQdMY26JGiq3M9cxFkiSU9PyfSEiXAPVR3
   kSWwj8Jdh2pJRPHV1ZTCQGwivo1a6494wNk7iasLpD2VrGrx -p x -v 0 -t 1
Path:
  C:\Windows\notepad.exe
```

Looking at the in-memory strings in ProcessExplorer, we can clearly see that it is not a real notepad running, but the xmrig Monero miner:



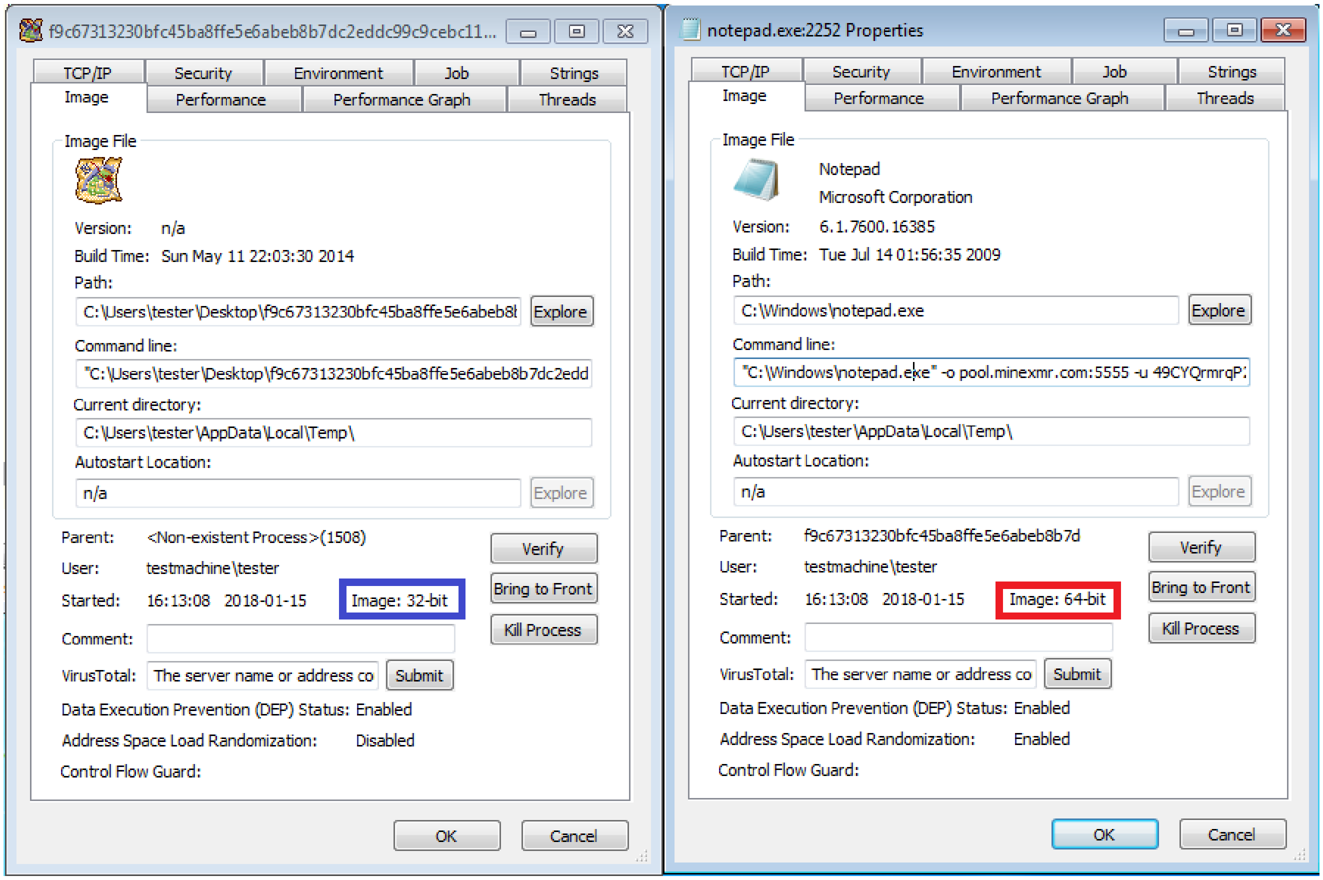
Printable strings found in the scan:

```
api-port
api-access-token
api-worker-id
a:c:khBp:Px:r:R:s:t:T:o:u:O:v:Vl:S
Usage: xmrig [OPTIONS]
Options:
-a, --algo=ALGO          cryptonight (default) or cryptonight-lite
-o, --url=URL            URL of mining server
-O, --userpass=U:P       username:password pair for mining server
-u, --user=USERNAME      username for mining server
-p, --pass=PASSWORD      password for mining server
-t, --threads=N          number of miner threads
-v, --av=N               algorithm variation, 0 auto select
-k, --keepalive          send keepalived for prevent timeout (need pool support)
-r, --retries=N          number of times to retry before switch to backup server (d
-R, --retry-pause=N      time to pause between retries (default: 5)
--cpu-affinity       set process affinity to CPU core(s), mask 0x3 for cores 0 and
--cpu-priority       set process priority (0 idle, 2 normal to 5 highest)
--no-huge-pages      disable huge pages support
--no-color           disable colored output
--donate-level=N     donate level, default 5%% (5 minutes in 100 minutes)
--user-agent         set custom user-agent string for pool
-B, --background         run the miner in the background
-c, --config=FILE        load a JSON-format configuration file
```

○ Image ● Memory

So, at this moment we’re confident that the notepad’s image has been replaced in memory, most probably by the RunPE (Process Hollowing) technique.

The main dropper is 32-bit, but it injects a payload into a 64-bit notepad:



f9c67313230bfc45ba8ffe5e6abeb8b7dc2eddc99c9cebc11...

Image File

Version: n/a
Build Time: Sun May 11 22:03:30 2014
Path:
C:\Users\tester\Desktop\f9c67313230bfc45ba8ffe5e6abeb8l
Command line:
"C:\Users\tester\Desktop\f9c67313230bfc45ba8ffe5e6abeb8b7dc2edd
Current directory:
C:\Users\tester\AppData\Local\Temp\
Autostart Location:
n/a
Parent: <Non-existent Process>(1508)
User: testmachine\tester
Started: 16:13:08 2018-01-15
Image: 32-bit
Comment:
VirusTotal: The server name or address co
Data Execution Prevention (DEP) Status: Enabled
Address Space Load Randomization: Disabled
Control Flow Guard:

notepad.exe:2252 Properties

Image File

Notepad
Microsoft Corporation
Version: 6.1.7600.16385
Build Time: Tue Jul 14 01:56:35 2009
Path:
C:\Windows\notepad.exe
Command line:
"C:\Windows\notepad.exe" -o pool.minexmr.com:5555 -u 49CYQrmrqP:
Current directory:
C:\Users\tester\AppData\Local\Temp\
Autostart Location:
n/a
Parent: f9c67313230bfc45ba8ffe5e6abeb8b7d
User: testmachine\tester
Started: 16:13:08 2018-01-15
Image: 64-bit
Comment:
VirusTotal: The server name or address co
Data Execution Prevention (DEP) Status: Enabled
Address Space Load Randomization: Enabled
Control Flow Guard:

The fun part is that this type of injection is not supported by the official Windows API. We can read/write the memory of 32-bit processes from a 64-bit application (using Wow64 API), but not the other way around.

There are, however, some unofficial solutions to this, such as the technique called “Heaven’s Gate.”

Heaven’s Gate overview

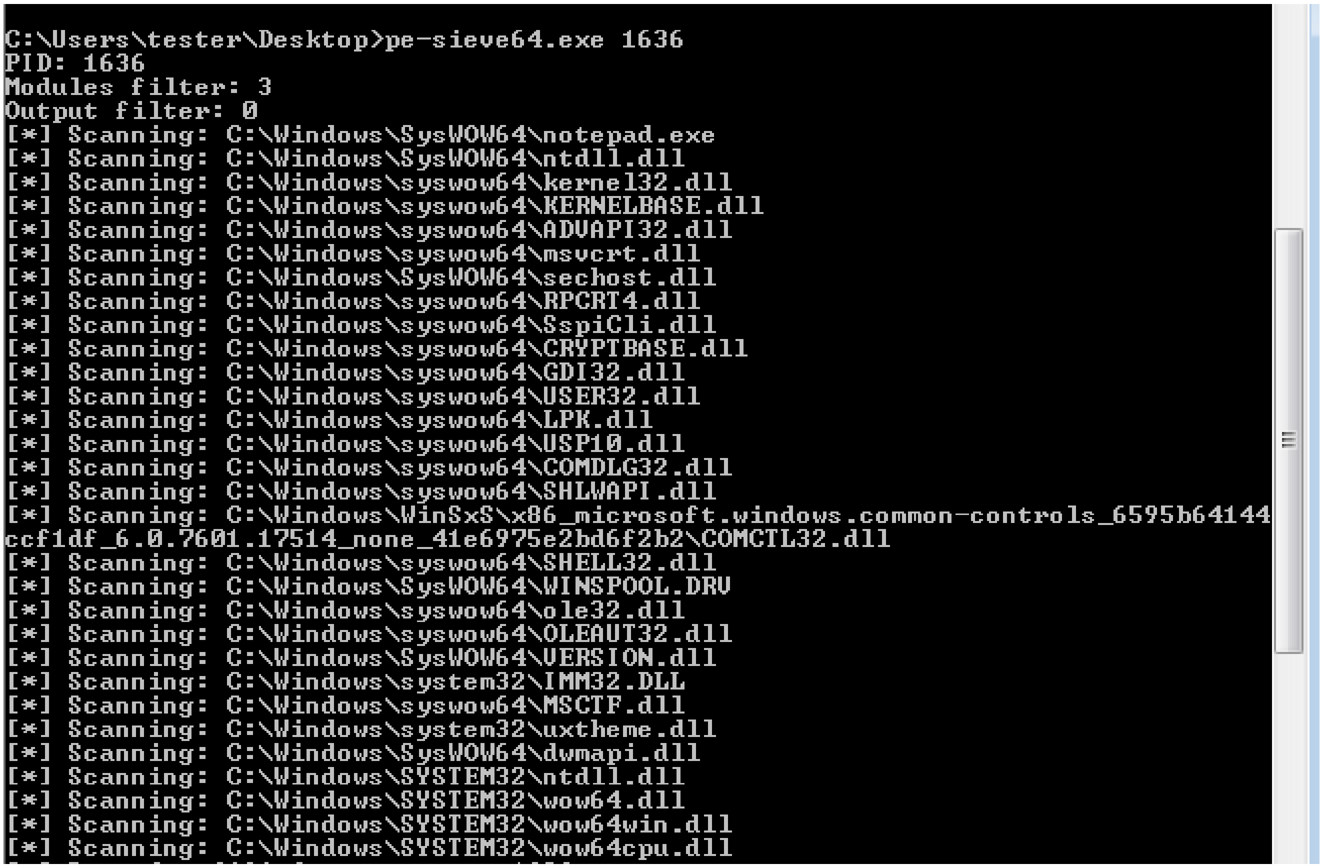
The Heaven’s Gate technique was first described in 2009, by a hacker nicknamed Roy G. Biv. Later, many adaptations were created, such as a library Wow64ext or, basing in it, W64oWoW64. In the blog post from 2015, Alex Ionescu described mitigations against this technique.

But let’s have a look at how it works.

Running 32-bit processes on 64-bit Windows

Every 32-bit process that runs on a 64-bit version of Windows runs in a special subsystem called WoW64 that emulates the 32-bit environment. We can explain it as a 32-bit sandbox that is created inside a 64-bit process. So, first the 64-bit environment for the process is created. Then, inside it, the 32-bit environment is created. The application is executed in this 32-bit environment and it has no access to the 64-bit part.

If we scan the 32-bit process from outside, via the 64-bit scanner, we can see that it has inside both 32 and 64 DLLs. Most importantly, it has two versions of NTDLL: 32-bit (loaded from a directory SysWow64) and 64-bit (loaded from a directory System32):



```
C:\Users\tester\Desktop>pe-sieve64.exe 1636
PID: 1636
Modules filter: 3
Output filter: 0
[*] Scanning: C:\Windows\SysWOW64\notepad.exe
[*] Scanning: C:\Windows\SysWOW64\ntdll.dll
[*] Scanning: C:\Windows\syswow64\kernel32.dll
[*] Scanning: C:\Windows\syswow64\KERNELBASE.dll
[*] Scanning: C:\Windows\syswow64\ADVAPI32.dll
[*] Scanning: C:\Windows\syswow64\msvcrt.dll
[*] Scanning: C:\Windows\SysWOW64\sechost.dll
[*] Scanning: C:\Windows\syswow64\RPCRT4.dll
[*] Scanning: C:\Windows\syswow64\SspiCli.dll
[*] Scanning: C:\Windows\syswow64\CRYPTBASE.dll
[*] Scanning: C:\Windows\syswow64\GDI32.dll
[*] Scanning: C:\Windows\syswow64\USER32.dll
[*] Scanning: C:\Windows\syswow64\LPK.dll
[*] Scanning: C:\Windows\syswow64\USP10.dll
[*] Scanning: C:\Windows\syswow64\COMDLG32.dll
[*] Scanning: C:\Windows\syswow64\SHLWAPI.dll
[*] Scanning: C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144
ccf1df_6.0.7601.17514_none_41e6975e2bd6f2b2\COMCTL32.dll
[*] Scanning: C:\Windows\syswow64\SHELL32.dll
[*] Scanning: C:\Windows\SysWOW64\WINSPOOL.DRV
[*] Scanning: C:\Windows\syswow64\ole32.dll
[*] Scanning: C:\Windows\syswow64\OLEAUT32.dll
[*] Scanning: C:\Windows\SysWOW64\VERSION.dll
[*] Scanning: C:\Windows\system32\IMM32.DLL
[*] Scanning: C:\Windows\syswow64\MSCTF.dll
[*] Scanning: C:\Windows\system32\uxtheme.dll
[*] Scanning: C:\Windows\SysWOW64\dwmapi.dll
[*] Scanning: C:\Windows\SYSTEM32\ntdll.dll
[*] Scanning: C:\Windows\SYSTEM32\wow64.dll
[*] Scanning: C:\Windows\SYSTEM32\wow64win.dll
[*] Scanning: C:\Windows\SYSTEM32\wow64cpu.dll
```

However, the 32-bit process itself can’t see the 64-bit part and is limited to using the 32-bit DLLs. To make an injection to a 64-bit process, we’d need to use the 64-bit versions of appropriate functions.

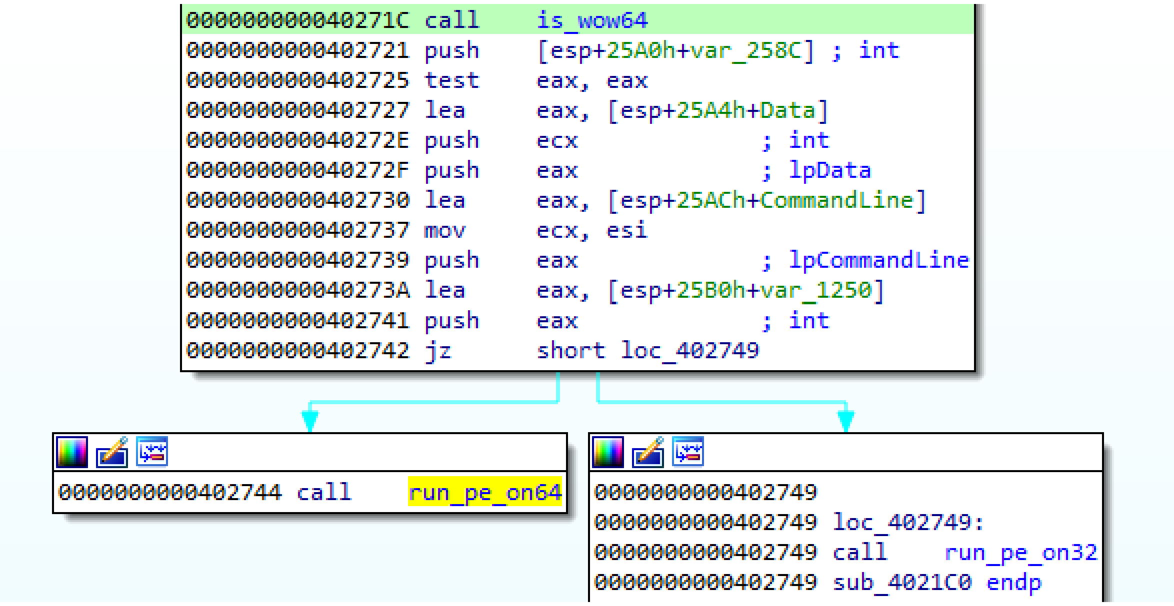
Code segments

In order to access the forbidden part of the environment, we need to understand how the isolation is made. It turns out that it’s quite simple. The 32- and 64-bit code execution is accessible via a different address of the code segment: 32-bit is 0x23 and 64-bit is 0x33.

If we call an address in a typical way, the mode that is used to interpret it is the one set by default. However, we can explicitly request to change it using assembler instructions.

Inside the miner: the Heaven’s Gate implementation

I will not do a full analysis of this miner because it has already been described here. Let’s jump directly to the place where the fun begins. The malware checks its environment, and if it finds that it’s running on a 64-bit system, it takes a different path to make an injection into a 64-bit process:

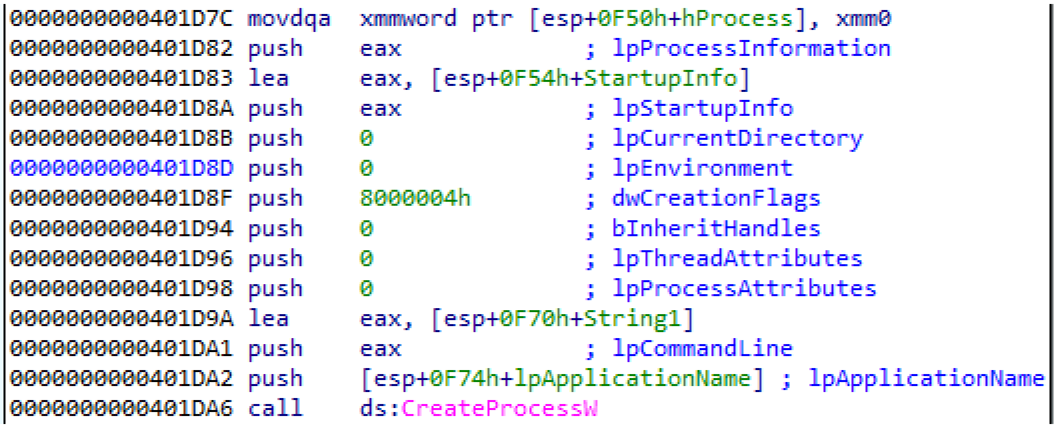


```
000000000040271C call    is_wow64
0000000000402721 push    [esp+25A0h+var_258C] ; int
0000000000402725 test    eax, eax
0000000000402727 lea     eax, [esp+25A4h+Data]
000000000040272E push    ecx              ; int
000000000040272F push    eax              ; lpData
0000000000402730 lea     eax, [esp+25ACh+CommandLine]
0000000000402737 mov     ecx, esi
0000000000402739 push    eax              ; lpCommandLine
000000000040273A lea     eax, [esp+25B0h+var_1250]
0000000000402741 push    eax              ; int
0000000000402742 jz      short loc_402749
```

```
0000000000402744 call    run_pe_on64
```

```
0000000000402749
0000000000402749 loc_402749:
0000000000402749 call    run_pe_on32
0000000000402749 sub_4021C0 endp
```

After some anti-analysis checks, it creates a new, suspended 64-bit process (in this case, it is a notepad):

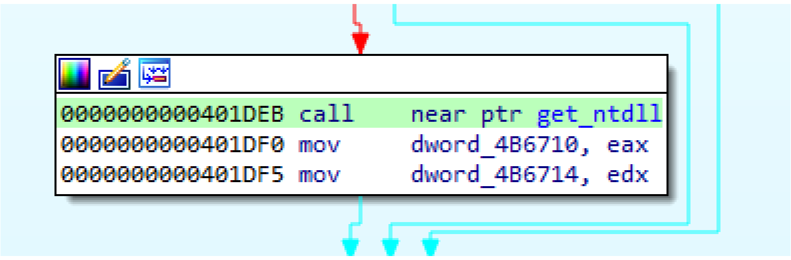


```
0000000000401D7C movdqa  xmmword ptr [esp+0F50h+hProcess], xmm0
0000000000401D82 push    eax              ; lpProcessInformation
0000000000401D83 lea     eax, [esp+0F54h+StartupInfo]
0000000000401D8A push    eax              ; lpStartupInfo
0000000000401D8B push    0                ; lpCurrentDirectory
0000000000401D8D push    0                ; lpEnvironment
0000000000401D8F push    8000004h         ; dwCreationFlags
0000000000401D94 push    0                ; bInheritHandles
0000000000401D96 push    0                ; lpThreadAttributes
0000000000401D98 push    0                ; lpProcessAttributes
0000000000401D9A lea     eax, [esp+0F70h+String1]
0000000000401DA1 push    eax              ; lpCommandLine
0000000000401DA2 push    [esp+0F74h+lpApplicationName] ; lpApplicationName
0000000000401DA6 call    ds:CreateProcessW
```

This is the target into which the malicious payload is going to be injected.

As we discussed before, in order to inject the payload into a 64-bit process, we need to use the appropriate 64-bit functions.

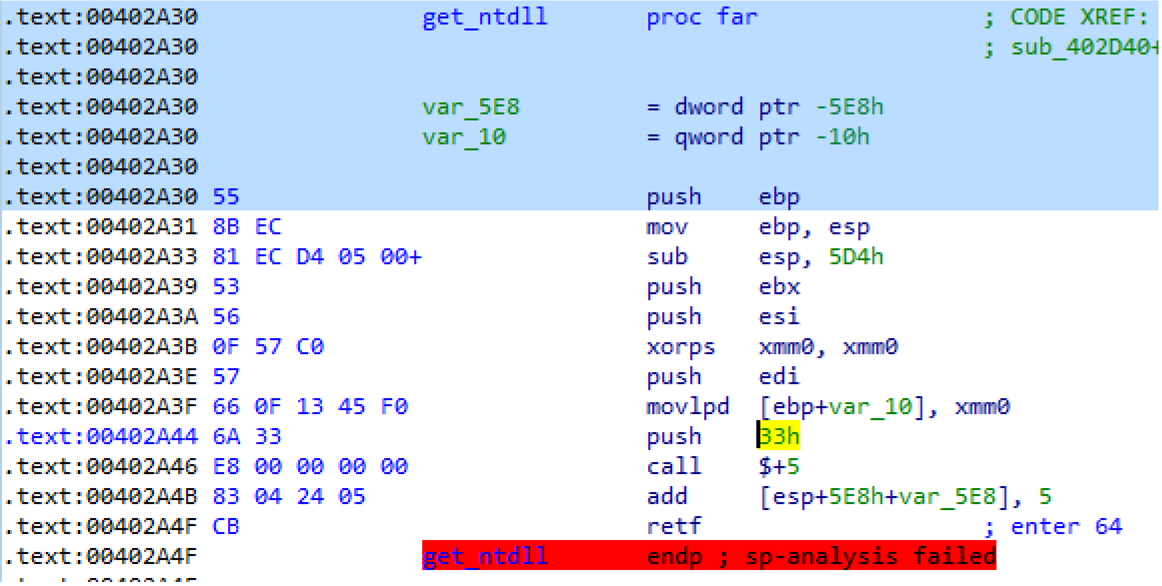
First, the loader takes a handle to a 64-bit NTDLL:



```
0000000000401DEB call    near ptr get_ntdll
0000000000401DF0 mov     dword_4B6710, eax
0000000000401DF5 mov     dword_4B6714, edx
```

What happens inside this function `get_ntdll` requires some deeper explanation. As a reference, we can also have a look at the analogical code in the ReWolf's library.

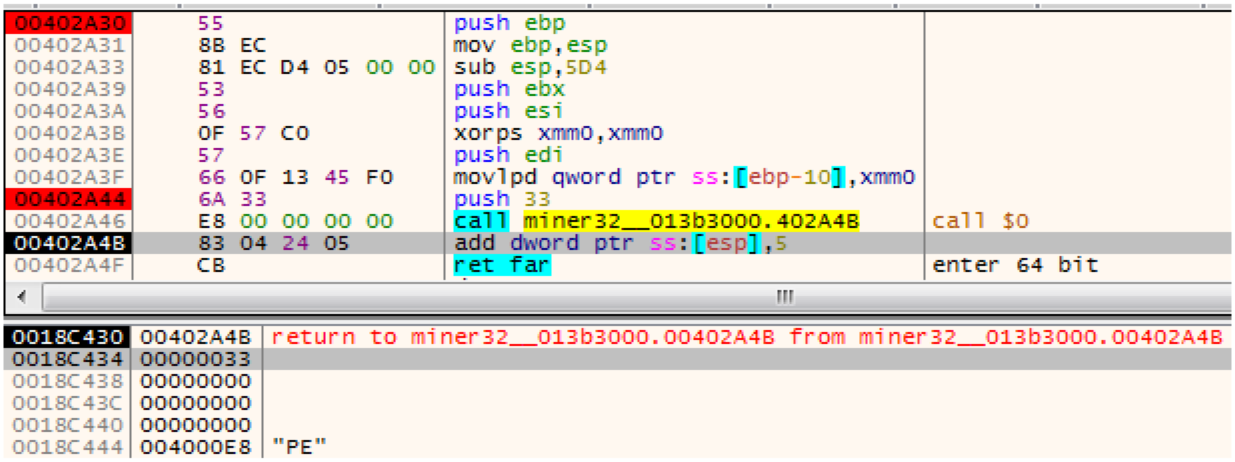
To get access to the 64-bit part of the process environment, we need to manipulate the segments selectors. Let's see how our malware enters the 64-bit mode:



```
.text:00402A30                  get_ntdll       proc far                ; CODE XREF:
.text:00402A30                                                          ; sub_402D40+
.text:00402A30
.text:00402A30                  var_5E8         = dword ptr -5E8h
.text:00402A30                  var_10          = qword ptr -10h
.text:00402A30
.text:00402A30 55                               push    ebp
.text:00402A31 8B EC                            mov     ebp, esp
.text:00402A33 81 EC D4 05 00+                  sub     esp, 5D4h
.text:00402A39 53                               push    ebx
.text:00402A3A 56                               push    esi
.text:00402A3B 0F 57 C0                         xorps   xmm0, xmm0
.text:00402A3E 57                               push    edi
.text:00402A3F 66 0F 13 45 F0                   movlpd  [ebp+var_10], xmm0
.text:00402A44 6A 33                            push    33h
.text:00402A46 E8 00 00 00 00                   call    $+5
.text:00402A4B 83 04 24 05                      add     [esp+5E8h+var_5E8], 5
.text:00402A4F CB                               retf                    ; enter 64
.text:00402A4F                  get_ntdll       endp ; sp-analysis failed
```

This code seems to be directly copied from the open source library: https://github.com/rwfpl/rewolf-wow64ext/blob/master/src/internal.h#L26

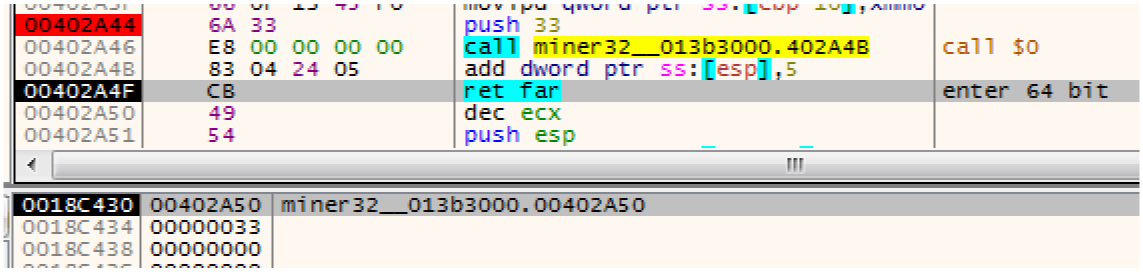
The segment selector 0x33 is pushed on the stack. Then, the malware calls the next line: (By this way, the next line's address is also pushed on the stack.)



```
00402A30   55                  push ebp
00402A31   8B EC               mov ebp,esp
00402A33   81 EC D4 05 00 00   sub esp,5D4
00402A39   53                  push ebx
00402A3A   56                  push esi
00402A3B   0F 57 C0            xorps xmm0,xmm0
00402A3E   57                  push edi
00402A3F   66 0F 13 45 F0      movlpd qword ptr ss:[ebp-10],xmm0
00402A44   6A 33               push 33
00402A46   E8 00 00 00 00      call miner32__013b3000.402A4B          call $0
00402A4B   83 04 24 05         add dword ptr ss:[esp],5
00402A4F   CB                  ret far                                enter 64 bit

0018C430 00402A4B return to miner32__013b3000.00402A4B from miner32__013b3000.00402A4B
0018C434 00000033
0018C438 00000000
0018C43C 00000000
0018C440 00000000
0018C444 004000E8 "PE"
```

An address that was pushed is fixed by adding 5 bytes and set after the retf :



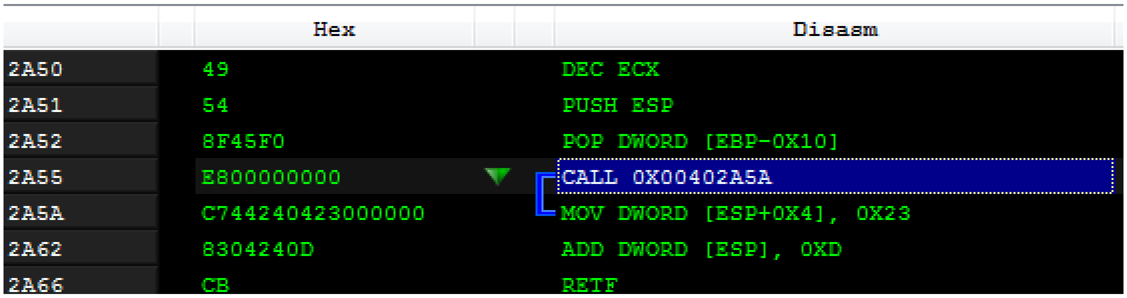
```
00402A44  6A 33           push 33
00402A46  E8 00 00 00 00  call miner32__013b3000.402A4B     call $0
00402A4B  83 04 24 05     add dword ptr ss:[esp],5
00402A4F  CB              ret far                           enter 64 bit
00402A50  49              dec ecx
00402A51  54              push esp

0018C430  00402A50  miner32__013b3000.00402A50
0018C434  00000033
0018C438  00000000
```

At the end, the instruction RETF is called. RETF is a “far return,” and in contrast to the casual RET, it allows to specify not only the address where the execution should return, but also the segment. It takes as arguments two DWORDs from the stack. So, when the RETF is hit, the actual return address is:

```
0x33:0x402A50
```

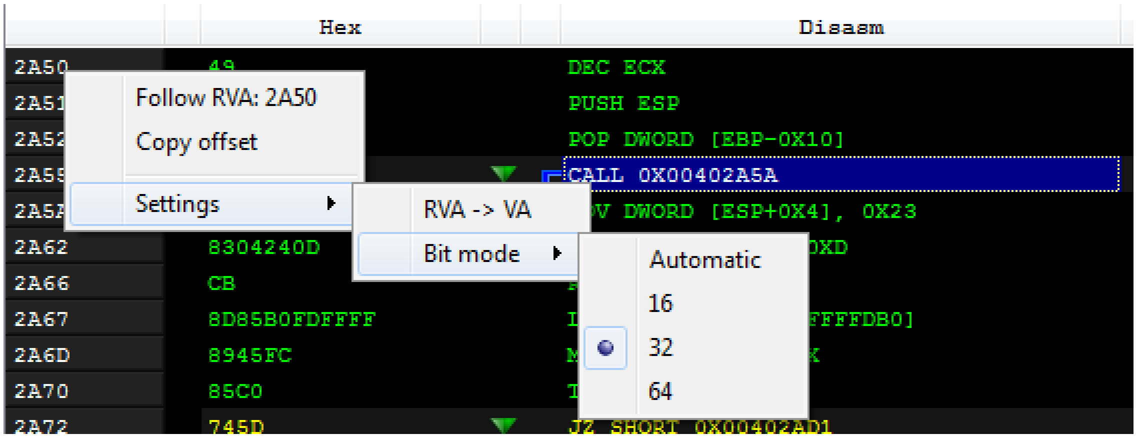
Thanks to the changed segment, the code that starts at the specified address is interpreted as 64-bit. So, the code that is visible under the debugger as 32-bit…



```
      Hex                 Disasm
2A50  49                  DEC ECX
2A51  54                  PUSH ESP
2A52  8F45F0              POP DWORD [EBP-0X10]
2A55  E800000000          CALL 0X00402A5A
2A5A  C744240423000000    MOV DWORD [ESP+0X4], 0X23
2A62  8304240D            ADD DWORD [ESP], 0XD
2A66  CB                  RETF
```

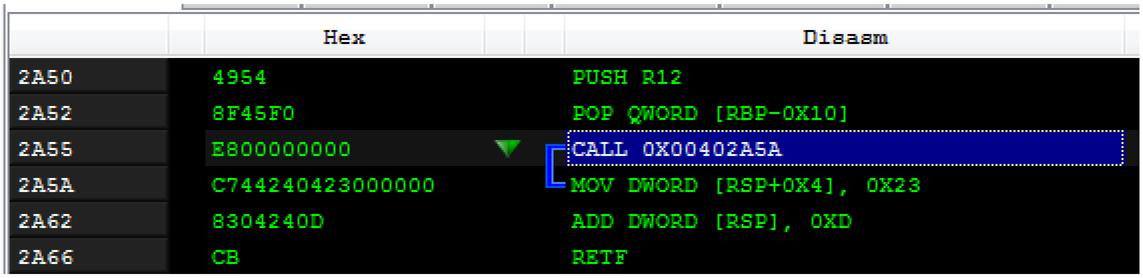
…is, in reality, 64-bit.

For the fast switching of those views, I used a feature of PE-bear:



```
2A50  49                  DEC ECX
2A51                      PUSH ESP
2A52                      POP DWORD [EBP-0X10]
2A55                      CALL 0X00402A5A
2A62  8304240D
2A66  CB
2A67  8D85B0FDFFFF
2A6D  8945FC
2A70  85C0
2A72  745D                JZ SHORT 0X00402AD1
```

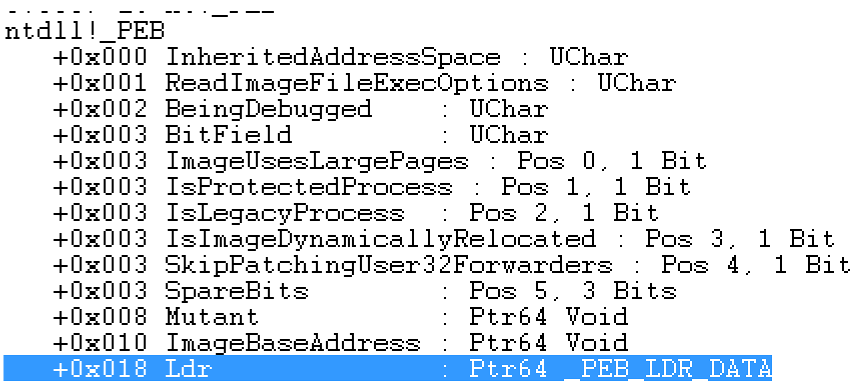
And this is how this piece of code looks, if it is interpreted as 64-bit:



```
       Hex                 Disasm
2A50   4954                PUSH R12
2A52   8F45F0              POP QWORD [RBP-0X10]
2A55   E800000000          CALL 0X00402A5A
2A5A   C744240423000000    MOV DWORD [RSP+0X4], 0X23
2A62   8304240D            ADD DWORD [RSP], 0XD
2A66   CB                  RETF
```

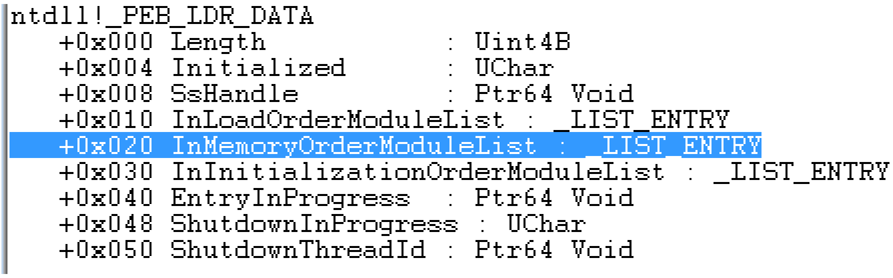
So, the code that is executed here is responsible for moving the content of the R12 register into a variable on the stack, and then switching back to the 32-bit mode. This is done for the purpose of getting 64bit Thread Environment Block (TEB), from which next we fetch the 64-bit Process Environment Block (PEB) —check the analogical code.

The 64-bit PEB is used as a starting point to search the 64-bit version of NTDLL. This part is implemented in a casual way (a "vanilla" implementation of this technique can be found here) using a pointer to the loaded libraries that is one of the fields in the PEB structure. So, from PEB we get a field called `Ldr`:



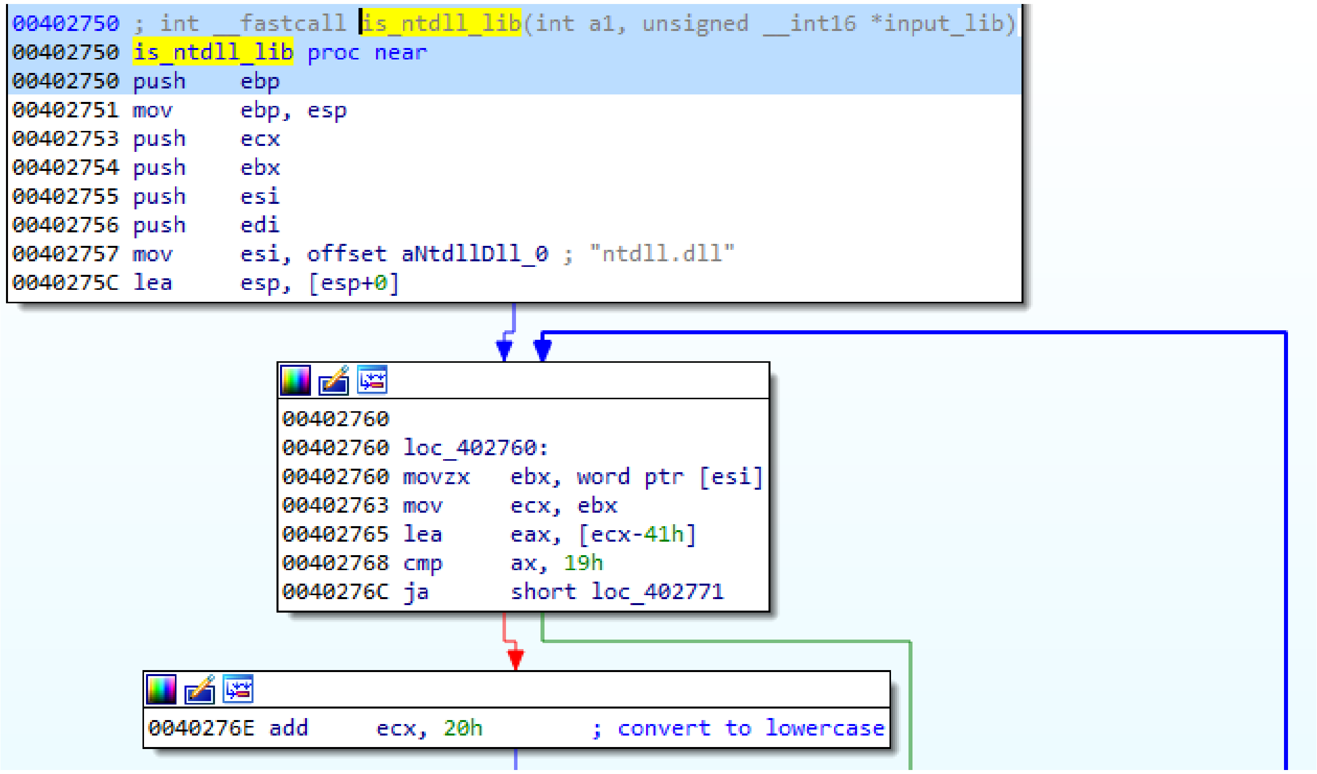
```
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsLegacyProcess  : Pos 2, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 4, 1 Bit
   +0x003 SpareBits        : Pos 5, 3 Bits
   +0x008 Mutant           : Ptr64 Void
   +0x010 ImageBaseAddress : Ptr64 Void
   +0x018 Ldr              : Ptr64 _PEB_LDR_DATA
```

Ldr is a structure of the type `_PEB_LDR_DATA`. It contains an entry called `InMemoryOrderModuleList`:



```
ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr64 Void
   +0x010 InLoadOrderModuleList : _LIST_ENTRY
   +0x020 InMemoryOrderModuleList : _LIST_ENTRY
   +0x030 InInitializationOrderModuleList : _LIST_ENTRY
   +0x040 EntryInProgress  : Ptr64 Void
   +0x048 ShutdownInProgress : UChar
   +0x050 ShutdownThreadId : Ptr64 Void
```

This list contains all the loaded DLLs that are present in the memory of the examined process. We browse through this list until we find the DLL of our interest that, in this case, is NTDLL. This is exactly what the mentioned function `get_ntdll` does. In order to find the appropriate name, it calls the following function—denoted as `is_ntdll_lib` —that checks the name of the library character-by-character and compares it with ntdll.dll. It is an equivalent of this code.

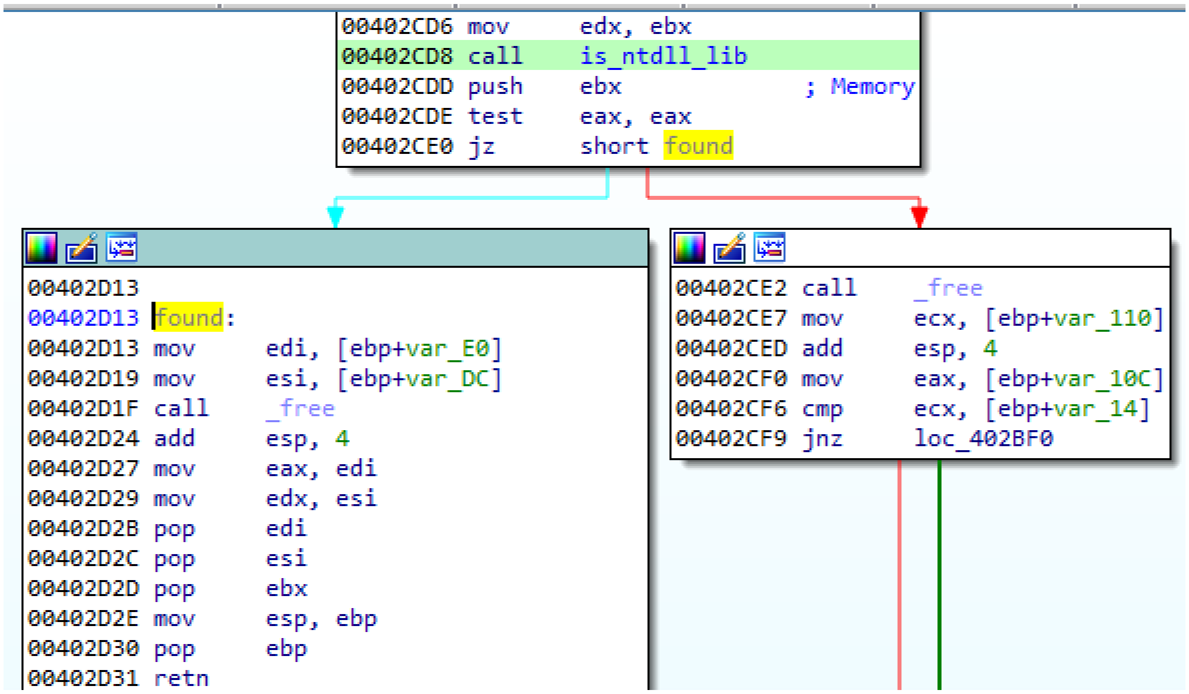


```
00402750 ; int __fastcall is_ntdll_lib(int a1, unsigned __int16 *input_lib)
00402750 is_ntdll_lib proc near
00402750 push    ebp
00402751 mov     ebp, esp
00402753 push    ecx
00402754 push    ebx
00402755 push    esi
00402756 push    edi
00402757 mov     esi, offset aNtdllDll_0 ; "ntdll.dll"
0040275C lea     esp, [esp+0]
```

```
00402760
00402760 loc_402760:
00402760 movzx   ebx, word ptr [esi]
00402763 mov     ecx, ebx
00402765 lea     eax, [ecx-41h]
00402768 cmp     ax, 19h
0040276C ja      short loc_402771
```

```
0040276E add     ecx, 20h          ; convert to lowercase
```

If the name matches, the address to the library is returned in a pair of registers:

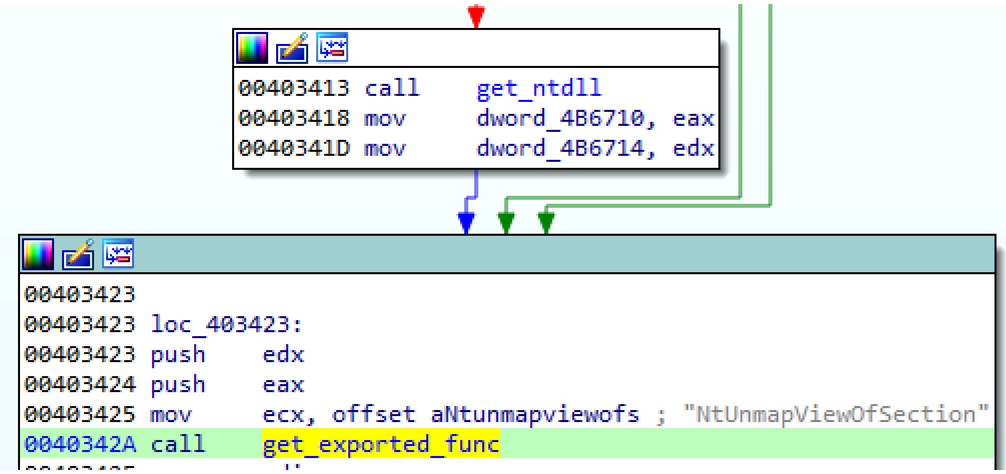


```
00402CD6 mov     edx, ebx
00402CD8 call    is_ntdll_lib
00402CDD push    ebx               ; Memory
00402CDE test    eax, eax
00402CE0 jz      short found
```

```
00402D13
00402D13 found:
00402D13 mov     edi, [ebp+var_E0]
00402D19 mov     esi, [ebp+var_DC]
00402D1F call    _free
00402D24 add     esp, 4
00402D27 mov     eax, edi
00402D29 mov     edx, esi
00402D2B pop     edi
00402D2C pop     esi
00402D2D pop     ebx
00402D2E mov     esp, ebp
00402D30 pop     ebp
00402D31 retn
```

```
00402CE2 call    _free
00402CE7 mov     ecx, [ebp+var_110]
00402CED add     esp, 4
00402CF0 mov     eax, [ebp+var_10C]
00402CF6 cmp     ecx, [ebp+var_14]
00402CF9 jnz     loc_402BF0
```

Once we found NTDLL, we just needed to fetch addresses of the appropriate functions. We did this by browsing the exports table of the DLL:



```
00403413 call    get_ntdll
00403418 mov     dword_4B6710, eax
0040341D mov     dword_4B6714, edx
```

```
00403423
00403423 loc_403423:
00403423 push    edx
00403424 push    eax
00403425 mov     ecx, offset aNtunmapviewofs ; "NtUnmapViewOfSection"
0040342A call    get_exported_func
```

The following functions are being fetched:

- NttUnmapViewOfSection
- NtGetContextThread
- NtAllocateVirtualMemory
- NtReadVirtualMemory
- NtWriteVirtualMemory
- NtSetContextThread

As we know, those functions are typical for RunPE technique. First, the `NtUnmapViewOfSection` is used to unmap the original PE file. Then, memory in the remote process is allocated, and the new PE is written. At the end, the context of the process is changed to start the execution from the injected module.

The addresses of the functions are saved and later called (similarly to this code) to manipulate the remote process.

Conclusion

So far, authors of coin miners don't show a lot of creativity. They achieve their goals by heavily relying on open-source components. The described case also shows this tendency – they made use of a ready made implementation.

The Heaven's Gate technique has been around for several years. Some malware use it for the purpose of being stealthy. But in case of this coin miner, authors probably aimed rather to maximize performance by using a payload version that best fit the target architecture.

COMMENTS