Reverse Engineering a Cobalt Strike Dropper With Binary Ninja

binary.ninja/2022/07/22/reverse-engineering-cobalt-strike.html

In this blog post, I will explain how I reverse engineered a Cobalt Strike dropper and obtained its payload. The payload is a custom executable file format based on DLL. The dropper decrypts, loads, and executes the payload. Initially, I thought this must not be a PE executable at all, but I gradually realized it was. Much of the effort was spent on fixing the file so it could be loaded by Binary Ninja for further analysis.

First Impressions

A friend of mine shared with me this sample (zip password: infected). It is an x86 PE binary that is 284kB in size. After loading it into Binary Ninja, I saw it was not packed or encrypted by any well-known packer or protector. However, there were only dozens of functions recognized, which is quite a small number relative to its size. This suggested the sample was packed by a custom packer/encryptor.

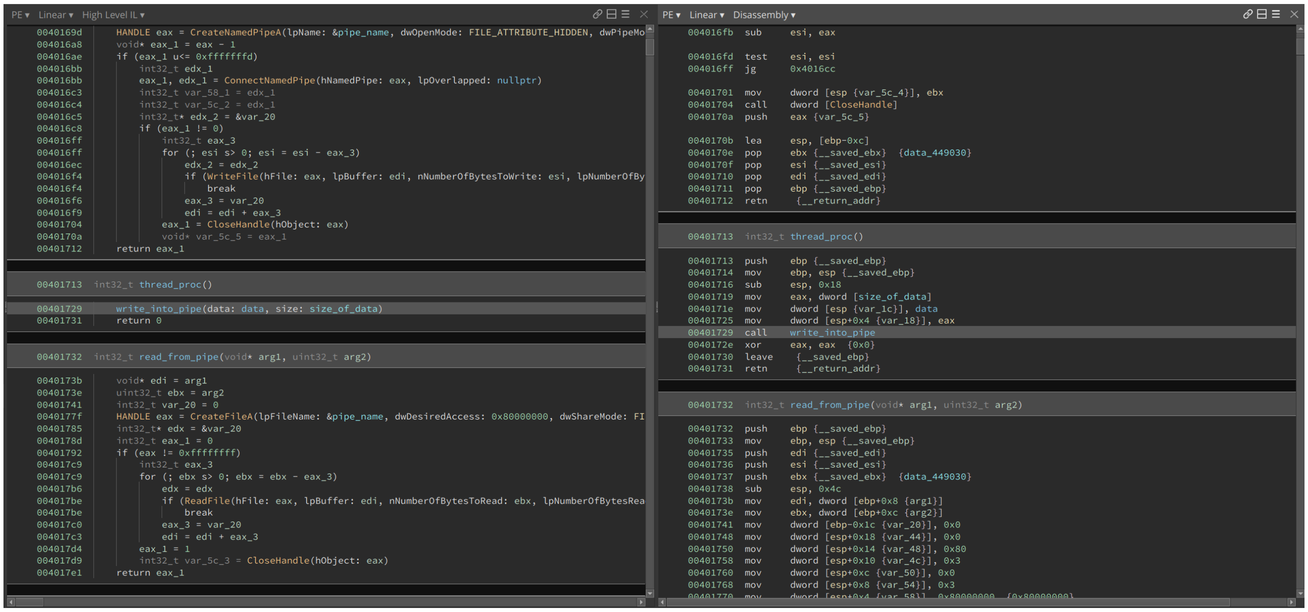
As is routine for malware analysis, I started by executing the sample in an online sandbox. In this case, I used Triage. The sample executed fine in the sandbox and was recognized as `cobaltstrike`.

Then, I uploaded the sample to UnpacMe to see if it could be unpacked automatically. UnpacMe also processed the sample and recognized it as Cobalt Strike, but the unpacked artifact did not make any sense.

At this point, I realized I wasn't going to get much further without analyzing the sample with Binary Ninja to see how it worked.

Thread and Pipe

The sample seemed to be compiler-generated and not obfuscated, so I decided to mainly analyze the sample in HLIL. Viewing code in HLIL can often speed up analysis. However, for handwritten or obfuscated code, I prefer to look at the disassembly, which offers a closer view of what is happening. Binary Ninja now supports split views, so we can conveniently view HLIL and disassembly side-by-side:

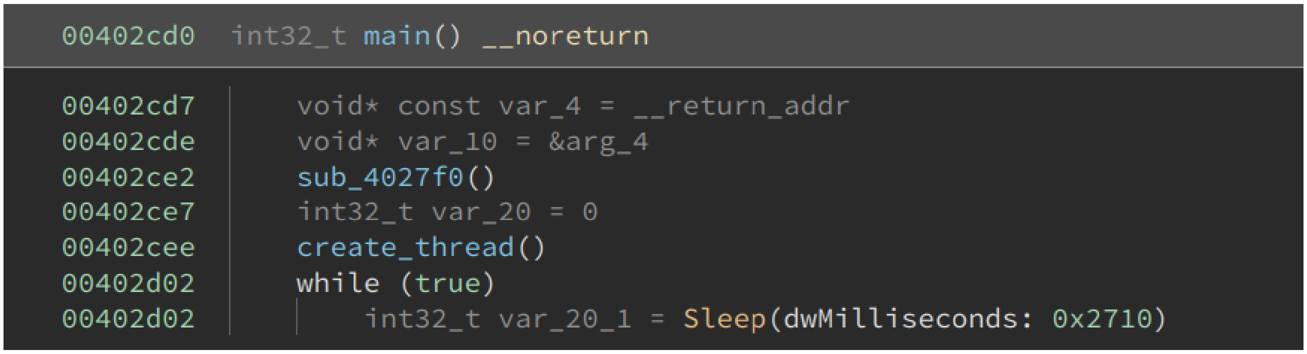


```
0040169d        HANDLE eax = CreateNamedPipeA(lpName: &pipe_name, dwOpenMode: FILE_ATTRIBUTE_HIDDEN, dwPipeMo
004016a8        void* eax_1 = eax - 1
004016ae        if (eax_1 u<= 0xfffffffd)
004016bb            int32_t edx_1
004016bb            eax_1, edx_1 = ConnectNamedPipe(hNamedPipe: eax, lpOverlapped: nullptr)
004016c3            int32_t var_58_1 = edx_1
004016c4            int32_t var_5c_2 = edx_1
004016c5            int32_t* edx_2 = &var_20
004016c8            if (eax_1 != 0)
004016ff                int32_t eax_3
004016ff                for (; esi s> 0; esi = esi - eax_3)
004016ec                    edx_2 = edx_2
004016f4                    if (WriteFile(hFile: eax, lpBuffer: edi, nNumberOfBytesToWrite: esi, lpNumberOfBy
004016f4                        break
004016f6                    eax_3 = var_20
004016f9                    edi = edi + eax_3
00401704                eax_1 = CloseHandle(hObject: eax)
0040170a                void* var_5c_5 = eax_1
00401712        return eax_1

00401713  int32_t thread_proc()

00401729        write_into_pipe(data: data, size: size_of_data)
00401731        return 0

00401732  int32_t read_from_pipe(void* arg1, uint32_t arg2)

0040173b        void* edi = arg1
0040173e        uint32_t ebx = arg2
00401741        int32_t var_20 = 0
0040177f        HANDLE eax = CreateFileA(lpFileName: &pipe_name, dwDesiredAccess: 0x80000000, dwShareMode: FI
00401785        int32_t* edx = &var_20
0040178d        int32_t eax_1 = 0
00401792        if (eax != 0xffffffff)
004017c9            int32_t eax_3
004017c9            for (; ebx s> 0; ebx = ebx - eax_3)
004017b6                edx = edx
004017be                if (ReadFile(hFile: eax, lpBuffer: edi, nNumberOfBytesToRead: ebx, lpNumberOfBytesRea
004017be                    break
004017c0                eax_3 = var_20
004017c3                edi = edi + eax_3
004017d4            eax_1 = 1
004017d9            int32_t var_5c_3 = CloseHandle(hObject: eax)
004017e1        return eax_1
```

```
004016fb  sub     esi, eax

004016fd  test    esi, esi
004016ff  jg      0x4016cc

00401701  mov     dword [esp {var_5c_4}], ebx
00401704  call    dword [CloseHandle]
0040170a  push    eax {var_5c_5}

0040170b  lea     esp, [ebp-0xc]
0040170e  pop     ebx {__saved_ebx}  {data_449030}
0040170f  pop     esi {__saved_esi}
00401710  pop     edi {__saved_edi}
00401711  pop     ebp {__saved_ebp}
00401712  retn     {__return_addr}

00401713  int32_t thread_proc()

00401713  push    ebp {__saved_ebp}
00401714  mov     ebp, esp {__saved_ebp}
00401716  sub     esp, 0x18
00401719  mov     eax, dword [size_of_data]
0040171e  mov     dword [esp {var_1c}], data
00401725  mov     dword [esp+0x4 {var_18}], eax
00401729  call    write_into_pipe
0040172e  xor     eax, eax  {0x0}
00401730  leave    {__saved_ebp}
00401731  retn     {__return_addr}

00401732  int32_t read_from_pipe(void* arg1, uint32_t arg2)

00401732  push    ebp {__saved_ebp}
00401733  mov     ebp, esp {__saved_ebp}
00401735  push    edi {__saved_edi}
00401736  push    esi {__saved_esi}
00401737  push    ebx {__saved_ebx}  {data_449030}
00401738  sub     esp, 0x4c
0040173b  mov     edi, dword [ebp+0x8 {arg1}]
0040173e  mov     ebx, dword [ebp+0xc {arg2}]
00401741  mov     dword [ebp-0x1c {var_20}], 0x0
00401748  mov     dword [esp+0x18 {var_44}], 0x0
00401750  mov     dword [esp+0x14 {var_48}], 0x80
00401758  mov     dword [esp+0x10 {var_4c}], 0x3
00401760  mov     dword [esp+0xc {var_50}], 0x0
00401768  mov     dword [esp+0x8 {var_54}], 0x3
```

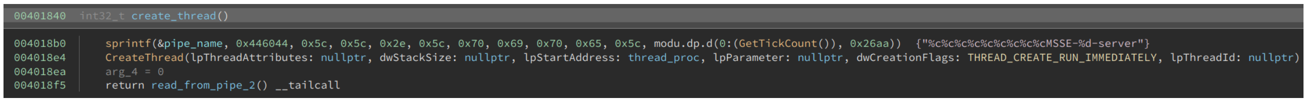
The `main` function is rather short. The first function call is part of the runtime and it is doing some initialization which we can ignore. The next function creates a new thread within it which we will analyze later. Then it enters into a loop that calls `Sleep(10000)` indefinitely.

As a note, the sample is stripped so it does not contain any function or variable names in it (except the Windows API imports). All names in the following screenshots were recovered or created during reverse engineering.



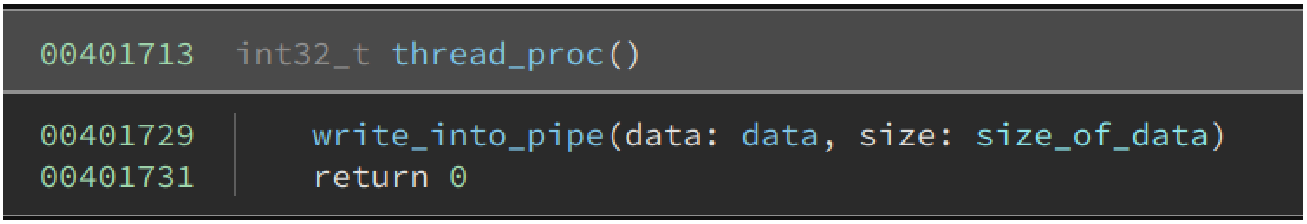
```
00402cd0  int32_t main() __noreturn

00402cd7        void* const var_4 = __return_addr
00402cde        void* var_10 = &arg_4
00402ce2        sub_4027f0()
00402ce7        int32_t var_20 = 0
00402cee        create_thread()
00402d02        while (true)
00402d02            int32_t var_20_1 = Sleep(dwMilliseconds: 0x2710)
```

The `create_thread` function is also not complex. It formats a string using values derived from `GetTickCount`, probably to make it random and avoid conflict. This string is later used as a name for a pipe. Then it creates a new thread by calling `CreateThread`.



```
00401840  int32_t create_thread()

004018b0        sprintf(&pipe_name, 0x446044, 0x5c, 0x5c, 0x2e, 0x5c, 0x70, 0x69, 0x70, 0x65, 0x5c, modu.dp.d(0:(GetTickCount()), 0x26aa))  {"%c%c%c%c%c%c%c%c%cMSSE-%d-server"}
004018e4        CreateThread(lpThreadAttributes: nullptr, dwStackSize: nullptr, lpStartAddress: thread_proc, lpParameter: nullptr, dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpThreadId: nullptr)
004018ea        arg_4 = 0
004018f5        return read_from_pipe_2() __tailcall
```

The `thread_proc` pushes two arguments onto the stack, and then calls `write_into_pipe`.



```
00401713  int32_t thread_proc()

00401729      write_into_pipe(data: data, size: size_of_data)
00401731      return 0
```

The `write_into_pipe` creates a named pipe using the randomized string, connects to it, and writes the buffer into it.

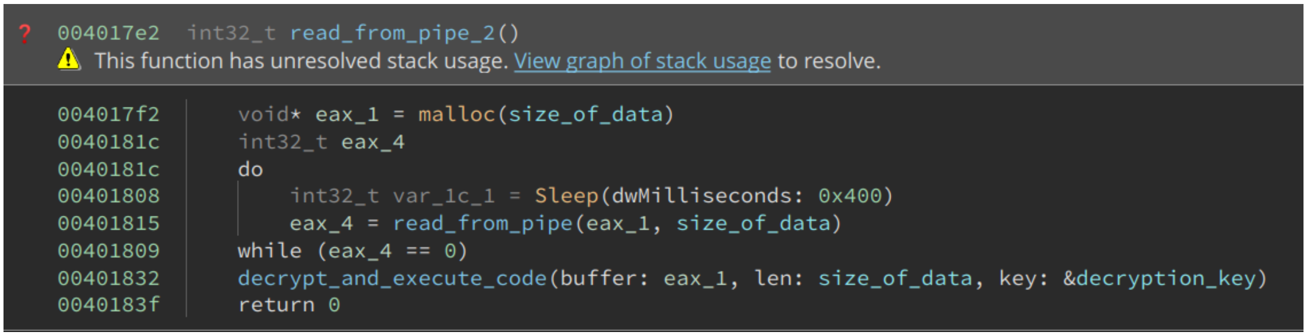
```
00401648  void* write_into_pipe(void* data, uint32_t size)

00401651      void* edi = data
00401654      uint32_t esi = size
00401657      int32_t var_20 = 0
0040169d      HANDLE eax = CreateNamedPipeA(lpName: &pipe_name, dwOpenMode: FILE_ATTRIBUTE_HIDDEN, dwPipeMode: PIPE_WAIT, nMaxInstances: 1, nOutBufferSize: 0, nInBuf
004016a8      void* eax_1 = eax - 1
004016ae      if (eax_1 u<= 0xfffffffd)
004016bb          int32_t edx_1
004016bb          eax_1, edx_1 = ConnectNamedPipe(hNamedPipe: eax, lpOverlapped: nullptr)
004016c3          int32_t var_58_1 = edx_1
004016c4          int32_t var_5c_2 = edx_1
004016c5          int32_t* edx_2 = &var_20
004016c8          if (eax_1 != 0)
004016ff              int32_t eax_3
004016ff              for (; esi s> 0; esi = esi - eax_3)
004016ec                  edx_2 = edx_2
004016f4                  if (WriteFile(hFile: eax, lpBuffer: edi, nNumberOfBytesToWrite: esi, lpNumberOfBytesWritten: edx_2, lpOverlapped: nullptr) == 0)
004016f4                      break
004016f6                  eax_3 = var_20
004016f9                  edi = edi + eax_3
00401704              eax_1 = CloseHandle(hObject: eax)
0040170a              void* var_5c_5 = eax_1
00401712      return eax_1
```

I quickly noticed `size_of_data` is huge – `0x33400` bytes. Almost the entire sample is made up of this huge buffer. This suggested the buffer was encrypted or compressed, and the dozens of functions that we see merely restore the code to its original content. Typically, at the end of it, execution will be handed to the decrypted/decompressed buffer.

At this point, we are only seeing the data being written into the named pipe. We cannot see how it is being accessed.

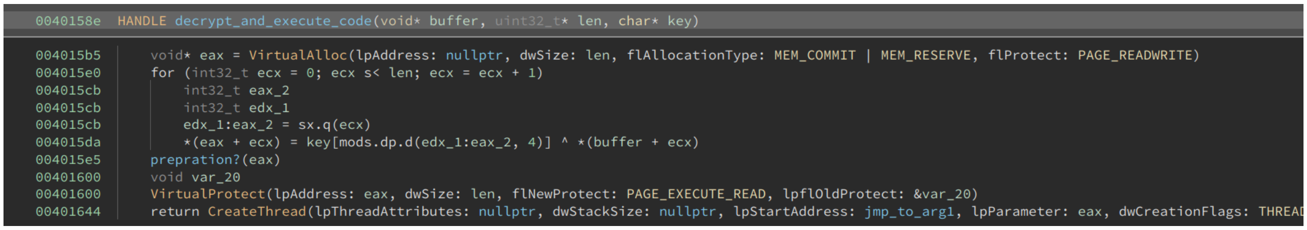
Decrypting the Buffer

After browsing the code, I realized that there was a function call at the end of `create_thead` that I had originally ignored.



```
?  004017e2  int32_t read_from_pipe_2()
⚠  This function has unresolved stack usage. View graph of stack usage to resolve.

   004017f2      void* eax_1 = malloc(size_of_data)
   0040181c      int32_t eax_4
   0040181c      do
   00401808          int32_t var_1c_1 = Sleep(dwMilliseconds: 0x400)
   00401815          eax_4 = read_from_pipe(eax_1, size_of_data)
   00401809      while (eax_4 == 0)
   00401832      decrypt_and_execute_code(buffer: eax_1, len: size_of_data, key: &decryption_key)
   0040183f      return 0
```

This function first uses `malloc` to allocate a buffer of the same size as the data written into the named pipe. It then loops and reads the content of the buffer. At the end of it, it decrypts the code and executes it.



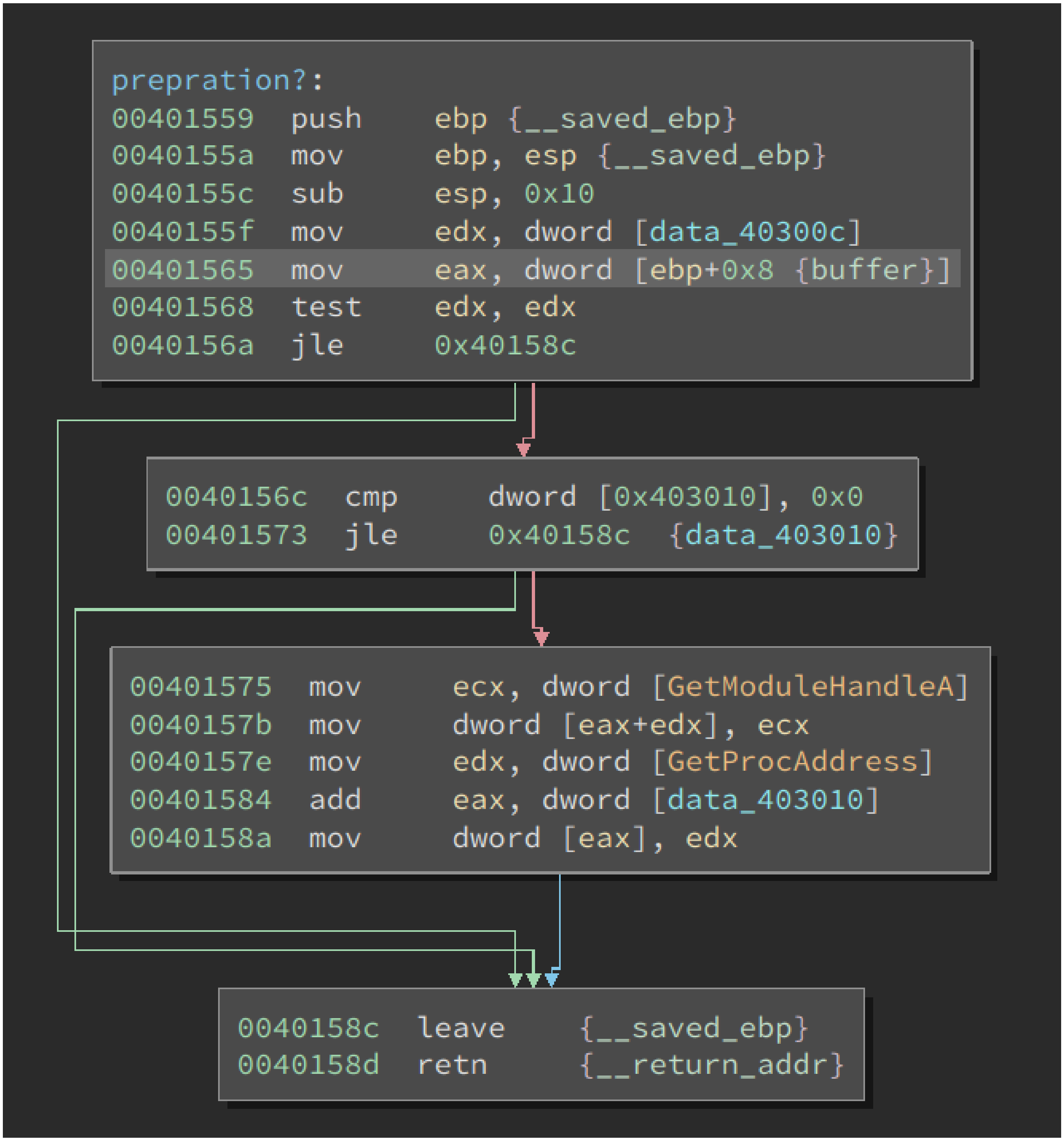
```
0040158e  HANDLE decrypt_and_execute_code(void* buffer, uint32_t* len, char* key)

004015b5      void* eax = VirtualAlloc(lpAddress: nullptr, dwSize: len, flAllocationType: MEM_COMMIT | MEM_RESERVE, flProtect: PAGE_READWRITE)
004015e0      for (int32_t ecx = 0; ecx s< len; ecx = ecx + 1)
004015cb          int32_t eax_2
004015cb          int32_t edx_1
004015cb          edx_1:eax_2 = sx.q(ecx)
004015da          *(eax + ecx) = key[mods.dp.d(edx_1:eax_2, 4)] ^ *(buffer + ecx)
004015e5      prepration?(eax)
00401600      void var_20
00401600      VirtualProtect(lpAddress: eax, dwSize: len, flNewProtect: PAGE_EXECUTE_READ, lpflOldProtect: &var_20)
00401644      return CreateThread(lpThreadAttributes: nullptr, dwStackSize: nullptr, lpStartAddress: jmp_to_arg1, lpParameter: eax, dwCreationFlags: THREAD
```

The decryption function first calls `VirtualAlloc` to allocate a buffer and sets its permission to `PAGE_READWRITE`. Then, it XORs the content with a four-byte hard-coded key. The key is `72432a9c`, in this case. Near the end of the function, it sets the permission of the buffer to `PAGE_EXECUTE_READ`. Finally, it creates another thread, which just jumps to its first argument. The address of the buffer is passed as the first argument. This starts execution from the beginning of the buffer. The code could, of course, have used the address of the buffer as the entry point of the thread. However, that might cause anti-virus software to detect it, so it used this small trick instead to disguise it.

So, in order to analyze the code of the payload, I needed to first decrypt the buffer by XORing with the four-byte key. There are two ways to do this. The first is to select the buffer, right-click, and then click `Transform -> XOR`. This is not super convenient in this case as the input buffer is huge and selecting it with a precise size is not easy. The second way is to use the Python API, which is what I did:

```
data = bv.read(0x403014, 0x33400)
xor = Transform['XOR']
output = xor.encode(data, {'key': b'\x72\x43\x2a\x9c'})
bv.write(0x403014, output)
```

Before I discuss analyzing the code in this buffer, there was a function that I initially did not quite understand. See the name I give it – `preparation`? I guessed it was doing some final preparation before executing the buffer. The HLIL for the function was also not very easy to read. However, after switching to disassembly and reading the instructions one by one, there came an “A-ha!” moment.



```
prepration?:
00401559  push    ebp {__saved_ebp}
0040155a  mov     ebp, esp {__saved_ebp}
0040155c  sub     esp, 0x10
0040155f  mov     edx, dword [data_40300c]
00401565  mov     eax, dword [ebp+0x8 {buffer}]
00401568  test    edx, edx
0040156a  jle     0x40158c
```

```
0040156c  cmp     dword [0x403010], 0x0
00401573  jle     0x40158c  {data_403010}
```

```
00401575  mov     ecx, dword [GetModuleHandleA]
0040157b  mov     dword [eax+edx], ecx
0040157e  mov     edx, dword [GetProcAddress]
00401584  add     eax, dword [data_403010]
0040158a  mov     dword [eax], edx
```

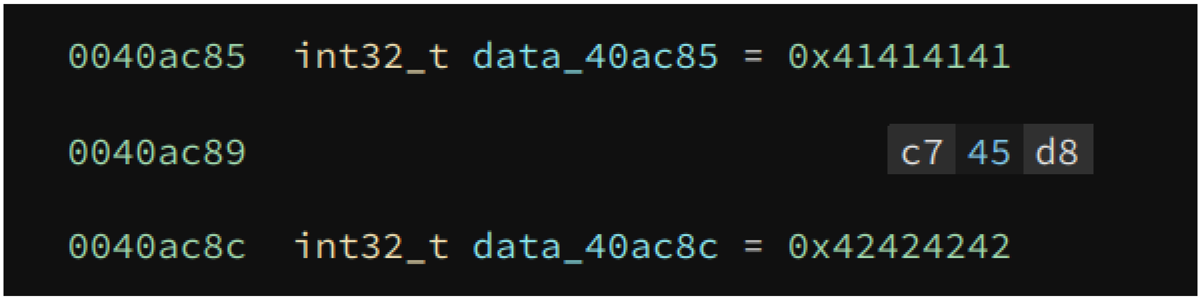
```
0040158c  leave   {__saved_ebp}
0040158d  retn    {__return_addr}
```

This function first tests whether two signed DWORDs are positive. If both of them are larger than 0, they are treated as offsets into the buffer. The code takes the address of functions `GetModuleHandleA` and `GetProcessAddress` and writes their addresses at the given offsets. In other words, it does the following:

```
*(uint32_t)(buffer + 0x7c71) = GetModuleHandleA;
*(uint32_t)(buffer + 0x7c78) = GetProcessAddress;
```

Why would the code write the address of these two functions into the middle of the buffer? Well, it is passing the function pointer into the code so that it can be used by it. This is a clever trick because the author does not have to use other (more complex) techniques to obtain these values while maintaining a low footprint in AV's eye.

Viewing the original content at those offsets confirms my guess:

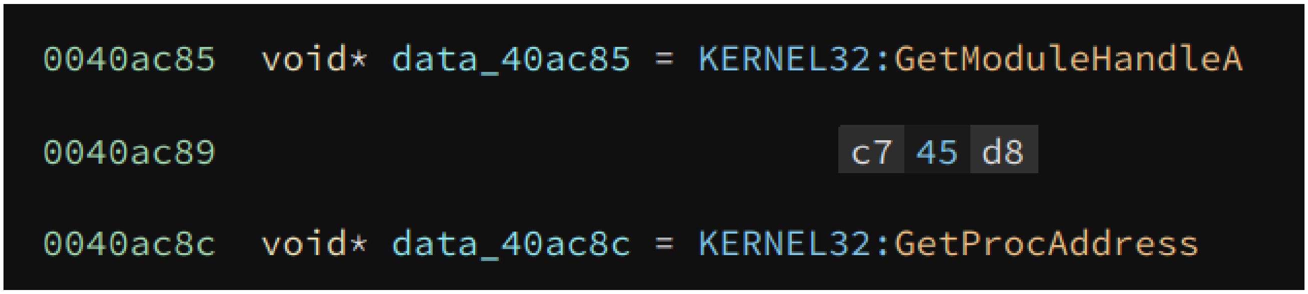


```
0040ac85  int32_t data_40ac85 = 0x41414141

0040ac89                                   c7 45 d8

0040ac8c  int32_t data_40ac8c = 0x42424242
```

The original value at the two offsets is `0x41414141` and `0x42424242`, which are obviously placeholder values. We can fix the values by writing the actual address of the two functions here. This can be done by hand, or using the following Python code:

```
addr = bv.get_symbols_by_name('GetModuleHandleA')[0].address
bv.write(0x403014 + 0x7c71, struct.pack('<I', addr))

addr = bv.get_symbols_by_name('GetProcAddress')[0].address
bv.write(0x403014 + 0x7c78, struct.pack('<I', addr))
```

If we redefine their types to `void*`, we can see the effect:

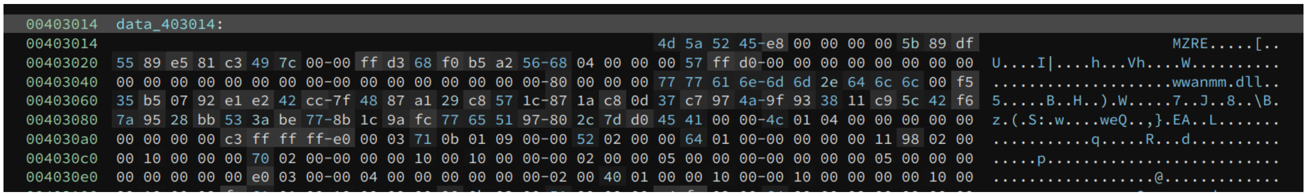


```
0040ac85  void* data_40ac85 = KERNEL32:GetModuleHandleA

0040ac89                                c7 45 d8

0040ac8c  void* data_40ac8c = KERNEL32:GetProcAddress
```

Alright, with the two values fixed, we are ready to analyze the code in the buffer.

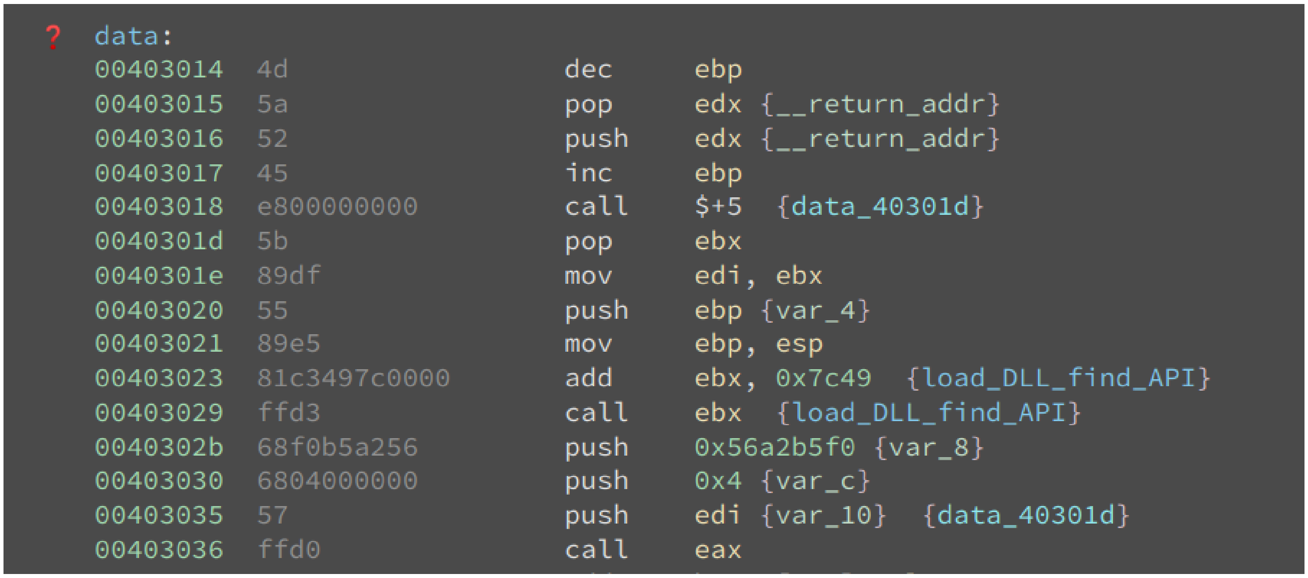
Finding Address of Windows APIs

I noticed the buffer started with `PE` as soon as it was decrypted. If this were actually a PE binary, we would simply need to dump it and load it with Binary Ninja. However, according to my analysis, this buffer is executed from the beginning. So, I quickly ruled out the possibility of this file being a PE. It must be a trick to confuse the analyst.



```
00403014  data_403014:
00403014                                                              4d 5a 52 45-e8 00 00 00 00 5b 89 df                      MZRE.....[..
00403020  55 89 e5 81 c3 49 7c 00-00 ff d3 68 f0 b5 a2 56-68 04 00 00 00 57 ff d0-00 00 00 00 00 00 00 00  U....I|....h...Vh....W..........
00403040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00-80 00 00 00 77 77 61 6e-6d 6d 2e 64 6c 6c 00 f5  ....................wwanmm.dll..
00403060  35 b5 07 92 e1 e2 42 cc-7f 48 87 a1 29 c8 57 1c-87 1a c8 0d 37 c7 97 4a-9f 93 38 11 c9 5c 42 f6  5.....B..H..).W.....7..J..8..\B.
00403080  7a 95 28 bb 53 3a be 77-8b 1c 9a fc 77 65 51 97-80 2c 7d d0 45 41 00 00-4c 01 04 00 00 00 00 00  z.(.S:.w....weQ..,}.EA..L.......
004030a0  00 00 00 00 c3 ff ff ff-e0 00 03 71 0b 01 09 00-00 52 02 00 00 64 01 00-00 00 00 00 11 98 02 00  ...........q.....R...d..........
004030c0  00 10 00 00 00 70 02 00-00 00 00 10 00 10 00 00-00 02 00 00 05 00 00 00-00 00 00 00 05 00 00 00  .....p..........................
004030e0  00 00 00 00 00 e0 03 00-00 04 00 00 00 00 00 00-02 00 40 01 00 00 10 00-00 10 00 00 00 00 10 00  ..................@.............
```

Defining a function at the entry point also produces meaningful code:



```
? data:
  00403014  4d                  dec     ebp
  00403015  5a                  pop     edx {__return_addr}
  00403016  52                  push    edx {__return_addr}
  00403017  45                  inc     ebp
  00403018  e800000000          call    $+5  {data_40301d}
  0040301d  5b                  pop     ebx
  0040301e  89df                mov     edi, ebx
  00403020  55                  push    ebp {var_4}
  00403021  89e5                mov     ebp, esp
  00403023  81c3497c0000        add     ebx, 0x7c49  {load_DLL_find_API}
  00403029  ffd3                call    ebx  {load_DLL_find_API}
  0040302b  68f0b5a256          push    0x56a2b5f0 {var_8}
  00403030  6804000000          push    0x4 {var_c}
  00403035  57                  push    edi {var_10}  {data_40301d}
  00403036  ffd0                call    eax
```

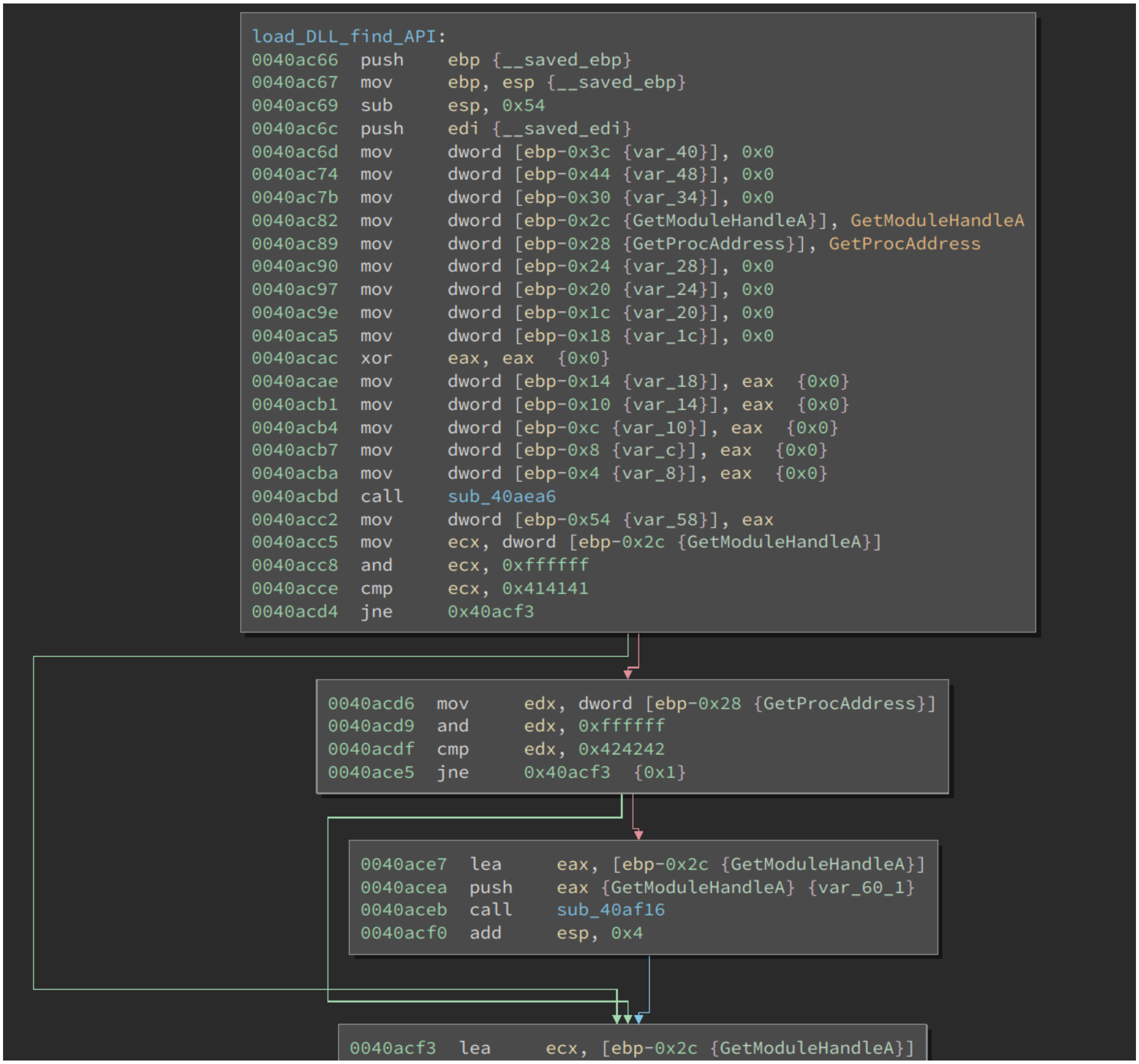
As we can see, the byte `0x4d5a (PE)` corresponds to `dec ebp; pop edx` and their effects are immediately undone by the following two instructions: `push edx; inc ebp`. Now, I am even more confident that this is not a PE, and I did not fall into the trap of the developer.

The next few instructions show a common way of getting the value of the `eip` register and then calculate an address based on it:

```
00403018  e800000000          call    $+5  {data_40301d}
0040301d  5b                  pop     ebx
......
00403023  81c3497c0000        add     ebx, 0x7c49  {load_DLL_find_API}
00403029  ffd3                call    ebx  {load_DLL_find_API}
```

Binary Ninja understands this technique, so it calculates and annotates the value of `ebx` at the call site. This is based on our dataflow analysis.

Moving on to function `load_DLL_find_API`, we can see the address of `GetModuleHandleA` and `GetProcAddress` are loaded into two stack variables, and their current values are checked against the placeholder values, i.e., `0x41414141` and `0x42424242`.



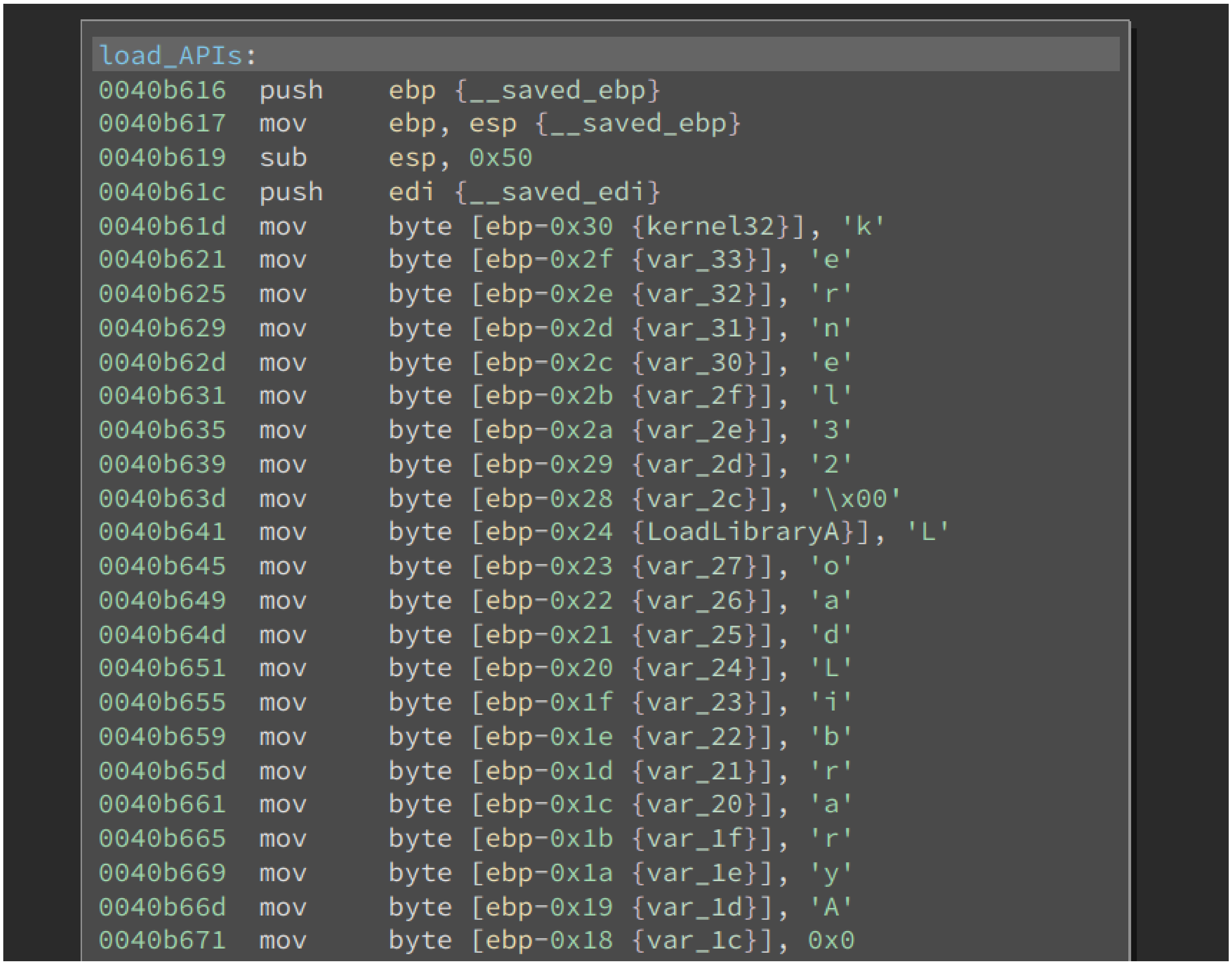
```
load_DLL_find_API:
0040ac66  push    ebp {__saved_ebp}
0040ac67  mov     ebp, esp {__saved_ebp}
0040ac69  sub     esp, 0x54
0040ac6c  push    edi {__saved_edi}
0040ac6d  mov     dword [ebp-0x3c {var_40}], 0x0
0040ac74  mov     dword [ebp-0x44 {var_48}], 0x0
0040ac7b  mov     dword [ebp-0x30 {var_34}], 0x0
0040ac82  mov     dword [ebp-0x2c {GetModuleHandleA}], GetModuleHandleA
0040ac89  mov     dword [ebp-0x28 {GetProcAddress}], GetProcAddress
0040ac90  mov     dword [ebp-0x24 {var_28}], 0x0
0040ac97  mov     dword [ebp-0x20 {var_24}], 0x0
0040ac9e  mov     dword [ebp-0x1c {var_20}], 0x0
0040aca5  mov     dword [ebp-0x18 {var_1c}], 0x0
0040acac  xor     eax, eax  {0x0}
0040acae  mov     dword [ebp-0x14 {var_18}], eax  {0x0}
0040acb1  mov     dword [ebp-0x10 {var_14}], eax  {0x0}
0040acb4  mov     dword [ebp-0xc {var_10}], eax  {0x0}
0040acb7  mov     dword [ebp-0x8 {var_c}], eax  {0x0}
0040acba  mov     dword [ebp-0x4 {var_8}], eax  {0x0}
0040acbd  call    sub_40aea6
0040acc2  mov     dword [ebp-0x54 {var_58}], eax
0040acc5  mov     ecx, dword [ebp-0x2c {GetModuleHandleA}]
0040acc8  and     ecx, 0xffffff
0040acce  cmp     ecx, 0x414141
0040acd4  jne     0x40acf3
```

```
0040acd6  mov     edx, dword [ebp-0x28 {GetProcAddress}]
0040acd9  and     edx, 0xffffff
0040acdf  cmp     edx, 0x424242
0040ace5  jne     0x40acf3  {0x1}
```

```
0040ace7  lea     eax, [ebp-0x2c {GetModuleHandleA}]
0040acea  push    eax {GetModuleHandleA} {var_60_1}
0040aceb  call    sub_40af16
0040acf0  add     esp, 0x4
```

```
0040acf3  lea     ecx, [ebp-0x2c {GetModuleHandleA}]
```

If their current values are different from the placeholder values, the following function is executed:

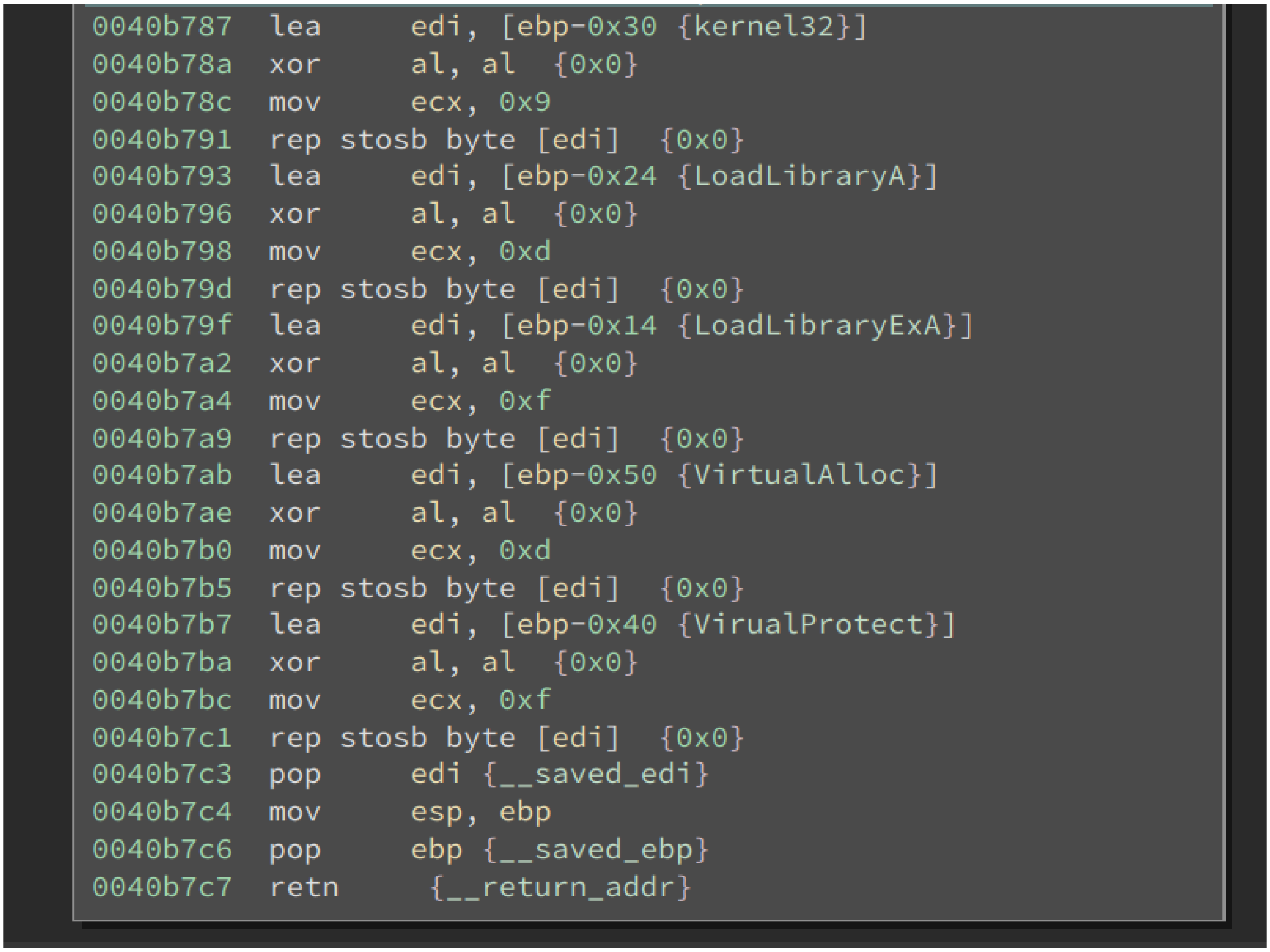


```
load_APIs:
0040b616  push    ebp {__saved_ebp}
0040b617  mov     ebp, esp {__saved_ebp}
0040b619  sub     esp, 0x50
0040b61c  push    edi {__saved_edi}
0040b61d  mov     byte [ebp-0x30 {kernel32}], 'k'
0040b621  mov     byte [ebp-0x2f {var_33}], 'e'
0040b625  mov     byte [ebp-0x2e {var_32}], 'r'
0040b629  mov     byte [ebp-0x2d {var_31}], 'n'
0040b62d  mov     byte [ebp-0x2c {var_30}], 'e'
0040b631  mov     byte [ebp-0x2b {var_2f}], 'l'
0040b635  mov     byte [ebp-0x2a {var_2e}], '3'
0040b639  mov     byte [ebp-0x29 {var_2d}], '2'
0040b63d  mov     byte [ebp-0x28 {var_2c}], '\x00'
0040b641  mov     byte [ebp-0x24 {LoadLibraryA}], 'L'
0040b645  mov     byte [ebp-0x23 {var_27}], 'o'
0040b649  mov     byte [ebp-0x22 {var_26}], 'a'
0040b64d  mov     byte [ebp-0x21 {var_25}], 'd'
0040b651  mov     byte [ebp-0x20 {var_24}], 'L'
0040b655  mov     byte [ebp-0x1f {var_23}], 'i'
0040b659  mov     byte [ebp-0x1e {var_22}], 'b'
0040b65d  mov     byte [ebp-0x1d {var_21}], 'r'
0040b661  mov     byte [ebp-0x1c {var_20}], 'a'
0040b665  mov     byte [ebp-0x1b {var_1f}], 'r'
0040b669  mov     byte [ebp-0x1a {var_1e}], 'y'
0040b66d  mov     byte [ebp-0x19 {var_1d}], 'A'
0040b671  mov     byte [ebp-0x18 {var_1c}], 0x0
```

These are all DLL and Windows API names. The function first finds `LoadLibraryA`, and then loads the needed DLLs. It also gets the addresses of the Windows API by `GetProcessAddress`. The addresses of these API calls are put into a function pointer array in the following order:

```
GetModuleHandleA
GetProcAddress
LoadLibraryA
LoadLibraryExA
VirtualAlloc
VirualProtect
```

An interesting behavior is the code zeros the strings of these API names, as seen below:



```
0040b787  lea     edi, [ebp-0x30 {kernel32}]
0040b78a  xor     al, al  {0x0}
0040b78c  mov     ecx, 0x9
0040b791  rep stosb byte [edi]  {0x0}
0040b793  lea     edi, [ebp-0x24 {LoadLibraryA}]
0040b796  xor     al, al  {0x0}
0040b798  mov     ecx, 0xd
0040b79d  rep stosb byte [edi]  {0x0}
0040b79f  lea     edi, [ebp-0x14 {LoadLibraryExA}]
0040b7a2  xor     al, al  {0x0}
0040b7a4  mov     ecx, 0xf
0040b7a9  rep stosb byte [edi]  {0x0}
0040b7ab  lea     edi, [ebp-0x50 {VirtualAlloc}]
0040b7ae  xor     al, al  {0x0}
0040b7b0  mov     ecx, 0xd
0040b7b5  rep stosb byte [edi]  {0x0}
0040b7b7  lea     edi, [ebp-0x40 {ViruralProtect}]
0040b7ba  xor     al, al  {0x0}
0040b7bc  mov     ecx, 0xf
0040b7c1  rep stosb byte [edi]  {0x0}
0040b7c3  pop     edi {__saved_edi}
0040b7c4  mov     esp, ebp
0040b7c6  pop     ebp {__saved_ebp}
0040b7c7  retn     {__return_addr}
```

This is another anti-virus evasion technique.

Is this a PE?

Since the code is quite long, I will summarize its behavior. After the above function returns, the sample does the following:

- Allocates a buffer, whose size is read from a particular offset in the buffer
- Reads section information from a section table, allocates a buffer for them, and copies the content of each section into the buffer
- Loads some DLLs specified at certain offsets in the buffer and resolve API names
- Some other things that aren’t important to our analysis

These operations very similar to loading an executable/library. Since I have ruled out this is a PE previously, I think this sample has a custom executable format. If that is the case, then I have to write a Binary View to load it. However, as I read the code more carefully, I started to realize this is a PE, though with some changes:

- The section names are XOR-ed with byte 0xc3
- The DLL names and function names are XOR-ed with 0xc3

- The `.text` section is XOR-ed with byte 0xc3

So, it turns out I have indeed been fooled by the developer: I incorrectly thought it was not a PE, whereas it turns out this *is* a modified PE format. The good news is I realized this fairly quickly and did not waste any time on writing a unnecessary loader for it.

I dumped the buffer to disk. Next, I needed to fix it so I could load it into Binary Ninja and analyze it.

The section names and `.text` section were easier to deal with. There are only a few sections, so manually XOR-ing the names was fast enough. I XOR-ed the entire `.text` section with the Transform API, as shown above.

The next problem was resolving DLL and API names. I tried to dump the file after the names were decrypted. However, it did not work because the sample copied the encrypted names into a buffer and then decrypted them. This buffer was also reused to decrypt different names. So, dumping it did not help me.

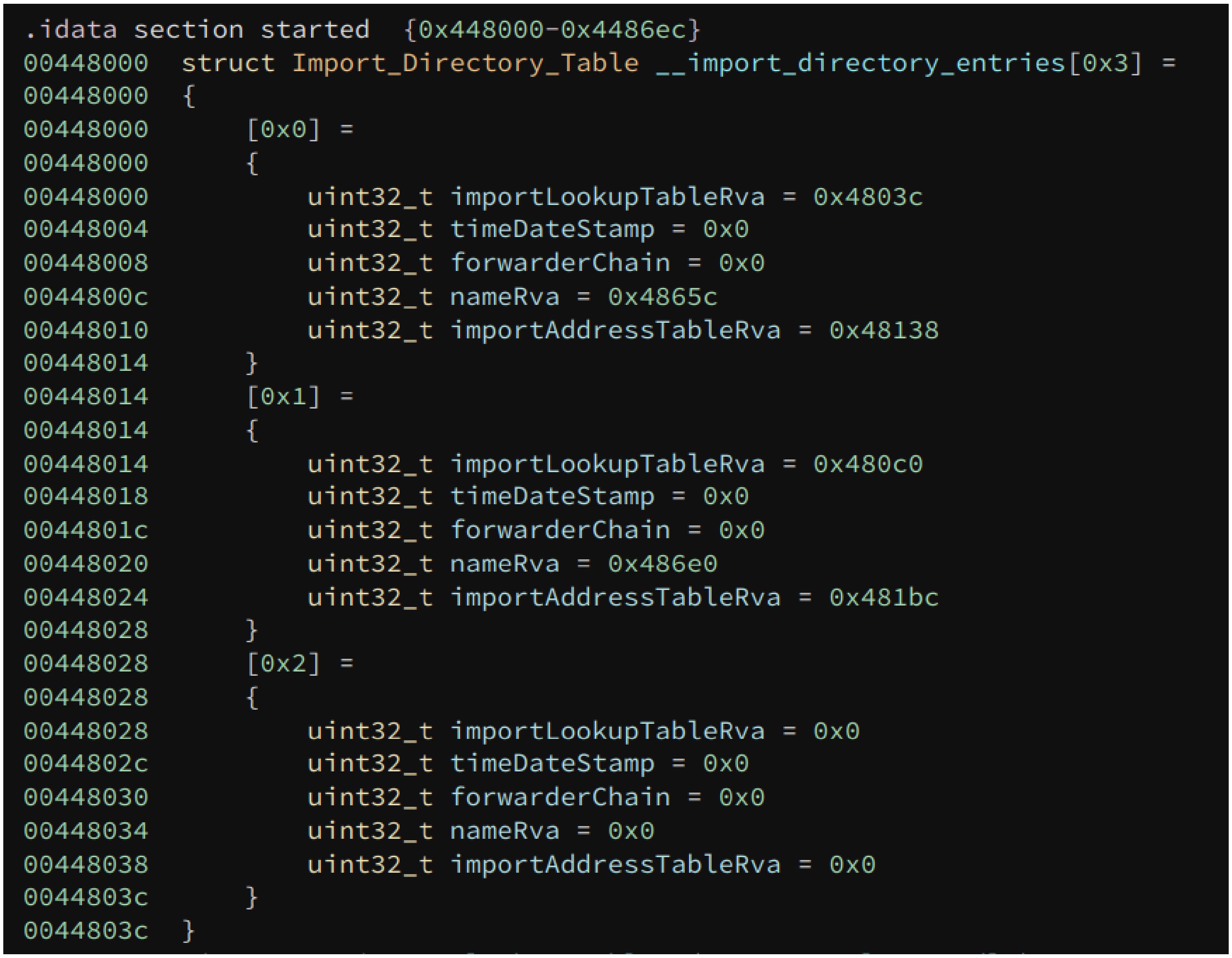
I decided to deal with this using Binary Ninja's Python API.

Fixing the Payload DLL

Let us first revisit the PE file format and see how we can find the addresses of the DLL and function names.

There are 16 `PE_Data_Directory_Entry` at the end of the `PE32_Optional_Header`. The import table is the second entry in it. The `PE_Data_Directory_Entry` contains the RVA (relative virtual address) and size of the table.

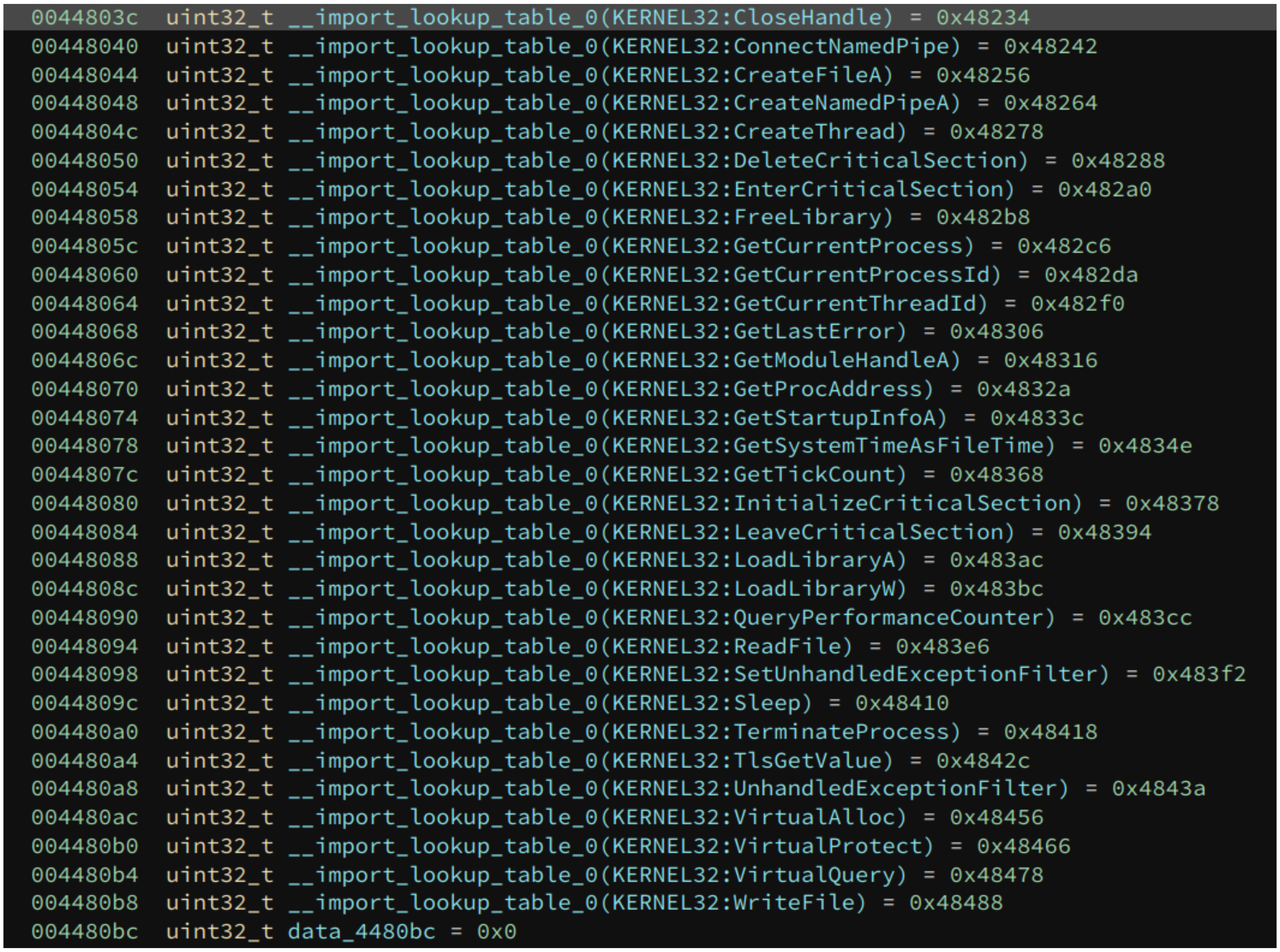
Once we calculate the VA (virtual address) of the import table from its RVA, there are multiple `Import_Directory_Table`s there. The number of entries is not specified – its end is marked by a structure whose values are NULL.



```
.idata section started  {0x448000-0x4486ec}
00448000  struct Import_Directory_Table __import_directory_entries[0x3] =
00448000  {
00448000      [0x0] =
00448000      {
00448000          uint32_t importLookupTableRva = 0x4803c
00448004          uint32_t timeDateStamp = 0x0
00448008          uint32_t forwarderChain = 0x0
0044800c          uint32_t nameRva = 0x4865c
00448010          uint32_t importAddressTableRva = 0x48138
00448014      }
00448014      [0x1] =
00448014      {
00448014          uint32_t importLookupTableRva = 0x480c0
00448018          uint32_t timeDateStamp = 0x0
0044801c          uint32_t forwarderChain = 0x0
00448020          uint32_t nameRva = 0x486e0
00448024          uint32_t importAddressTableRva = 0x481bc
00448028      }
00448028      [0x2] =
00448028      {
00448028          uint32_t importLookupTableRva = 0x0
0044802c          uint32_t timeDateStamp = 0x0
00448030          uint32_t forwarderChain = 0x0
00448034          uint32_t nameRva = 0x0
00448038          uint32_t importAddressTableRva = 0x0
0044803c      }
0044803c  }
```

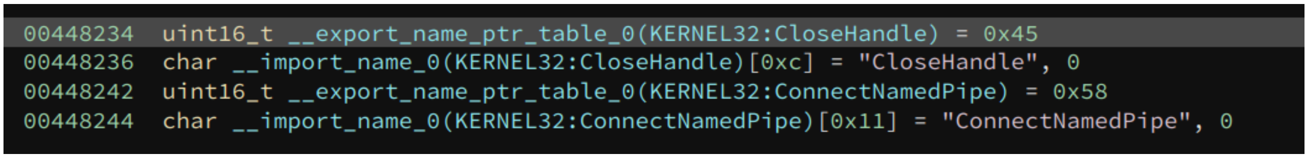
If we view the import table of the sample (the original one, not the one we have dumped), there are two entries in it. Each of these represents a DLL import and multiple function imports. The `nameRva` field is the RVA of the DLL name, so we can find the DLL names base on this.

The function names are slightly more complex. We need to follow the `importLookupTableRva` to get the `INT` (import name table).



```
0044803c  uint32_t __import_lookup_table_0(KERNEL32:CloseHandle) = 0x48234
00448040  uint32_t __import_lookup_table_0(KERNEL32:ConnectNamedPipe) = 0x48242
00448044  uint32_t __import_lookup_table_0(KERNEL32:CreateFileA) = 0x48256
00448048  uint32_t __import_lookup_table_0(KERNEL32:CreateNamedPipeA) = 0x48264
0044804c  uint32_t __import_lookup_table_0(KERNEL32:CreateThread) = 0x48278
00448050  uint32_t __import_lookup_table_0(KERNEL32:DeleteCriticalSection) = 0x48288
00448054  uint32_t __import_lookup_table_0(KERNEL32:EnterCriticalSection) = 0x482a0
00448058  uint32_t __import_lookup_table_0(KERNEL32:FreeLibrary) = 0x482b8
0044805c  uint32_t __import_lookup_table_0(KERNEL32:GetCurrentProcess) = 0x482c6
00448060  uint32_t __import_lookup_table_0(KERNEL32:GetCurrentProcessId) = 0x482da
00448064  uint32_t __import_lookup_table_0(KERNEL32:GetCurrentThreadId) = 0x482f0
00448068  uint32_t __import_lookup_table_0(KERNEL32:GetLastError) = 0x48306
0044806c  uint32_t __import_lookup_table_0(KERNEL32:GetModuleHandleA) = 0x48316
00448070  uint32_t __import_lookup_table_0(KERNEL32:GetProcAddress) = 0x4832a
00448074  uint32_t __import_lookup_table_0(KERNEL32:GetStartupInfoA) = 0x4833c
00448078  uint32_t __import_lookup_table_0(KERNEL32:GetSystemTimeAsFileTime) = 0x4834e
0044807c  uint32_t __import_lookup_table_0(KERNEL32:GetTickCount) = 0x48368
00448080  uint32_t __import_lookup_table_0(KERNEL32:InitializeCriticalSection) = 0x48378
00448084  uint32_t __import_lookup_table_0(KERNEL32:LeaveCriticalSection) = 0x48394
00448088  uint32_t __import_lookup_table_0(KERNEL32:LoadLibraryA) = 0x483ac
0044808c  uint32_t __import_lookup_table_0(KERNEL32:LoadLibraryW) = 0x483bc
00448090  uint32_t __import_lookup_table_0(KERNEL32:QueryPerformanceCounter) = 0x483cc
00448094  uint32_t __import_lookup_table_0(KERNEL32:ReadFile) = 0x483e6
00448098  uint32_t __import_lookup_table_0(KERNEL32:SetUnhandledExceptionFilter) = 0x483f2
0044809c  uint32_t __import_lookup_table_0(KERNEL32:Sleep) = 0x48410
004480a0  uint32_t __import_lookup_table_0(KERNEL32:TerminateProcess) = 0x48418
004480a4  uint32_t __import_lookup_table_0(KERNEL32:TlsGetValue) = 0x4842c
004480a8  uint32_t __import_lookup_table_0(KERNEL32:UnhandledExceptionFilter) = 0x4843a
004480ac  uint32_t __import_lookup_table_0(KERNEL32:VirtualAlloc) = 0x48456
004480b0  uint32_t __import_lookup_table_0(KERNEL32:VirtualProtect) = 0x48466
004480b4  uint32_t __import_lookup_table_0(KERNEL32:VirtualQuery) = 0x48478
004480b8  uint32_t __import_lookup_table_0(KERNEL32:WriteFile) = 0x48488
004480bc  uint32_t data_4480bc = 0x0
```

This is an array of RVAs, each describing an API function import. Again, the number of entries in this array is not specified – its end is marked by a value of NULL.



```
00448234  uint16_t __export_name_ptr_table_0(KERNEL32:CloseHandle) = 0x45
00448236  char __import_name_0(KERNEL32:CloseHandle)[0xc] = "CloseHandle", 0
00448242  uint16_t __export_name_ptr_table_0(KERNEL32:ConnectNamedPipe) = 0x58
00448244  char __import_name_0(KERNEL32:ConnectNamedPipe)[0x11] = "ConnectNamedPipe", 0
```

If we follow the VA of the first entry, we can see it comes with a two-byte ordinal of the API, followed by its name. This is how we find the names of the API.

Using BinaryReader

The entire processing script I wrote can be accessed here. Below is a walkthrough for it.

We start with the following code to find the VA of the import table:

```
from binaryninja import BinaryViewType, BinaryReader

bv = BinaryViewType.get_view_of_file('extracted_3.exe')
print(bv.start)

importTableEntry_offset = 0x100

br = BinaryReader(bv)
br.seek(bv.start + importTableEntry_offset)
import_table_va = bv.start + br.read32()
br.seek(import_table_va)
```

Two things are worth noting. First, many of the offsets in the PE file format are in RVA form, which are offsets from the start of the module. Adding `bv.start` to it converts the RVA to a VA.

Second, we are using the `BinaryReader` to read the binary. `BinaryReader` internally tracks the current offset, so it is very suitable for the case of consecutive reading. Of course, we can simply use `bv.read()` to do the job, but we would have to track the offset by ourselves, which is more effort (and more error-prone).

Strings in the PE file format are NULL-terminated. We know they are XOR-ed with a magic byte, so we need to look for it as the end of the string:

```
def read_until_byte(br, offset, byte_val):
    old_offset = br.offset
    br.seek(offset)
    result = b''
    while True:
        c = br.read(1)
        result += c
        if ord(c) == byte_val:
            break

    br.seek(old_offset)
    return result
```

Recovering the original name is very simple:

```
def xor(input, byte_val):
    result = ''
    for i in range(len(input)):
        c = chr(input[i] ^ byte_val)
        result += c

    return result
```

The main code is a loop that processes each DLL:

```
while True:
    table_rva = br.read32()
    if table_rva == 0:
        break

    br.seek(br.offset + 8)
    name_rva = br.read32()
    # print('name_rva: 0x%x' % name_rva)
    name_va = bv.start + name_rva
    name = read_until_byte(br, name_va, 0xc3)
    restored_name = xor(name, 0xc3)
    print(restored_name)
    bv.write(name_va, restored_name)

    table_va = bv.start + table_rva
    # print("table_va", hex(table_va))
    process_table(br, bv.start, table_va)

    br.seek(br.offset + 4)
```

The code to process each table (DLL) is also a loop:

```
def process_table(br, start, offset):
    old_offset = br.offset
    br.seek(offset)

    while True:
        int_rva = br.read32()
        if (int_rva == 0):
            break
        if (int_rva & 0x80000000 != 0):
            continue
        else:
            int_va = start + int_rva
            # print('int_va', hex(int_va))
            process_one_entry(br, start, int_va)

    br.seek(old_offset)
```

Note that if the INT RVA has its highest bit set, then this API is not imported by name. Instead, it is imported by ordinal. In that case, we should skip it.

Finally, we get to process an individual API name:

```
def process_one_entry(br, start, address):
    old_offset = br.offset
    br.seek(address)
    br.read16()

    # print('br.offset', hex(br.offset))
    name = read_until_byte(br, br.offset, 0xc3)
    restored_name = xor(name, 0xc3)
    print(restored_name)
    bv.write(address + 2, restored_name)

    br.seek(old_offset)
```

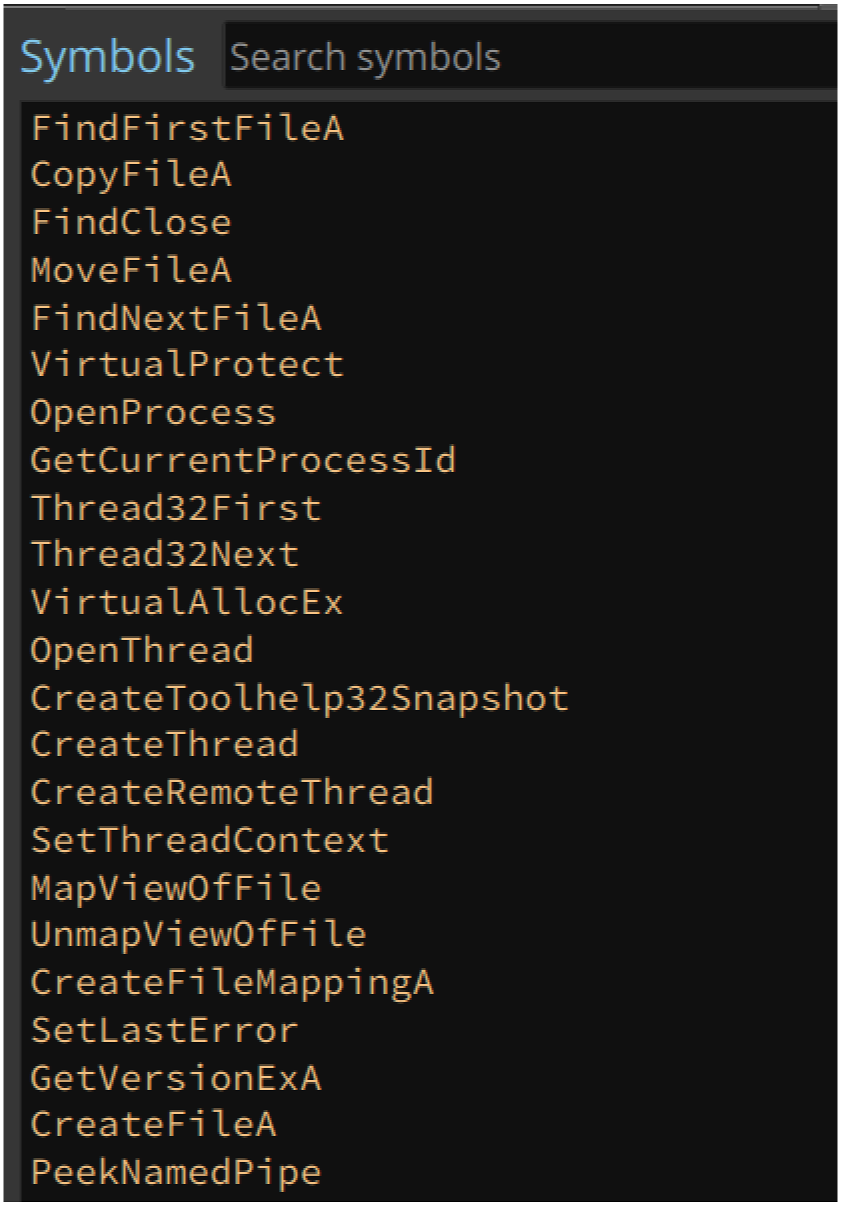
Once we are done processing, we can export the DLL to disk:

```
bv.save('extracted.dll')
```

The DLL can be downloaded from here (zip password: infected).

This sample has another trick to slow down the analyst: Its entry point offset is not read from the `PE32_Optional_Header.addressOfEntryPoint` (offset 0x28). Instead, it is read from the `PE32_Optional_Header.loaderFlags` (offset 0x70). To fix this, we simply change the value of `addressOfEntryPoint` accordingly.

Now, we can load the extracted DLL into Binary Ninja and analyze it. We can see all the Windows APIs it imports.



```
FindFirstFileA
CopyFileA
FindClose
MoveFileA
FindNextFileA
VirtualProtect
OpenProcess
GetCurrentProcessId
Thread32First
Thread32Next
VirtualAllocEx
OpenThread
CreateToolhelp32Snapshot
CreateThread
CreateRemoteThread
SetThreadContext
MapViewOfFile
UnmapViewOfFile
CreateFileMappingA
SetLastError
GetVersionExA
CreateFileA
PeekNamedPipe
```

There is a giant `switch` statement in it (with 0x65 `case` s), which handles different commands. Analyzing each of them is beyond the scope of this blog post.

```
huge_switch:
10007e9e  push    ebx {__saved_ebx}  {0x0}
10007e9f  push    esi {__saved_esi}
10007ea0  dec     eax
10007ea1  push    edi {__saved_edi}
10007ea2  mov     esi, edx
10007ea4  cmp     eax, 0x65
10007ea7  ja      0x100081f4
```

```
10007ead  jmp     dword [eax*4+0x100081fa]
```

```
1000801a  push    esi {var_10_3}
1000801b  push    ecx {var_14_3}
1000801c  call    sub_10002c4b
10008021  jmp     0x10007ee1
```

```
10008026  push    esi {var_10_3}
10008027  push    ecx {var_14_3}
10008028  call    sub_1000ac7c
1000802d  jmp     0x10007ee1
```

```
10008032  push    0x0 {var_10_7}
10008034  jmp     0x100081ea
```

```
10008039  push    esi {var_10_3}
1000803a  push    ecx {var_14_3}
1000803b  call    sub_1000728d
10008040  jmp     0x10007ee1
```

```
10008045  push    esi {var_10_3}
10008046  push    ecx {var_14_3}
10008047  call    sub_10002ce5
1000804c  jmp     0x10007ee1
```

```
10008051  push    esi {var_10_3}
10008052  push    ecx {var_14_3}
10008053  call    sub_10004023
10008058  jmp     0x10007ee1
```

```
1000805d  push    esi {var_10_3}
1000805e  push    ecx {var_14_3}
1000805f  call    sub_10003ee4
10008064  jmp     0x10007ee1
```

```
10008069  push    esi {var_10_3}
1000806a  push    ecx {var_14_3}
1000806b  call    sub_10003dbe
10008070  jmp     0x10007ee1
```

```
10008075  push    esi {var_10_3}
10008076  push    ecx {var_14_3}
10008077  call    sub_10003e95
1000807c  jmp     0x10007ee1
```

However, since we have fixed the imports, a glance can already give us a good guess at what each might be doing. For example, the following function is likely searching for certain files:

```
100086e6  BOOL sub_100086e6(int32_t arg1, struct WIN32_FIND_DATAA* arg2, int32_t arg3)

100086e9      int32_t ecx
100086e9      int32_t var_8 = ecx
100086f3      void* eax = sub_10014855(0x8000)
100086f8      int32_t var_1c = arg1
10008704      sub_100149a6(eax, 0x8000, "%s\*")
10008709      struct WIN32_FIND_DATAA* esi = arg2
10008711      FindFileHandle eax_1 = FindFirstFileA(lpFileName: eax, lpFindFileData: esi)
10008717      void* var_18_1 = eax
1000871b      BOOL eax_2 = sub_10014778()
10008725      if (eax_1 != 0xffffffff)
1000879a          BOOL eax_5
1000879a          do
1000872a              CHAR (* eax_3)[0x104] = &esi->cFileName
1000872d              if ((esi->dwFileAttributes.b & 0x10) == 0)
100087b0                  arg3(arg1, eax_3, 0)
10008731              else
10008731                  char* edi_1 = &data_1002d70c
10008736                  CHAR (* esi_1)[0x104] = eax_3
10008738                  int32_t ecx_2 = 2
10008739                  bool cond:3_1 = true
1000873b                  while (ecx_2 != 0)
1000873b                      CHAR temp0_1 = *esi_1
1000873b                      char temp1_1 = *edi_1
1000873b                      cond:3_1 = temp0_1 == temp1_1
1000873b                      esi_1 = &(*esi_1)[1]
1000873b                      edi_1 = &edi_1[1]
1000873b                      ecx_2 = ecx_2 - 1
1000873b                      if (temp0_1 != temp1_1)
1000873b                          break
```

Alright, we have successfully reverse-engineered this Cobalt Strike sample and fixed its payload DLL!