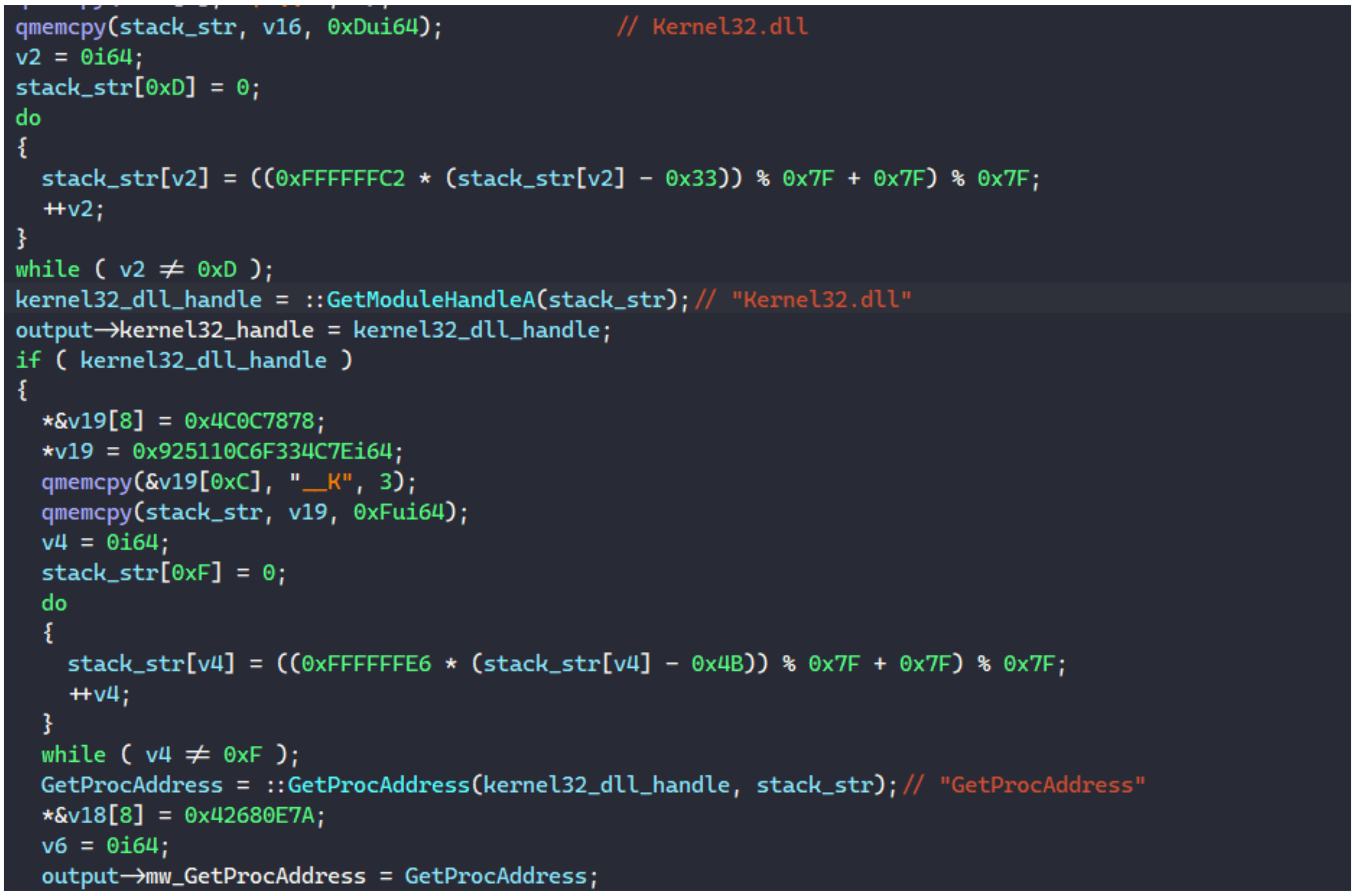


```
qmemcpy(stack_str, v16, 0xDui64);              // Kernel32.dll
v2 = 0i64;
stack_str[0xD] = 0;
do
{
  stack_str[v2] = ((0xFFFFFFC2 * (stack_str[v2] - 0x33)) % 0x7F + 0x7F) % 0x7F;
  ++v2;
}
while ( v2 ≠ 0xD );
kernel32_dll_handle = ::GetModuleHandleA(stack_str);// "Kernel32.dll"
output→kernel32_handle = kernel32_dll_handle;
if ( kernel32_dll_handle )
{
  *&v19[8] = 0x4C0C7878;
  *v19 = 0x925110C6F334C7Ei64;
  qmemcpy(&v19[0xC], "__K", 3);
  qmemcpy(stack_str, v19, 0xFui64);
  v4 = 0i64;
  stack_str[0xF] = 0;
  do
  {
    stack_str[v4] = ((0xFFFFFFE6 * (stack_str[v4] - 0x4B)) % 0x7F + 0x7F) % 0x7F;
    ++v4;
  }
  while ( v4 ≠ 0xF );
  GetProcAddress = ::GetProcAddress(kernel32_dll_handle, stack_str);// "GetProcAddress"
  *&v18[8] = 0x42680E7A;
  v6 = 0i64;
  output→mw_GetProcAddress = GetProcAddress;
```

- Chuong Dong
- 19th April 2022
- No Comments

BAZARLOADER (aka BAZARBACKDOOR) is a Windows-based loader that spreads through attachments in phishing emails. During an infection, the final loader payload typically downloads and executes a Cobalt Strike beacon to provide remote access for the threat actors, which, in a lot of cases, leads to ransomware being deployed to the victim's machine.

In this initial post, we will unpack the different stages of a BAZARLOADER infection that comes in the form of an optical disk image (ISO) file. We will also dive into the obfuscation methods used by the main BAZARLOADER payload.

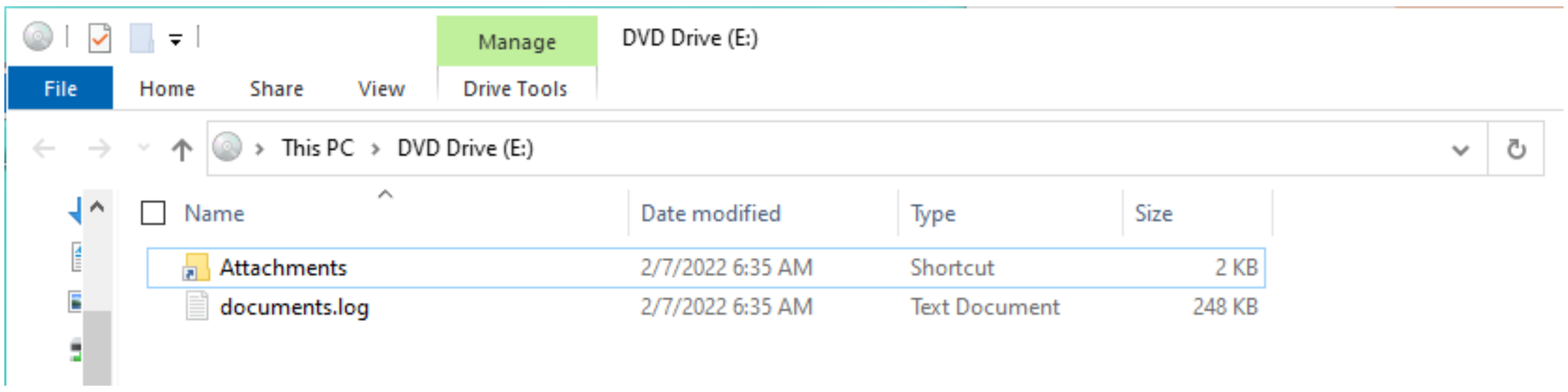
To follow along, you can grab the sample as well as the PCAP files for it on Malware-Traffic-Analysis.net.

SHA256: 0900b4eb02bdcaefd21df169d21794c8c70bfbc68b2f0612861fcabc82f28149

Step 1: Mounting ISO File & Extracting Stage 1 Executable

Recent BAZARLOADER samples arrive in emails containing OneDrive links to download an ISO file to avoid detection since most AVs tend to ignore this particular file type. With Windows 7 and above integrating the mounting functionality into Windows Explorer, we can mount any ISO file as a virtual drive by double-clicking on it.

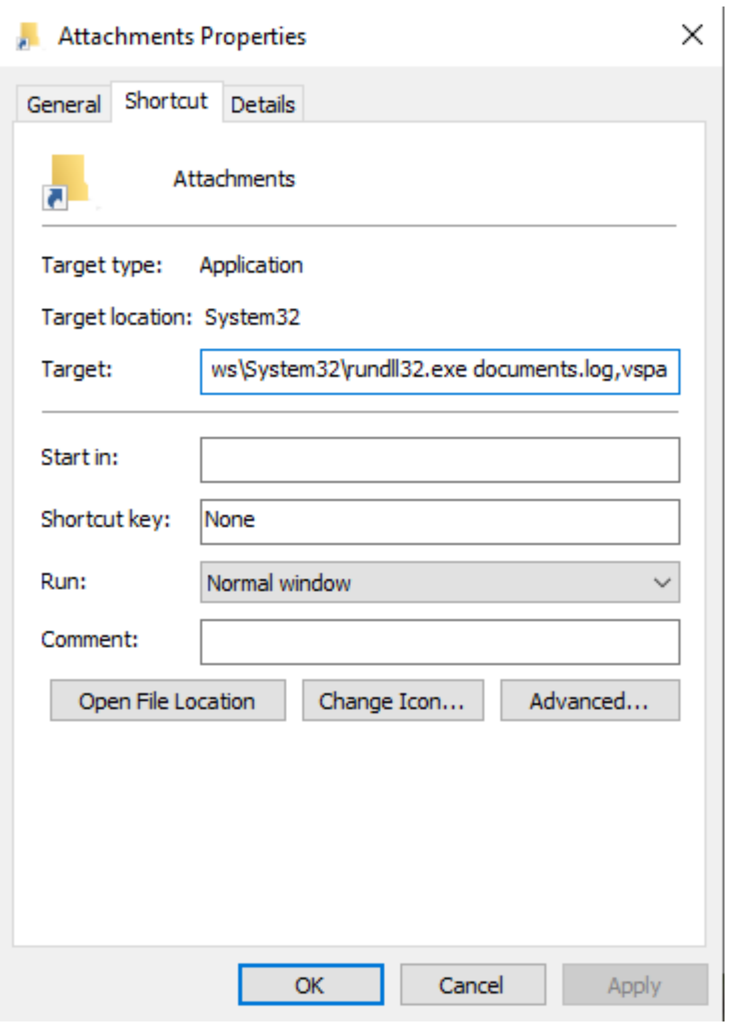
When we mount the malicious ISO file, we see that a drive is mounted on the system that contains a shortcut file named "Attachments.lnk" and a hidden file named "documents.log".



Name	Date modified	Type	Size
Attachments	2/7/2022 6:35 AM	Shortcut	2 KB
documents.log	2/7/2022 6:35 AM	Text Document	248 KB

The shortcut file has to be run by the victim to begin the chain of infection. We can quickly extract the actual command being executed by this shortcut from its Properties window.

```
C:\Windows\System32\rundll32.exe documents.log,vspa
```



Attachments Properties

Target type: Application

Target location: System32

Target: ws\System32\rundll32.exe documents.log,vspa

Start in:

Shortcut key: None

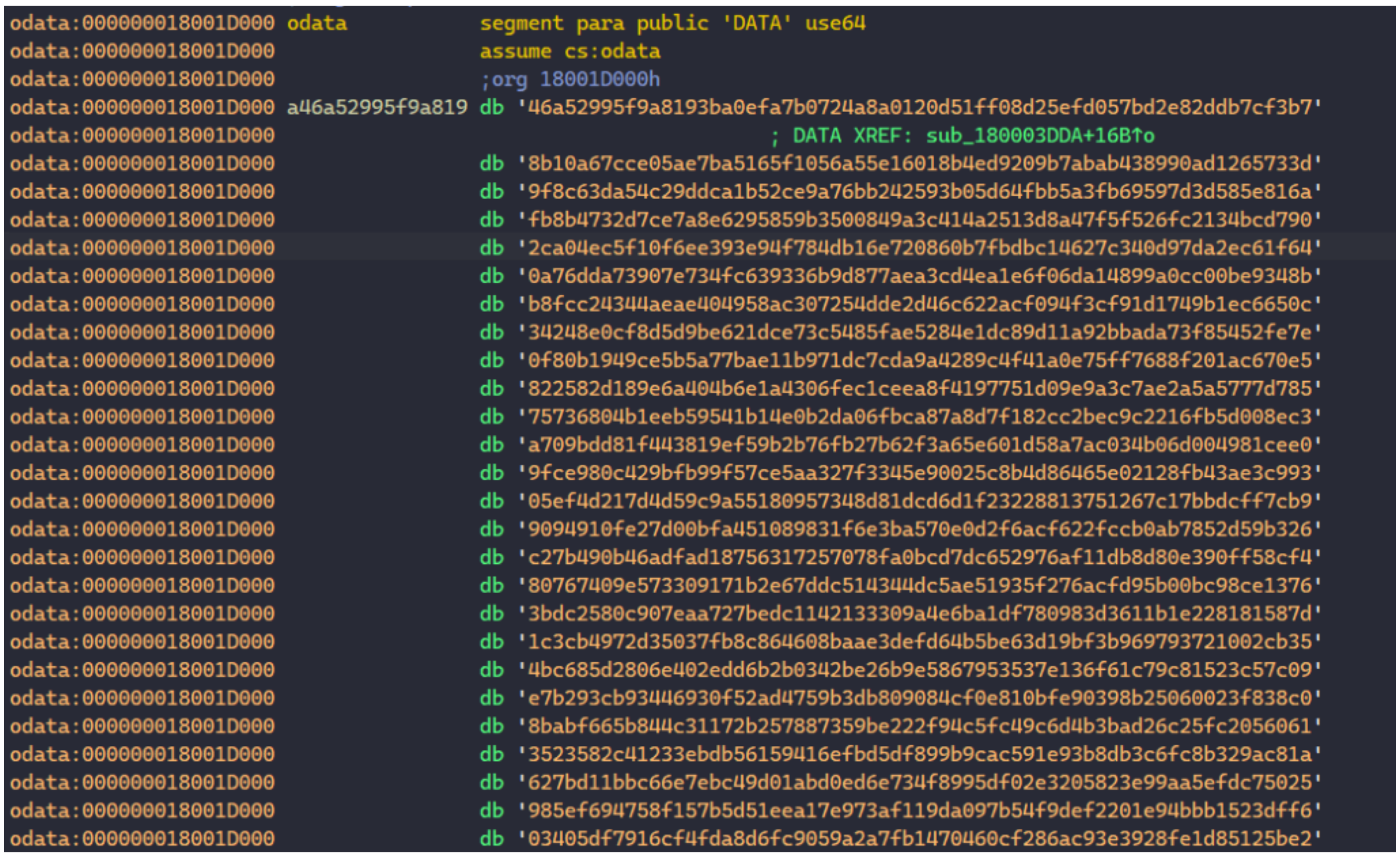
Run: Normal window

Comment:

Once the victim double-clicks on the shortcut file, the command executes the Windows rundll32.exe program to launch the “documents.log” file. This lets us know that the file being launched is a DLL file, and the entry point is its export function vspa.

Step 2: Extracting Second Stage Shellcode

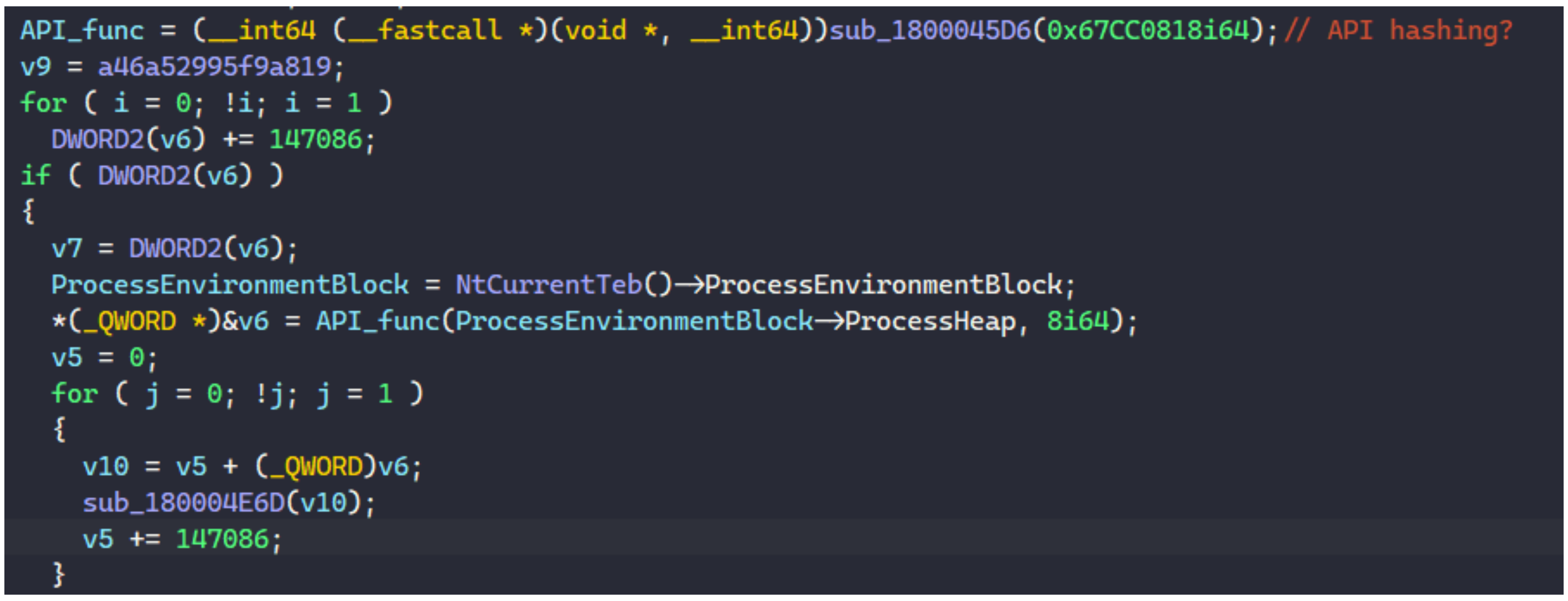
Taking a quick look in IDA, we can somewhat tell that the extracted DLL is packed since it has only a few functions and a really suspicious looking buffer of ASCII characters in its custom .odata section.



```
odata:000000018001D000 odata           segment para public 'DATA' use64
odata:000000018001D000                 assume cs:odata
odata:000000018001D000                 ;org 18001D000h
odata:000000018001D000 a46a52995f9a819 db '46a52995f9a8193ba0efa7b0724a8a0120d51ff08d25efd057bd2e82ddb7cf3b7'
odata:000000018001D000                                         ; DATA XREF: sub_180003DDA+16B↑o
odata:000000018001D000                 db '8b10a67cce05ae7ba5165f1056a55e16018b4ed9209b7abab438990ad1265733d'
odata:000000018001D000                 db '9f8c63da54c29ddca1b52ce9a76bb242593b05d64fbb5a3fb69597d3d585e816a'
odata:000000018001D000                 db 'fb8b4732d7ce7a8e6295859b3500849a3c414a2513d8a47f5f526fc2134bcd790'
odata:000000018001D000                 db '2ca04ec5f10f6ee393e94f784db16e720860b7fbdbc14627c340d97da2ec61f64'
odata:000000018001D000                 db '0a76dda73907e734fc639336b9d877aea3cd4ea1e6f06da14899a0cc00be9348b'
odata:000000018001D000                 db 'b8fcc24344aeae404958ac307254dde2d46c622acf094f3cf91d1749b1ec6650c'
odata:000000018001D000                 db '34248e0cf8d5d9be621dce73c5485fae5284e1dc89d11a92bbada73f85452fe7e'
odata:000000018001D000                 db '0f80b1949ce5b5a77bae11b971dc7cda9a4289c4f41a0e75ff7688f201ac670e5'
odata:000000018001D000                 db '822582d189e6a404b6e1a4306fec1ceea8f4197751d09e9a3c7ae2a5a5777d785'
odata:000000018001D000                 db '75736804b1eeb59541b14e0b2da06fbca87a8d7f182cc2bec9c2216fb5d008ec3'
odata:000000018001D000                 db 'a709bdd81f443819ef59b2b76fb27b62f3a65e601d58a7ac034b06d004981cee0'
odata:000000018001D000                 db '9fce980c429bfb99f57ce5aa327f3345e90025c8b4d86465e02128fb43ae3c993'
odata:000000018001D000                 db '05ef4d217d4d59c9a551809573 48d81dcd6d1f23228813751267c17bbdcff7cb9'
odata:000000018001D000                 db '9094910fe27d00bfa451089831f6e3ba570e0d2f6acf622fccb0ab7852d59b326'
odata:000000018001D000                 db 'c27b490b46adfad18756317257078fa0bcd7dc652976af11db8d80e390ff58cf4'
odata:000000018001D000                 db '80767409e573309171b2e67ddc514344dc5ae51935f276acfd95b00bc98ce1376'
odata:000000018001D000                 db '3bdc2580c907eaa727bedc1142133309a4e6ba1df780983d3611b1e228181587d'
odata:000000018001D000                 db '1c3cb4972d35037fb8c864608baae3defd64b5be63d19bf3b969793721002cb35'
odata:000000018001D000                 db '4bc685d2806e402edd6b2b0342be26b9e5867953537e136f61c79c81523c57c09'
odata:000000018001D000                 db 'e7b293cb93446930f52ad4759b3db809084cf0e810bfe90398b25060023f838c0'
odata:000000018001D000                 db '8babf665b844c31172b257887359be222f94c5fc49c6d4b3bad26c25fc2056061'
odata:000000018001D000                 db '3523582c41233ebdb56159416efbd5df899b9cac591e93b8db3c6fc8b329ac81a'
odata:000000018001D000                 db '627bd11bbc66e7ebc49d01abd0ed6e734f8995df02e3205823e99aa5efdc75025'
odata:000000018001D000                 db '985ef694758f157b5d51eea17e973af119da097b54f9def2201e94bbb1523dff6'
odata:000000018001D000                 db '03405df7916cf4fda8d6fc9059a2a7fb1470460cf286ac93e3928fe1d85125be2'
```

With that in mind, we will just perform some quick static analysis to determine where we can dump the next stage.

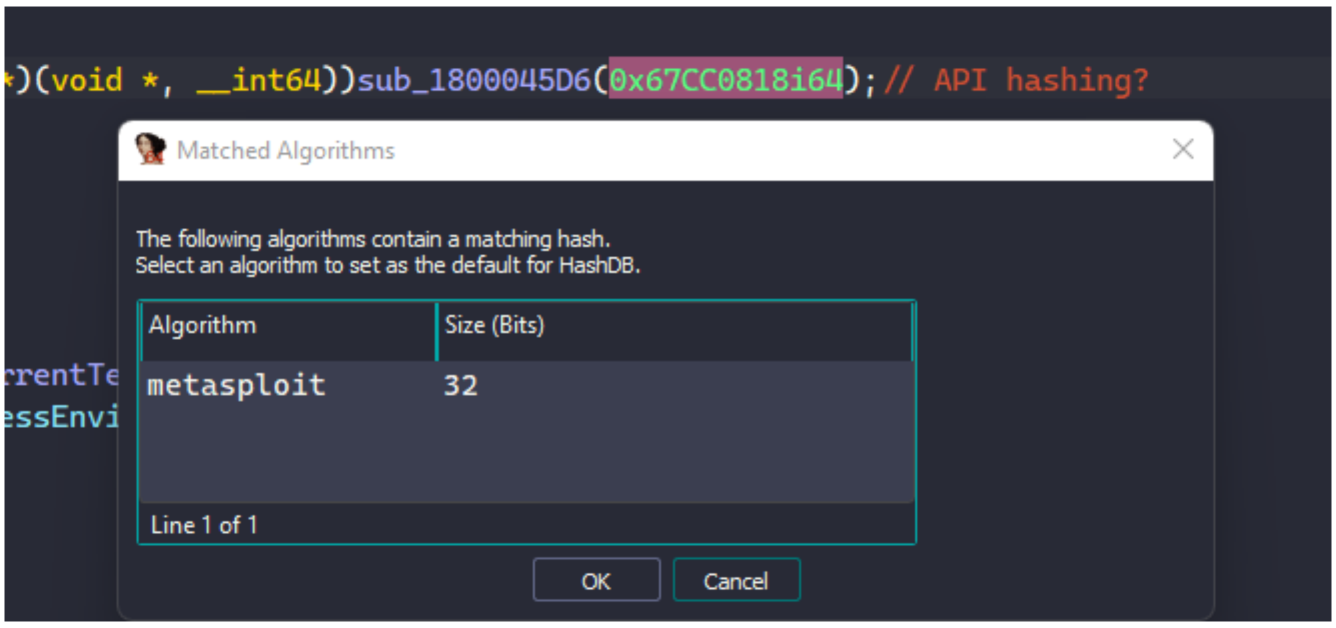
In the first function of the vspa export, we see sub_1800045D6 takes a DWORD in as the parameter. This function returns a variable that contains an address to a function that is later called in the code.



```
API_func = (__int64 (__fastcall *)(void *, __int64))sub_1800045D6(0x67CC0818i64);// API hashing?
v9 = a46a52995f9a819;
for ( i = 0; !i; i = 1 )
  DWORD2(v6) += 147086;
if ( DWORD2(v6) )
{
  v7 = DWORD2(v6);
  ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
  *(_QWORD *)&v6 = API_func(ProcessEnvironmentBlock->ProcessHeap, 8i64);
  v5 = 0;
  for ( j = 0; !j; j = 1 )
  {
    v10 = v5 + (_QWORD)v6;
    sub_180004E6D(v10);
    v5 += 147086;
  }
```

At this point, we can safely guess that sub_1800045D6 is an API resolving function, and the parameter it takes is the hash of the API's name. Because this is still the unpacking phase, we won't dive too deep into analyzing this function.

Instead, I'll just use OALabs's HashDB IDA plugin to quickly reverse-lookup the hashing algorithm used from the hash. The result shows that the hash corresponds to an API name hashed with Metasploit's hashing algorithm ROR13.



```
*)(void *, __int64))sub_1800045D6(0x67CC0818i64);// API hashing?
```

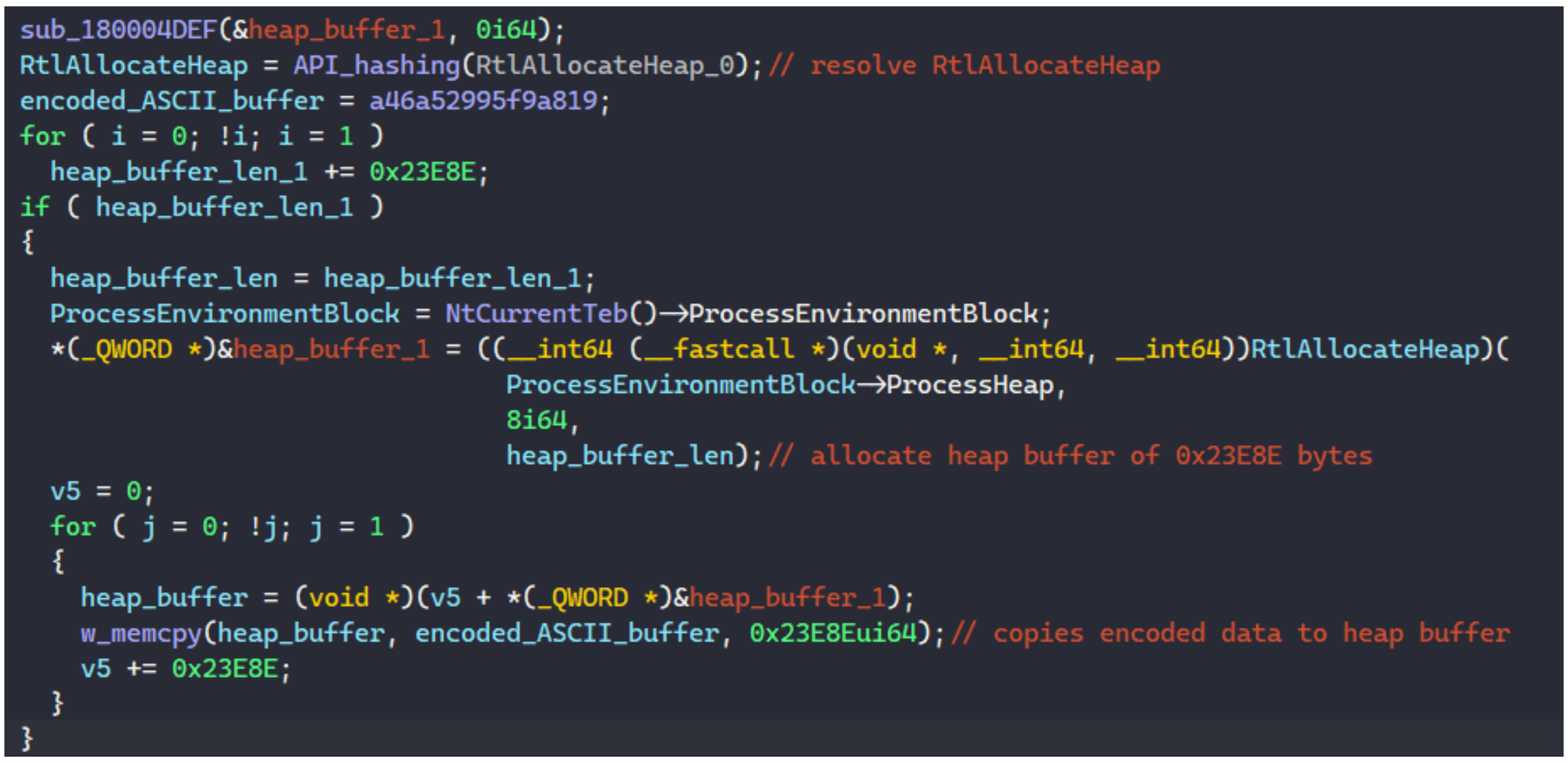
Matched Algorithms

The following algorithms contain a matching hash.
Select an algorithm to set as the default for HashDB.

Algorithm	Size (Bits)
metasploit	32

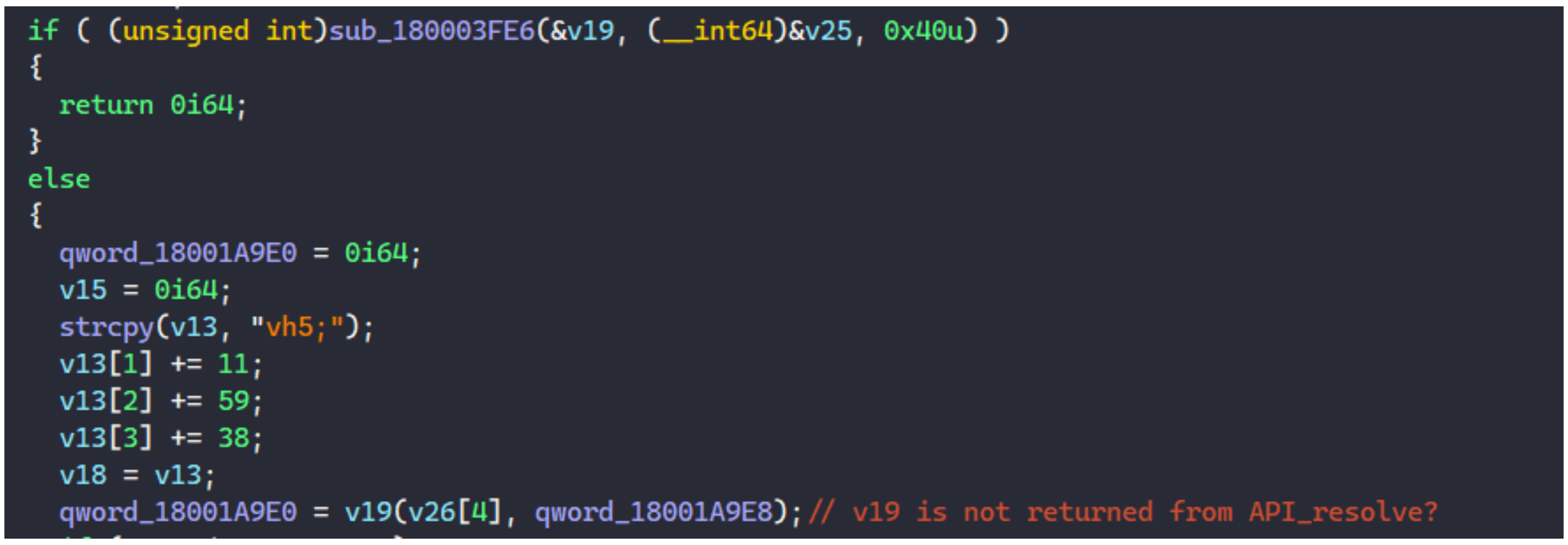
Line 1 of 1

After determining the hashing algorithm, we can use HashDB to quickly look up the APIs being resolved by this function. It becomes clear that this function resolves the RtlAllocateHeap API, calls that to allocate a heap buffer and writes the encoded ASCII data to it.



```
sub_180004DEF(&heap_buffer_1, 0i64);
RtlAllocateHeap = API_hashing(RtlAllocateHeap_0);// resolve RtlAllocateHeap
encoded_ASCII_buffer = a46a52995f9a819;
for ( i = 0; !i; i = 1 )
  heap_buffer_len_1 += 0x23E8E;
if ( heap_buffer_len_1 )
{
  heap_buffer_len = heap_buffer_len_1;
  ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
  *(_QWORD *)&heap_buffer_1 = ((__int64 (__fastcall *)(void *, __int64, __int64))RtlAllocateHeap)(
                               ProcessEnvironmentBlock->ProcessHeap,
                               8i64,
                               heap_buffer_len);// allocate heap buffer of 0x23E8E bytes
  v5 = 0;
  for ( j = 0; !j; j = 1 )
  {
    heap_buffer = (void *)(v5 + *(_QWORD *)&heap_buffer_1);
    w_memcpy(heap_buffer, encoded_ASCII_buffer, 0x23E8Eui64);// copies encoded data to heap buffer
    v5 += 0x23E8E;
  }
}
```

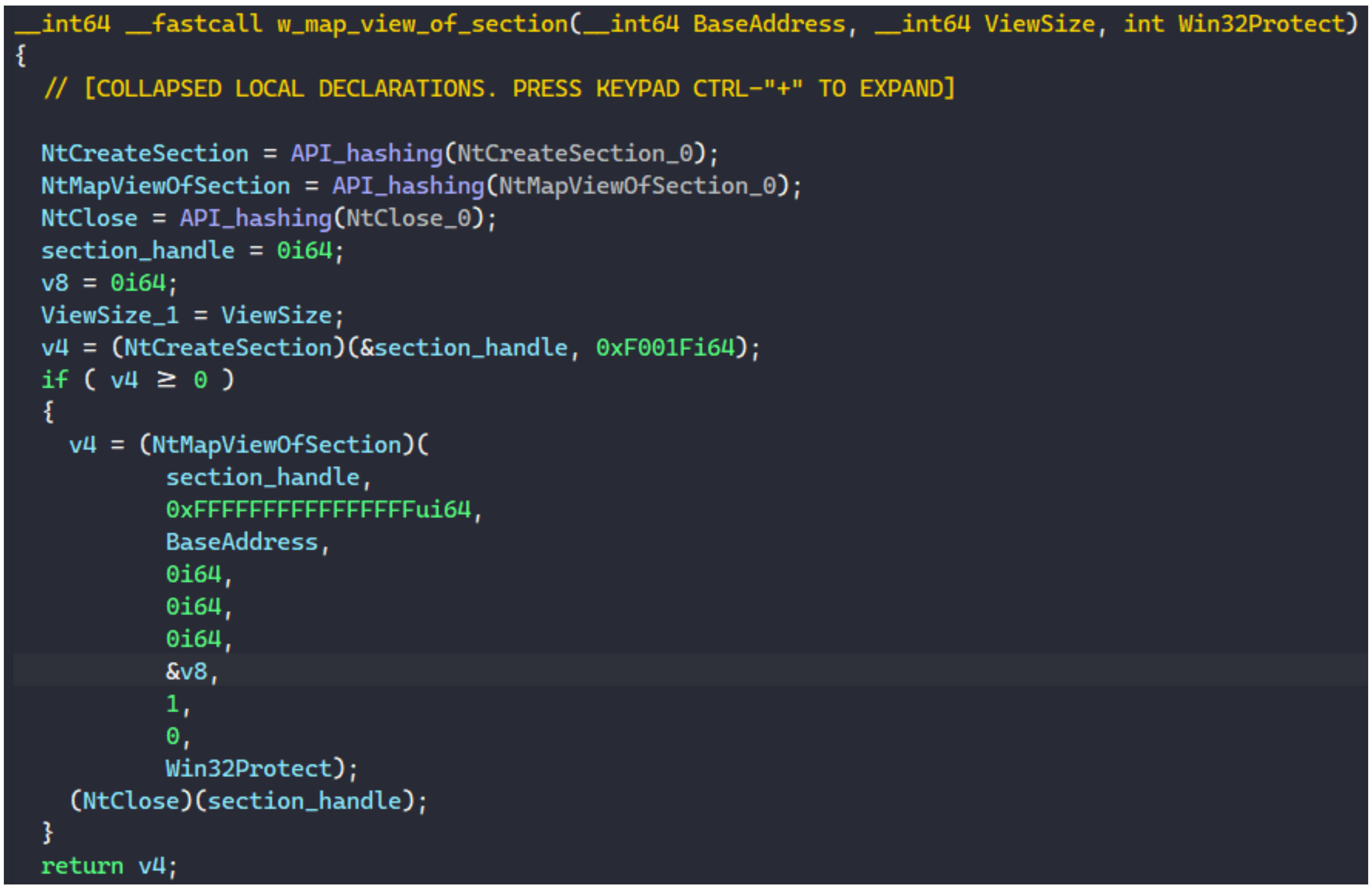
From this point onward, we can guess that the packer will decode this buffer and launch it somewhere later in the code. If we skip toward the end of the vspa export, we see a call instruction on a variable that is not returned from the API resolving function, so it can potentially be our tail jump.



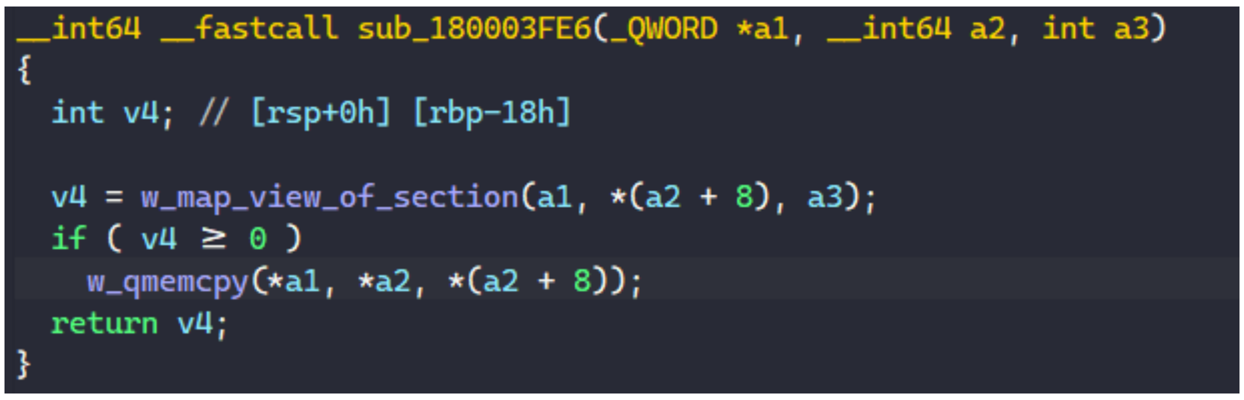
```
if ( (unsigned int)sub_180003FE6(&v19, (__int64)&v25, 0x40u) )
{
  return 0i64;
}
else
{
  qword_18001A9E0 = 0i64;
  v15 = 0i64;
  strcpy(v13, "vh5;");
  v13[1] += 11;
  v13[2] += 59;
  v13[3] += 38;
  v18 = v13;
  qword_18001A9E0 = v19(v26[4], qword_18001A9E8);// v19 is not returned from API_resolve?
```

The last function to modify that v19 variable is sub_180003FE6, so we can quickly take a look at that.

It turns out the sub_180003FE6 function just resolves and calls NtMapViewOfSection to map a view of a section into the virtual address space and writes the base address of the view into the v19 variable. Then, it just executes qmemcpy to copy the data in the second variable to the returned virtual base address.



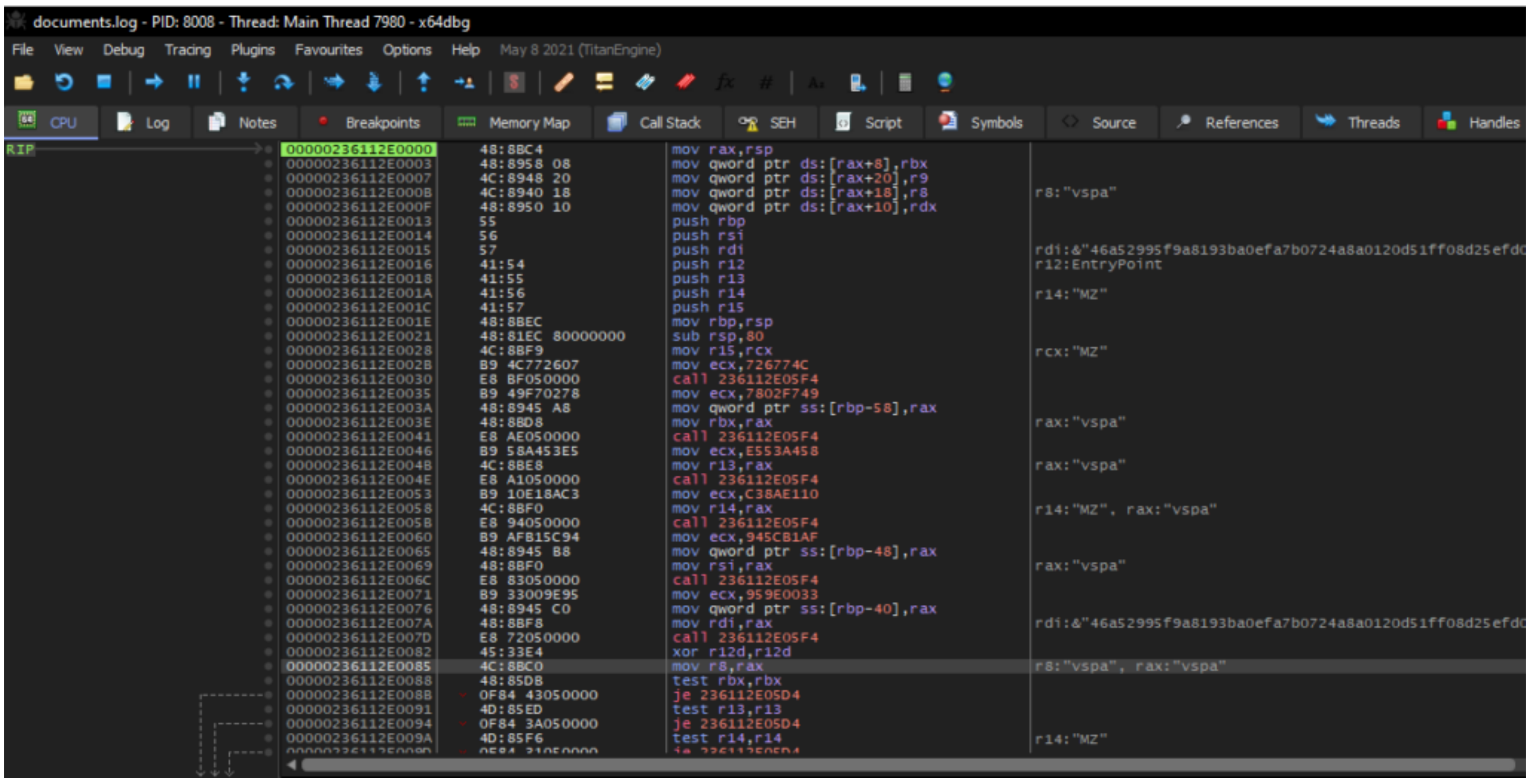
```
__int64 __fastcall w_map_view_of_section(__int64 BaseAddress, __int64 ViewSize, int Win32Protect)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  NtCreateSection = API_hashing(NtCreateSection_0);
  NtMapViewOfSection = API_hashing(NtMapViewOfSection_0);
  NtClose = API_hashing(NtClose_0);
  section_handle = 0i64;
  v8 = 0i64;
  ViewSize_1 = ViewSize;
  v4 = (NtCreateSection)(&section_handle, 0xF001Fi64);
  if ( v4 ≥ 0 )
  {
    v4 = (NtMapViewOfSection)(
           section_handle,
           0xFFFFFFFFFFFFFFFFui64,
           BaseAddress,
           0i64,
           0i64,
           0i64,
           &v8,
           1,
           0,
           Win32Protect);
    (NtClose)(section_handle);
  }
  return v4;
```



```
__int64 __fastcall sub_180003FE6(_QWORD *a1, __int64 a2, int a3)
{
  int v4; // [rsp+0h] [rbp-18h]

  v4 = w_map_view_of_section(a1, *(a2 + 8), a3);
  if ( v4 ≥ 0 )
    w_qmemcpy(*a1, *a2, *(a2 + 8));
  return v4;
}
```

This tells us two things. First, our guess that the v19 variable will contain the address to executable code is correct. Second, we know that the executable code is shellcode since the data is mapped and executed directly at offset 0 from where it is written.

From here, we can set up x64dbg, execute the DLL file at the vspa export, and break at the call instruction. After stepping into the function, we will be at the head of the shellcode.

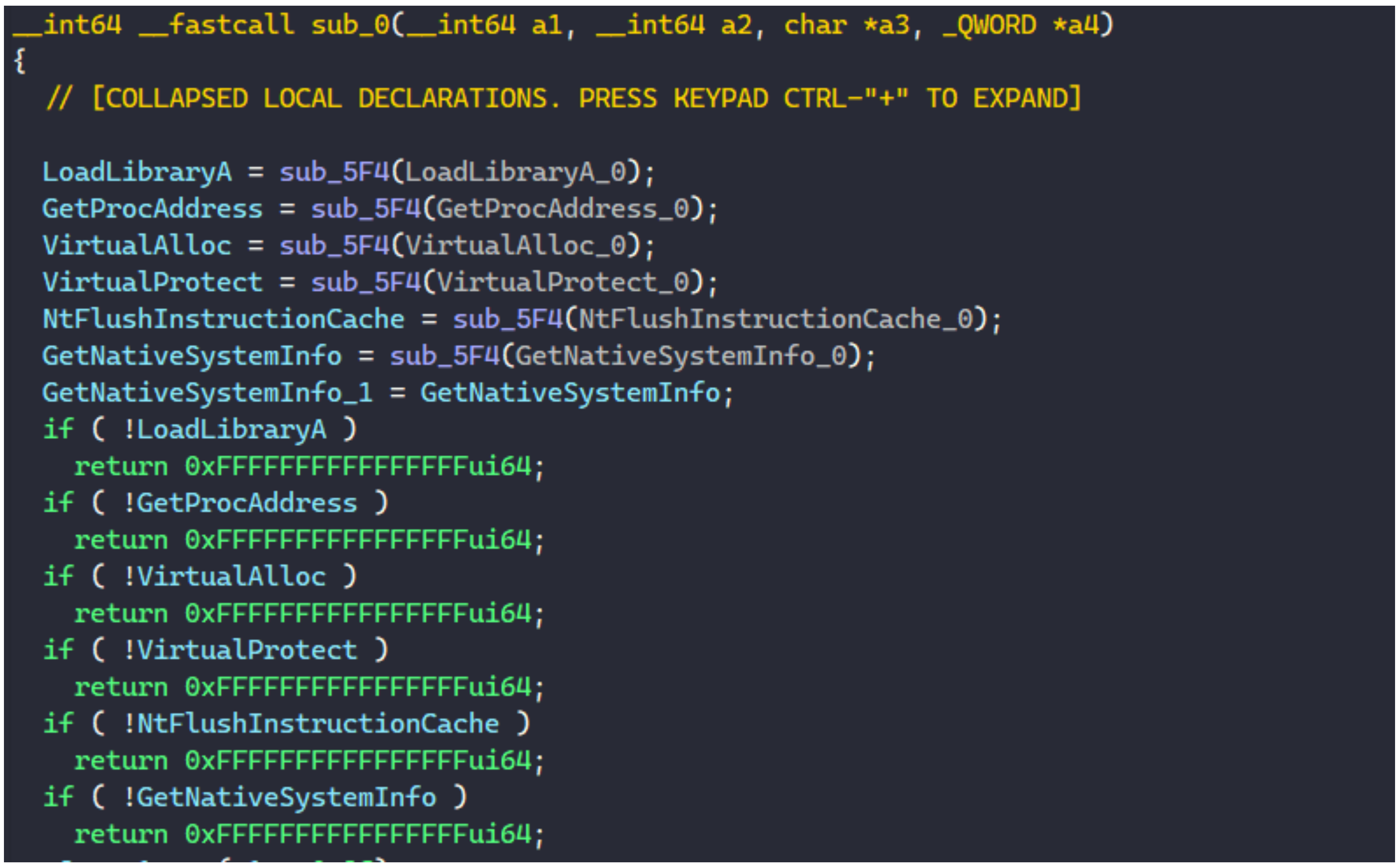

documents.log - PID: 8008 - Thread: Main Thread 7980 - x64dbg
File View Debug Tracing Plugins Favourites Options Help May 8 2021 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles

```
RIP → 00000236112E0000   48:8BC4            mov rax,rsp
      00000236112E0003   48:8958 08         mov qword ptr ds:[rax+8],rbx
      00000236112E0007   4C:8948 20         mov qword ptr ds:[rax+20],r9
      00000236112E000B   4C:8940 18         mov qword ptr ds:[rax+18],r8      r8:"vspa"
      00000236112E000F   48:8950 10         mov qword ptr ds:[rax+10],rdx
      00000236112E0013   55                 push rbp
      00000236112E0014   56                 push rsi
      00000236112E0015   57                 push rdi                          rdi:&"46a52995f9a8193ba0efa7b0724a8a0120d51ff08d25efd
      00000236112E0016   41:54              push r12                          r12:EntryPoint
      00000236112E0018   41:55              push r13
      00000236112E001A   41:56              push r14                          r14:"MZ"
      00000236112E001C   41:57              push r15
      00000236112E001E   48:8BEC            mov rbp,rsp
      00000236112E0021   48:81EC 80000000   sub rsp,80
      00000236112E0028   4C:8BF9            mov r15,rcx                       rcx:"MZ"
      00000236112E002B   B9 4C772607        mov ecx,726774C
      00000236112E0030   E8 BF050000        call 236112E05F4
      00000236112E0035   B9 49F70278        mov ecx,7802F749
      00000236112E003A   48:8945 A8         mov qword ptr ss:[rbp-58],rax
      00000236112E003E   48:8BD8            mov rbx,rax                       rax:"vspa"
      00000236112E0041   E8 AE050000        call 236112E05F4
      00000236112E0046   B9 58A453E5        mov ecx,E553A458
      00000236112E004B   4C:8BE8            mov r13,rax                       rax:"vspa"
      00000236112E004E   E8 A1050000        call 236112E05F4
      00000236112E0053   B9 10E18AC3        mov ecx,C38AE110
      00000236112E0058   4C:8BF0            mov r14,rax                       r14:"MZ", rax:"vspa"
      00000236112E005B   E8 94050000        call 236112E05F4
      00000236112E0060   B9 AFB15C94        mov ecx,945CB1AF
      00000236112E0065   48:8945 B8         mov qword ptr ss:[rbp-48],rax
      00000236112E0069   48:8BF0            mov rsi,rax                       rax:"vspa"
      00000236112E006C   E8 83050000        call 236112E05F4
      00000236112E0071   B9 33009E95        mov ecx,959E0033
      00000236112E0076   48:8945 C0         mov qword ptr ss:[rbp-40],rax
      00000236112E007A   48:8BF8            mov rdi,rax                       rdi:&"46a52995f9a8193ba0efa7b0724a8a0120d51ff08d25efd
      00000236112E007D   E8 72050000        call 236112E05F4
      00000236112E0082   45:33E4            xor r12d,r12d
      00000236112E0085   4C:8BC0            mov r8,rax                        r8:"vspa", rax:"vspa"
      00000236112E0088   48:85DB            test rbx,rbx
      00000236112E008B   0F84 43050000      je 236112E05D4
      00000236112E0091   4D:85ED            test r13,r13
      00000236112E0094   0F84 3A050000      je 236112E05D4
      00000236112E009A   4D:85F6            test r14,r14                      r14:"MZ"
      00000236112E009D   0F84 31050000      je 236112E05D4
```

We can now dump this virtual memory buffer to retrieve the second stage shellcode for the next unpacking step.

Step 3: Extracting The Final BAZARLOADER Payload

When we examine the shellcode in IDA, we can quickly use the same trick with HashDB above to see that the shellcode also performs API hashing with Metasploit's ROR13.

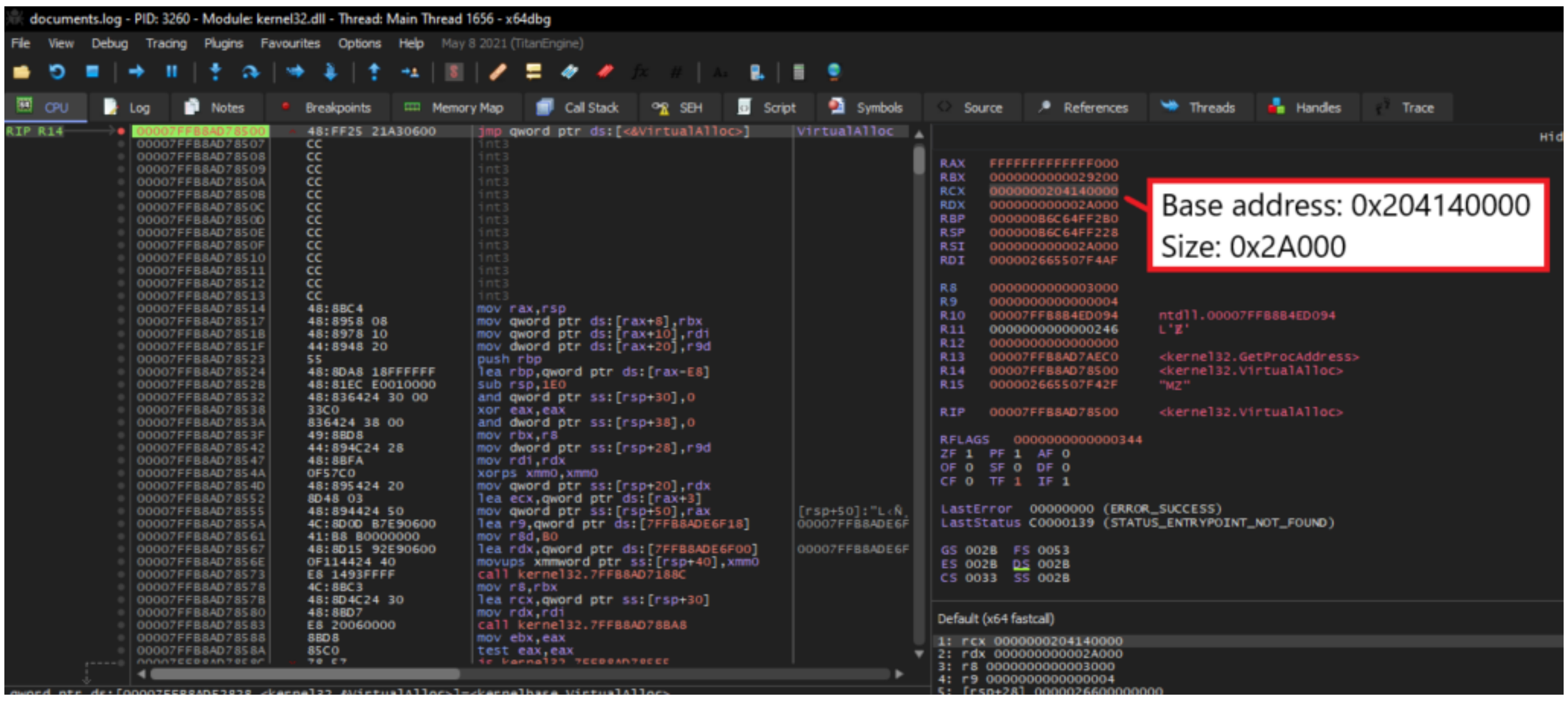


```
__int64 __fastcall sub_0(__int64 a1, __int64 a2, char *a3, _QWORD *a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  LoadLibraryA = sub_5F4(LoadLibraryA_0);
  GetProcAddress = sub_5F4(GetProcAddress_0);
  VirtualAlloc = sub_5F4(VirtualAlloc_0);
  VirtualProtect = sub_5F4(VirtualProtect_0);
  NtFlushInstructionCache = sub_5F4(NtFlushInstructionCache_0);
  GetNativeSystemInfo = sub_5F4(GetNativeSystemInfo_0);
  GetNativeSystemInfo_1 = GetNativeSystemInfo;
  if ( !LoadLibraryA )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !GetProcAddress )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !VirtualAlloc )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !VirtualProtect )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !NtFlushInstructionCache )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !GetNativeSystemInfo )
    return 0xFFFFFFFFFFFFFFFFui64;
```

At the entry point above, the shellcode resolves a set of functions that it will call, most notably VirtualAlloc and VirtualProtect. These two functions are typically used by packers to allocate virtual memory to decode and write the next stage executable in before launching it.

With this in mind, our next step should be debugging the shellcode and setting breakpoints at these two API calls. We can pick up where we are after dumping in x64dbg during Step 2, or we can launch the shellcode directly in our debugger using OALabs's BlobRunner or similar shellcode launcher.

Our first hit with VirtualAlloc is a call to allocate a virtual memory buffer at virtual address 0x204140000 with the size of 0x2A000 bytes.

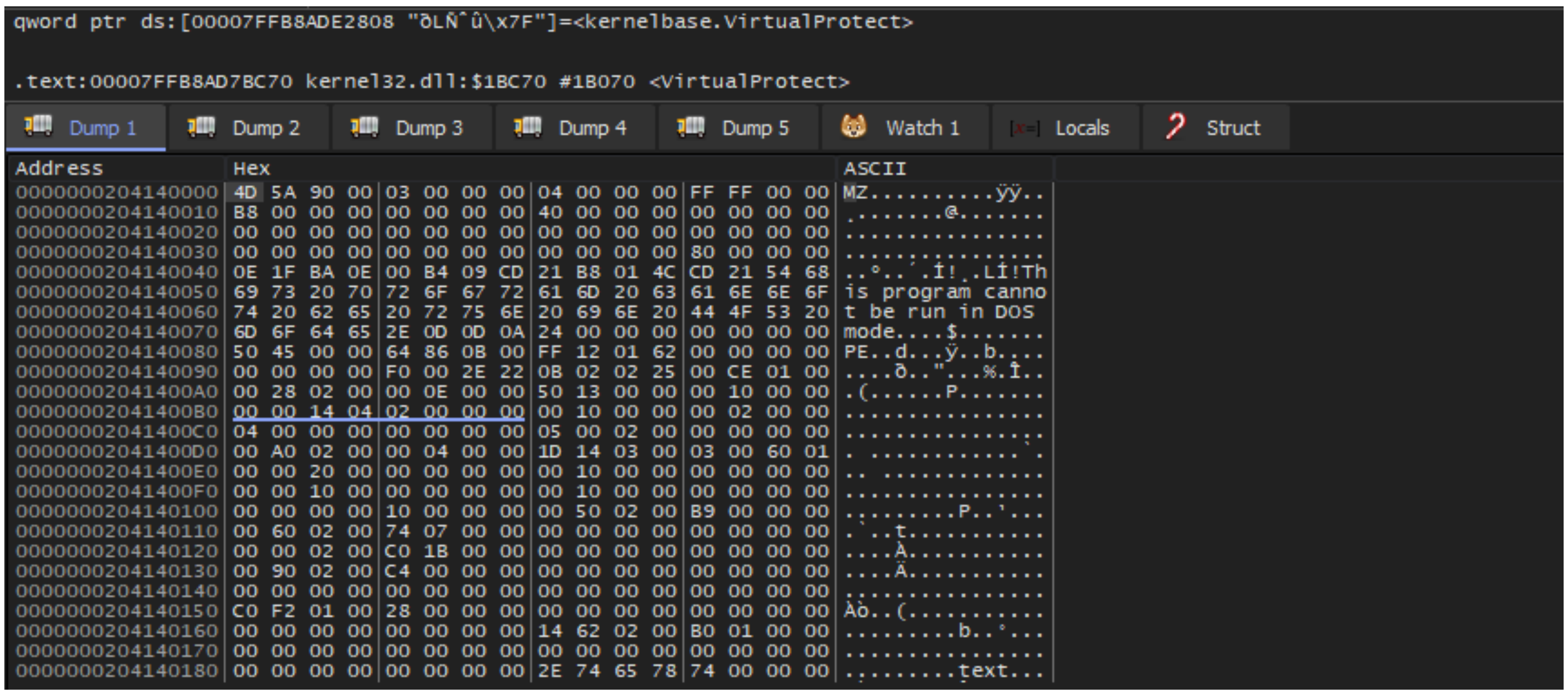

documents.log - PID: 3260 - Module: kernel32.dll - Thread: Main Thread 1656 - x64dbg
File View Debug Tracing Plugins Favourites Options Help May 8 2021 (TitanEngine)
CPU Log Notes Breakpoints Memory Map Call Stack SEH Script Symbols Source References Threads Handles Trace

```
RIP R14 → 00007FFB8AD78500  48:FF25 21A30600   jmp qword ptr ds:[<&VirtualAlloc>]   VirtualAlloc
          00007FFB8AD78507  CC                 int3
          00007FFB8AD78508  CC                 int3
          00007FFB8AD78509  CC                 int3
          00007FFB8AD7850A  CC                 int3
          00007FFB8AD7850B  CC                 int3
          00007FFB8AD7850C  CC                 int3
          00007FFB8AD7850D  CC                 int3
          00007FFB8AD7850E  CC                 int3
          00007FFB8AD7850F  CC                 int3
          00007FFB8AD78510  CC                 int3
          00007FFB8AD78511  CC                 int3
          00007FFB8AD78512  CC                 int3
          00007FFB8AD78513  CC                 int3
          00007FFB8AD78514  48:8BC4            mov rax,rsp
          00007FFB8AD78517  48:8958 08         mov qword ptr ds:[rax+8],rbx
          00007FFB8AD7851B  48:8978 10         mov qword ptr ds:[rax+10],rdi
          00007FFB8AD7851F  44:8948 20         mov dword ptr ds:[rax+20],r9d
          00007FFB8AD78523  55                 push rbp
          00007FFB8AD78524  48:8DA8 18FFFFFF   lea rbp,qword ptr ds:[rax-E8]
          00007FFB8AD7852B  48:81EC E0010000   sub rsp,1E0
          00007FFB8AD78532  48:836424 30 00    and qword ptr ss:[rsp+30],0
          00007FFB8AD78538  33C0               xor eax,eax
          00007FFB8AD7853A  836424 38 00       and dword ptr ss:[rsp+38],0
          00007FFB8AD7853F  49:8BD8            mov rbx,r8
          00007FFB8AD78542  44:894C24 28       mov dword ptr ss:[rsp+28],r9d
          00007FFB8AD78547  48:8BFA            mov rdi,rdx
          00007FFB8AD7854A  0F57C0             xorps xmm0,xmm0
          00007FFB8AD7854D  48:895424 20       mov qword ptr ss:[rsp+20],rdx
          00007FFB8AD78552  8D48 03            lea ecx,qword ptr ds:[rax+3]
          00007FFB8AD78555  48:894424 50       mov qword ptr ss:[rsp+50],rax         [rsp+50]:"L‹Ñ
          00007FFB8AD7855A  4C:8D0D B7E90600   lea r9,qword ptr ds:[7FFB8ADE6F18]    00007FFB8ADE6F
          00007FFB8AD78561  41:B8 B0000000     mov r8d,B0
          00007FFB8AD78567  48:8D15 92E90600   lea rdx,qword ptr ds:[7FFB8ADE6F00]   00007FFB8ADE6F
          00007FFB8AD7856E  0F114424 40        movups xmmword ptr ss:[rsp+40],xmm0
          00007FFB8AD78573  E8 1493FFFF        call kernel32.7FFB8AD7188C
          00007FFB8AD78578  4C:8BC3            mov r8,rbx
          00007FFB8AD7857B  48:8D4C24 30       lea rcx,qword ptr ss:[rsp+30]
          00007FFB8AD78580  48:8BD7            mov rdx,rdi
          00007FFB8AD78583  E8 20060000        call kernel32.7FFB8AD78BA8
          00007FFB8AD78588  8BD8               mov ebx,eax
          00007FFB8AD7858A  85C0               test eax,eax
```

```
RAX    FFFFFFFFFFFFF000
RBX    0000000000029200
RCX    0000000204140000
RDX    000000000002A000
RBP    000000B6C64FF2B0
RSP    000000B6C64FF228
RSI    000000000002A000
RDI    000002665507F4AF

R8     0000000000003000
R9     0000000000000004
R10    00007FFB884ED094     ntdll.00007FFB884ED094
R11    0000000000000246     L'Ɇ'
R12    0000000000000000
R13    00007FFB8AD7AEC0     <kernel32.GetProcAddress>
R14    00007FFB8AD78500     <kernel32.VirtualAlloc>
R15    000002665507F42F     "MZ"

RIP    00007FFB8AD78500     <kernel32.VirtualAlloc>

RFLAGS   0000000000000344
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus C0000139 (STATUS_ENTRYPOINT_NOT_FOUND)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0033  SS 002B

Default (x64 fastcall)
1: rcx 0000000204140000
2: rdx 000000000002A000
3: r8 0000000000003000
4: r9 0000000000000004
```

Base address: 0x204140000
Size: 0x2A000

We can run until VirtualAlloc returns and start monitoring the memory at address 0x204140000. After running until the next VirtualProtect call, we see that a valid PE executable has been written to this memory region.



```
qword ptr ds:[00007FFB8ADE2808 "ðLÑˆû\x7F"]=<kernelbase.VirtualProtect>

.text:00007FFB8AD7BC70 kernel32.dll:$1BC70 #1B070 <VirtualProtect>
```

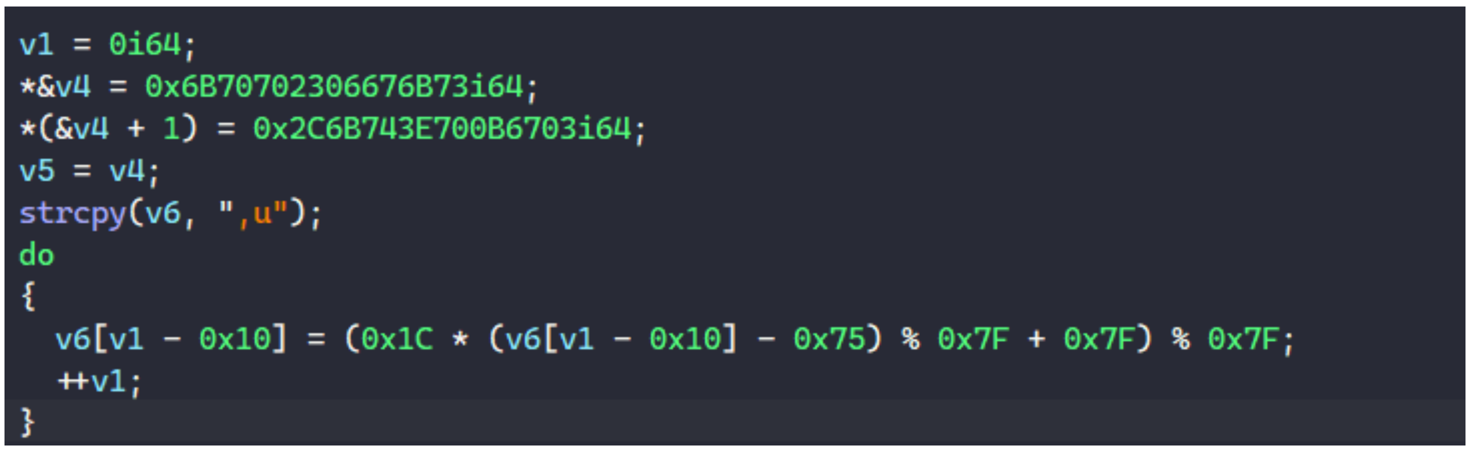
```
Address          Hex                                              ASCII
0000000204140000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
0000000204140010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ,.......@.......
0000000204140020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000000204140030 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ................
0000000204140040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
0000000204140050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
0000000204140060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
0000000204140070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
0000000204140080 50 45 00 00 64 86 0B 00 FF 12 01 62 00 00 00 00  PE..d...ÿ..b....
0000000204140090 00 00 00 00 F0 00 2E 22 0B 02 02 25 00 CE 01 00  ....ð.."...%.Î..
00000002041400A0 00 28 02 00 00 0E 00 00 50 13 00 00 00 10 00 00  .(......P.......
00000002041400B0 00 00 14 04 02 00 00 00 00 10 00 00 00 02 00 00  ................
00000002041400C0 04 00 00 00 00 00 00 00 05 00 02 00 00 00 00 00  ................
00000002041400D0 00 A0 02 00 00 04 00 00 1D 14 03 00 03 00 60 01  . ............`.
00000002041400E0 00 00 20 00 00 00 00 00 00 10 00 00 00 00 00 00  .. .............
00000002041400F0 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00  ................
0000000204140100 00 00 00 00 10 00 00 00 00 50 02 00 B9 00 00 00  .........P..¹...
0000000204140110 00 60 02 00 74 07 00 00 00 00 00 00 00 00 00 00  .`..t...........
0000000204140120 00 00 02 00 C0 1B 00 00 00 00 00 00 00 00 00 00  ....À...........
0000000204140130 00 90 02 00 C4 00 00 00 00 00 00 00 00 00 00 00  ....Ä...........
0000000204140140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000000204140150 C0 F2 01 00 28 00 00 00 00 00 00 00 00 00 00 00  Àò..(...........
0000000204140160 00 00 00 00 00 00 00 00 14 62 02 00 B0 01 00 00  .........b..°...
0000000204140170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000000204140180 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
```

Finally, we can dump this memory region into a file to extract the BAZARLOADER payload.

Step 4: BAZARLOADER’s String Obfuscation

As we begin performing static analysis on BAZARLOADER, it is crucial that we identify obfuscation methods that the malware uses.

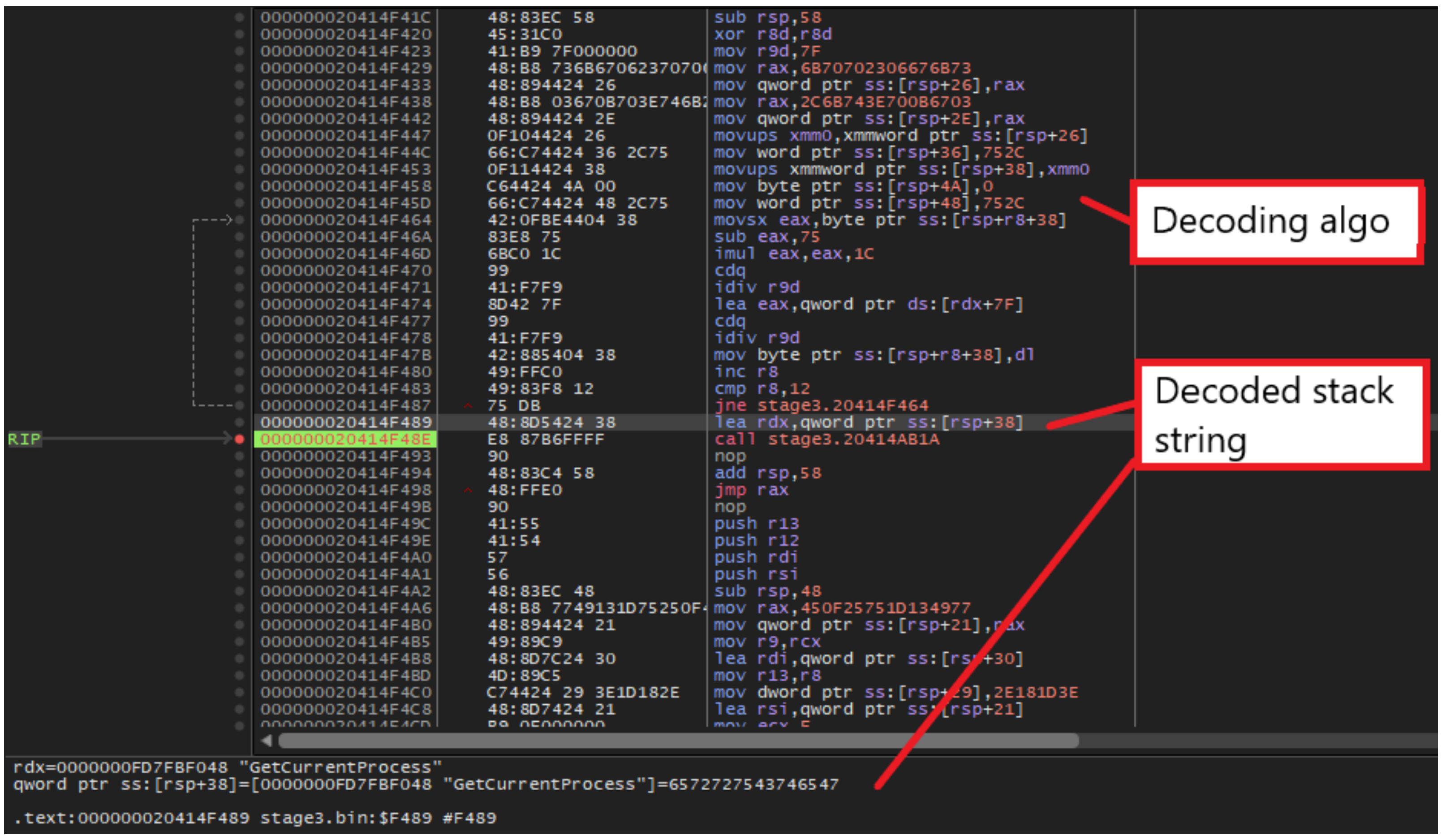
One of those methods is string obfuscation, where the malware uses encoded stack strings to hide them from static analysis.



```
v1 = 0i64;
*&v4 = 0x6B70702306676B73i64;
*(&v4 + 1) = 0x2C6B743E700B6703i64;
v5 = v4;
strcpy(v6, ",u");
do
{
  v6[v1 - 0x10] = (0x1C * (v6[v1 - 0x10] - 0x75) % 0x7F + 0x7F) % 0x7F;
  ++v1;
}
```

As shown, a typical encoded string is pushed on the stack and decoded dynamically using some multiplication, subtraction, and modulus operations.

There are different ways to resolve these stack strings, such as writing IDAPython scripts, emulation, or just running the program in a debugger and dumping the stack strings when they are resolved.



```
000000020414F41C   48:83EC 58                 sub rsp,58
000000020414F420   45:31C0                    xor r8d,r8d
000000020414F423   41:B9 7F000000             mov r9d,7F
000000020414F429   48:B8 736B6706237070(      mov rax,6B70702306676B73
000000020414F433   48:894424 26               mov qword ptr ss:[rsp+26],rax
000000020414F438   48:B8 03670B703E746B2      mov rax,2C6B743E700B6703
000000020414F442   48:894424 2E               mov qword ptr ss:[rsp+2E],rax
000000020414F447   0F104424 26                movups xmm0,xmmword ptr ss:[rsp+26]
000000020414F44C   66:C74424 36 2C75          mov word ptr ss:[rsp+36],752C
000000020414F453   0F114424 38                movups xmmword ptr ss:[rsp+38],xmm0
000000020414F458   C64424 4A 00               mov byte ptr ss:[rsp+4A],0
000000020414F45D   66:C74424 48 2C75          mov word ptr ss:[rsp+48],752C
000000020414F464   42:0FBE4404 38             movsx eax,byte ptr ss:[rsp+r8+38]
000000020414F46A   83E8 75                    sub eax,75
000000020414F46D   6BC0 1C                    imul eax,eax,1C
000000020414F470   99                         cdq
000000020414F471   41:F7F9                    idiv r9d
000000020414F474   8D42 7F                    lea eax,qword ptr ds:[rdx+7F]
000000020414F477   99                         cdq
000000020414F478   41:F7F9                    idiv r9d
000000020414F47B   42:885404 38               mov byte ptr ss:[rsp+r8+38],dl
000000020414F480   49:FFC0                    inc r8
000000020414F483   49:83F8 12                 cmp r8,12
000000020414F487   75 DB                      jne stage3.20414F464
000000020414F489   48:8D5424 38               lea rdx,qword ptr ss:[rsp+38]
RIP 000000020414F48E   E8 87B6FFFF            call stage3.20414AB1A
000000020414F493   90                         nop
000000020414F494   48:83C4 58                 add rsp,58
000000020414F498   48:FFE0                    jmp rax
000000020414F49B   90                         nop
000000020414F49C   41:55                      push r13
000000020414F49E   41:54                      push r12
000000020414F4A0   57                         push rdi
000000020414F4A1   56                         push rsi
000000020414F4A2   48:83EC 48                 sub rsp,48
000000020414F4A6   48:B8 7749131D75250F4      mov rax,450F25751D134977
000000020414F4B0   48:894424 21               mov qword ptr ss:[rsp+21],rax
000000020414F4B5   49:89C9                    mov r9,rcx
000000020414F4B8   48:8D7C24 30               lea rdi,qword ptr ss:[rsp+30]
000000020414F4BD   4D:89C5                    mov r13,r8
000000020414F4C0   C74424 29 3E1D182E         mov dword ptr ss:[rsp+29],2E181D3E
000000020414F4C8   48:8D7424 21               lea rsi,qword ptr ss:[rsp+21]

rdx=0000000FD7FBF048 "GetCurrentProcess"
qword ptr ss:[rsp+38]=[0000000FD7FBF048 "GetCurrentProcess"]=6572727543746547

.text:000000020414F489 stage3.bin:$F489 #F489
```

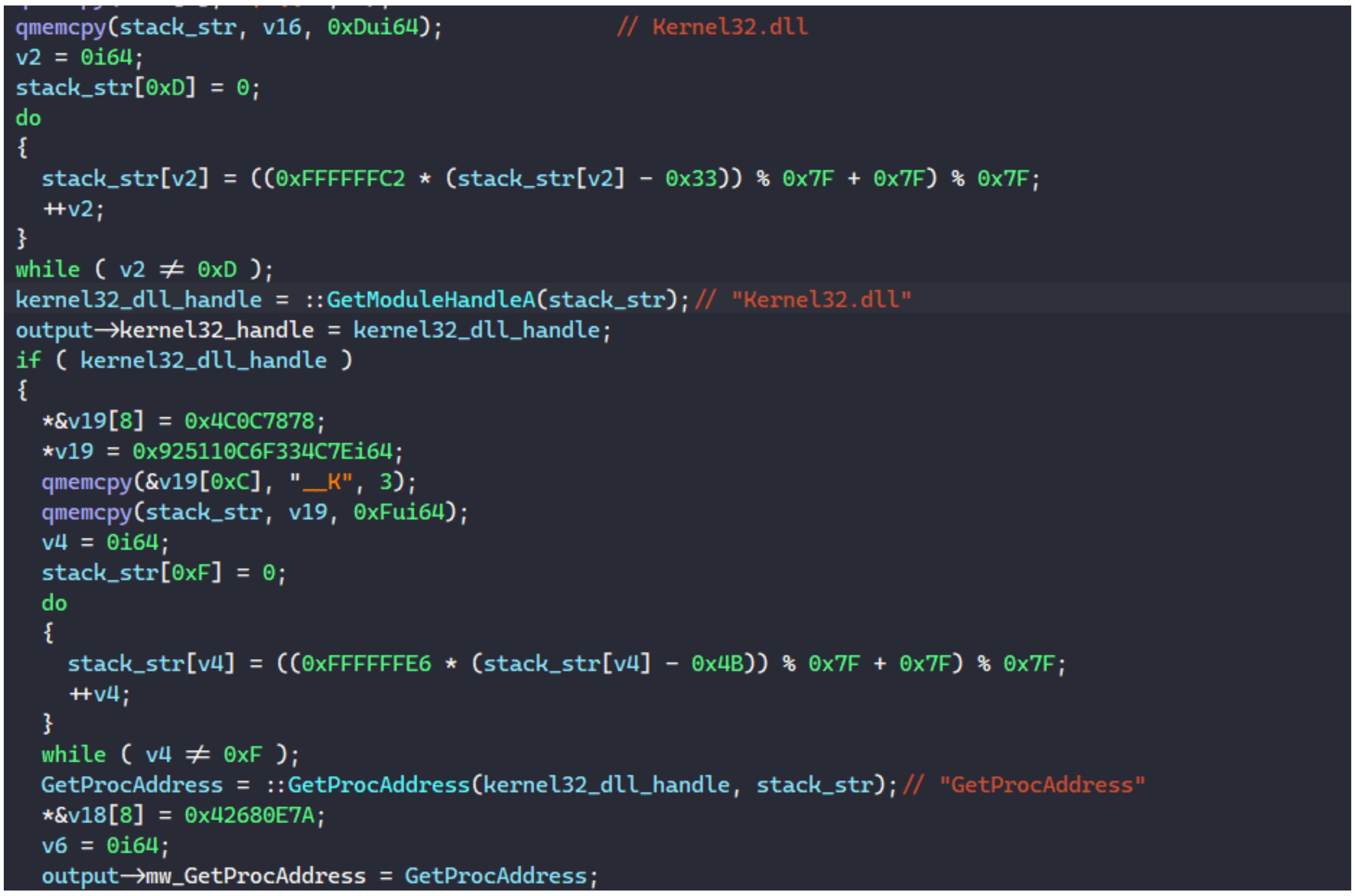
Step 5: BAZARLOADER's API Obfuscation

BAZARLOADER obfuscates most of its API calls through a few structures that it constructs in the DllEntryPoint function.

First, the malware populates the following structure that contains a handle to Kernel32.dll and addresses to API required to load libraries and get their API addresses.

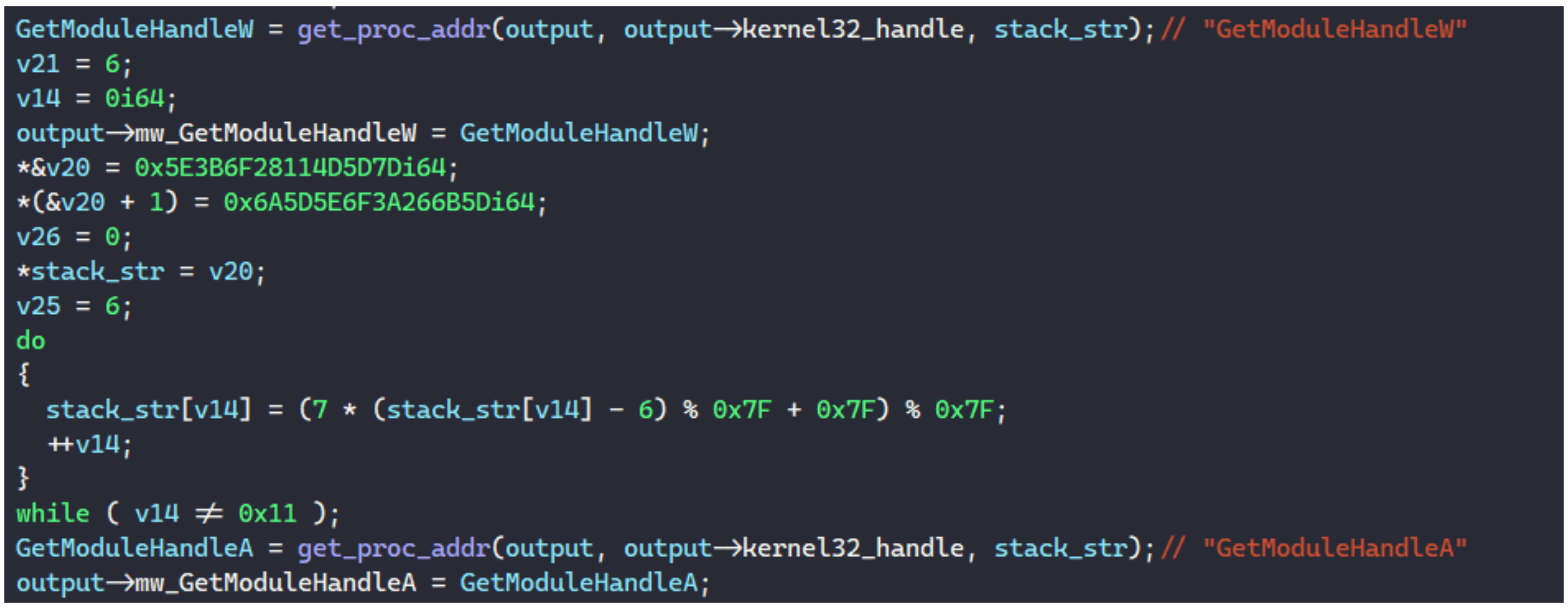
```
struct API_IMPORT_STRUCT { HANDLE kernel32_handle; FARPROC mw_GetProcAddress; FARPROC mw_LoadLibraryW;
FARPROC mw_LoadLibraryA; FARPROC mw_LoadLibraryA2; FARPROC mw_FreeLibrary; FARPROC mw_GetModuleHandleW;
FARPROC mw_GetModuleHandleA; };
```

It calls GetModuleHandle to retrieve the handle to Kernel32.dll, calls GetProcAddress to retrieve the address of the GetProcAddress API, and writes those in the structure.



```
qmemcpy(stack_str, v16, 0xDui64);              // Kernel32.dll
v2 = 0i64;
stack_str[0xD] = 0;
do
{
  stack_str[v2] = ((0xFFFFFFC2 * (stack_str[v2] - 0x33)) % 0x7F + 0x7F) % 0x7F;
  ++v2;
}
while ( v2 ≠ 0xD );
kernel32_dll_handle = ::GetModuleHandleA(stack_str);// "Kernel32.dll"
output→kernel32_handle = kernel32_dll_handle;
if ( kernel32_dll_handle )
{
  *&v19[8] = 0x4C0C7878;
  *v19 = 0x925110C6F334C7Ei64;
  qmemcpy(&v19[0xC], "__K", 3);
  qmemcpy(stack_str, v19, 0xFui64);
  v4 = 0i64;
  stack_str[0xF] = 0;
  do
  {
    stack_str[v4] = ((0xFFFFFFE6 * (stack_str[v4] - 0x4B)) % 0x7F + 0x7F) % 0x7F;
    ++v4;
  }
  while ( v4 ≠ 0xF );
  GetProcAddress = ::GetProcAddress(kernel32_dll_handle, stack_str);// "GetProcAddress"
  *&v18[8] = 0x42680E7A;
  v6 = 0i64;
  output→mw_GetProcAddress = GetProcAddress;
```

Using the structure's GetProcAddress API field, BAZARLOADER retrieves the rest of the required APIs to populate other fields in the structure. This API_IMPORT_STRUCT structure will later be used to import other libraries' APIs.



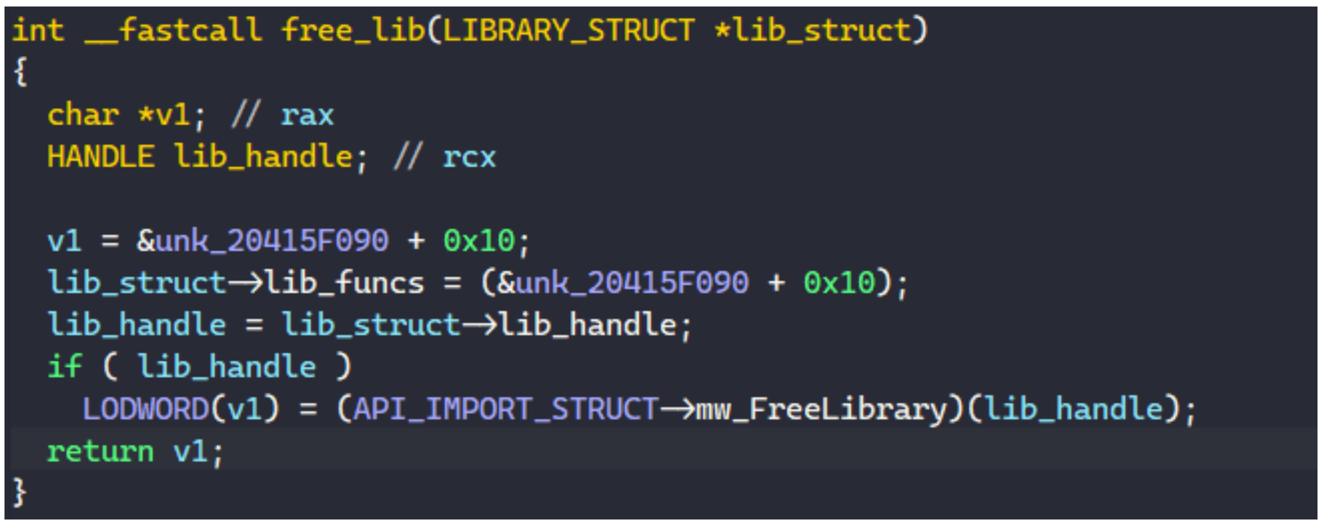
```
GetModuleHandleW = get_proc_addr(output, output->kernel32_handle, stack_str);// "GetModuleHandleW"
v21 = 6;
v14 = 0i64;
output->mw_GetModuleHandleW = GetModuleHandleW;
*&v20 = 0x5E3B6F28114D5D7Di64;
*(&v20 + 1) = 0x6A5D5E6F3A266B5Di64;
v26 = 0;
*stack_str = v20;
v25 = 6;
do
{
  stack_str[v14] = (7 * (stack_str[v14] - 6) % 0x7F + 0x7F) % 0x7F;
  ++v14;
}
while ( v14 != 0x11 );
GetModuleHandleA = get_proc_addr(output, output->kernel32_handle, stack_str);// "GetModuleHandleA"
output->mw_GetModuleHandleA = GetModuleHandleA;
```

Next, for each library to be imported, BAZARLOADER populates the following LIBRARY_STRUCT structure that contains a set of functions to interact with the library and the library handle.

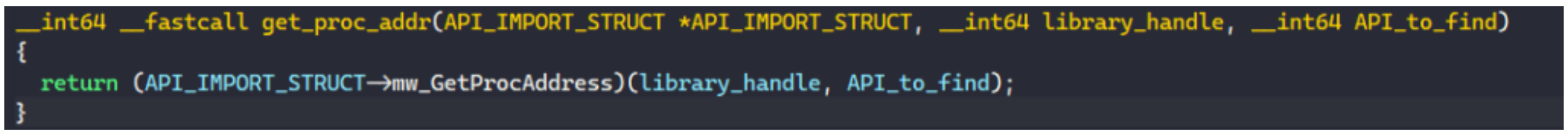
```
struct LIB_FUNCS { FARPROC free_lib; FARPROC w_free_lib; __int64 (__fastcall *get_API_addr)
(API_IMPORT_STRUCT*, HANDLE, char*); }; struct LIBRARY_STRUCT { LIB_FUNCS *lib_funcs; HANDLE lib_handle;
};
```

The first 2 functions in the LIB_FUNCS structure just call the FreeLibrary API from the global API_IMPORT_STRUCT to free the library module.

The third function calls the GetProcAddress from the API_IMPORT_STRUCT's field to retrieve the address of an API exported from that specific library.

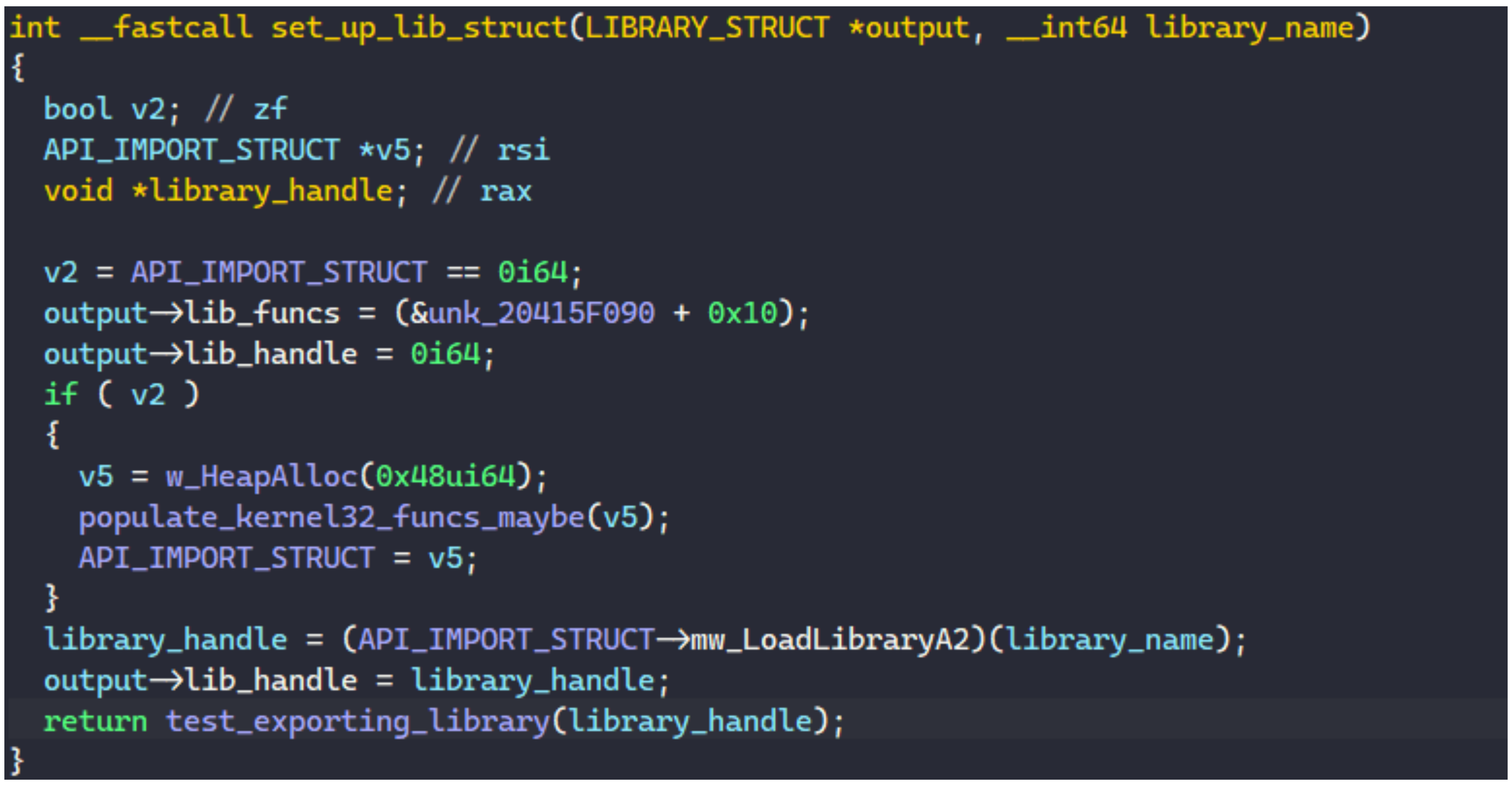


```
int __fastcall free_lib(LIBRARY_STRUCT *lib_struct)
{
  char *v1; // rax
  HANDLE lib_handle; // rcx

  v1 = &unk_20415F090 + 0x10;
  lib_struct->lib_funcs = (&unk_20415F090 + 0x10);
  lib_handle = lib_struct->lib_handle;
  if ( lib_handle )
    LODWORD(v1) = (API_IMPORT_STRUCT->mw_FreeLibrary)(lib_handle);
  return v1;
}
```



```
__int64 __fastcall get_proc_addr(API_IMPORT_STRUCT *API_IMPORT_STRUCT, __int64 library_handle, __int64 API_to_find)
{
  return (API_IMPORT_STRUCT->mw_GetProcAddress)(library_handle, API_to_find);
}
```

To begin populating each LIBRARY_STRUCT structure, BAZARLOADER decodes the library name from a stack string and populates it with the corresponding set of functions and the library handle retrieved from calling LoadLibraryA.



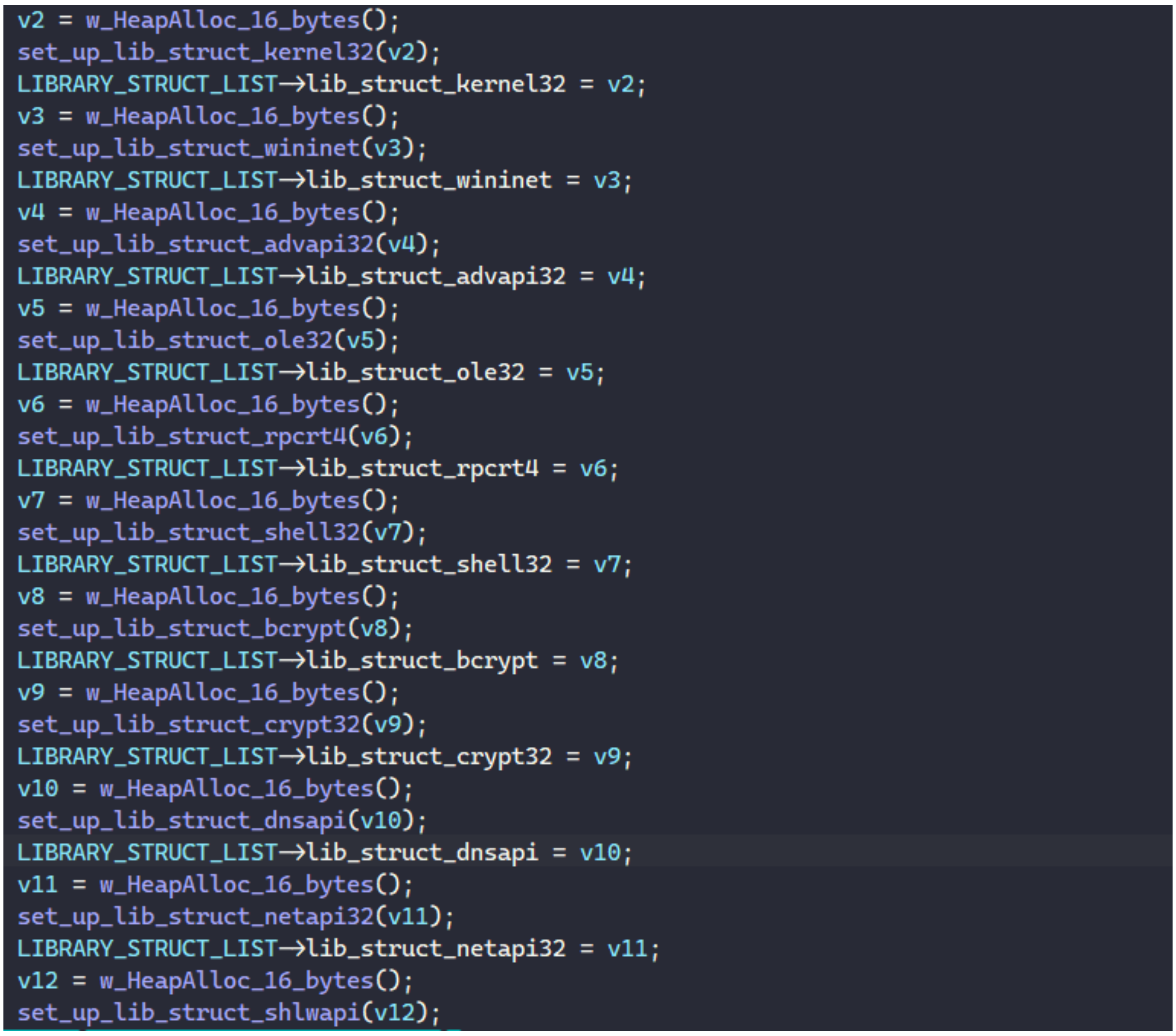
```
int __fastcall set_up_lib_struct(LIBRARY_STRUCT *output, __int64 library_name)
{
  bool v2; // zf
  API_IMPORT_STRUCT *v5; // rsi
  void *library_handle; // rax

  v2 = API_IMPORT_STRUCT == 0i64;
  output->lib_funcs = (&unk_20415F090 + 0x10);
  output->lib_handle = 0i64;
  if ( v2 )
  {
    v5 = w_HeapAlloc(0x48ui64);
    populate_kernel32_funcs_maybe(v5);
    API_IMPORT_STRUCT = v5;
  }
  library_handle = (API_IMPORT_STRUCT->mw_LoadLibraryA2)(library_name);
  output->lib_handle = library_handle;
  return test_exporting_library(library_handle);
}
```

Below is the list of all libraries used by the malware.

`kernel32.dll, wininet.dll, advapi32.dll, ole32.dll, rpcrt4.dll, shell32.dll, bcrypt.dll, crypt32.dll, dnsapi.dll, netapi32.dll, shlwapi.dll, user32.dll, ktmw32.dll`

The LIBRARY_STRUCT structures corresponding to these are pushed into a global list in the order below.

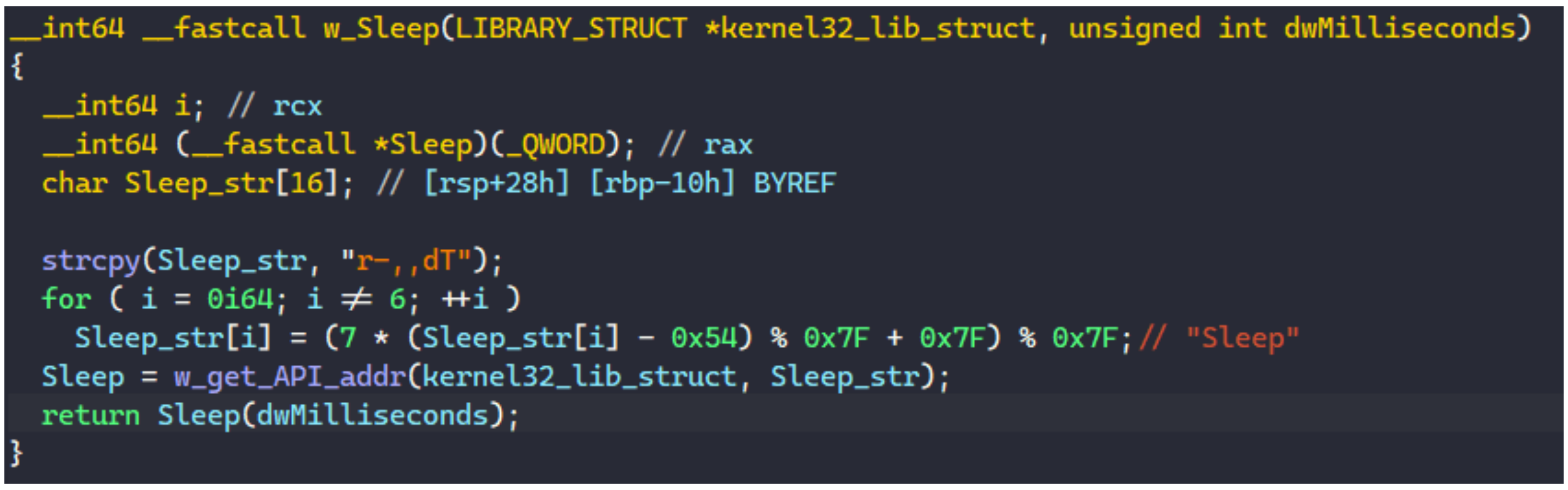
```
struct LIBRARY_STRUCT_LIST { LIBRARY_STRUCT *lib_struct_kernel32; LIBRARY_STRUCT *lib_struct_wininet;
LIBRARY_STRUCT *lib_struct_advapi32; LIBRARY_STRUCT *lib_struct_ole32; LIBRARY_STRUCT *lib_struct_rpcrt4;
LIBRARY_STRUCT *lib_struct_shell32; LIBRARY_STRUCT *lib_struct_bcrypt; LIBRARY_STRUCT *lib_struct_crypt32;
LIBRARY_STRUCT *lib_struct_dnsapi; LIBRARY_STRUCT *lib_struct_netapi32; LIBRARY_STRUCT *lib_struct_shlwapi;
LIBRARY_STRUCT *lib_struct_user32; LIBRARY_STRUCT *lib_struct_ktmw32; };
```



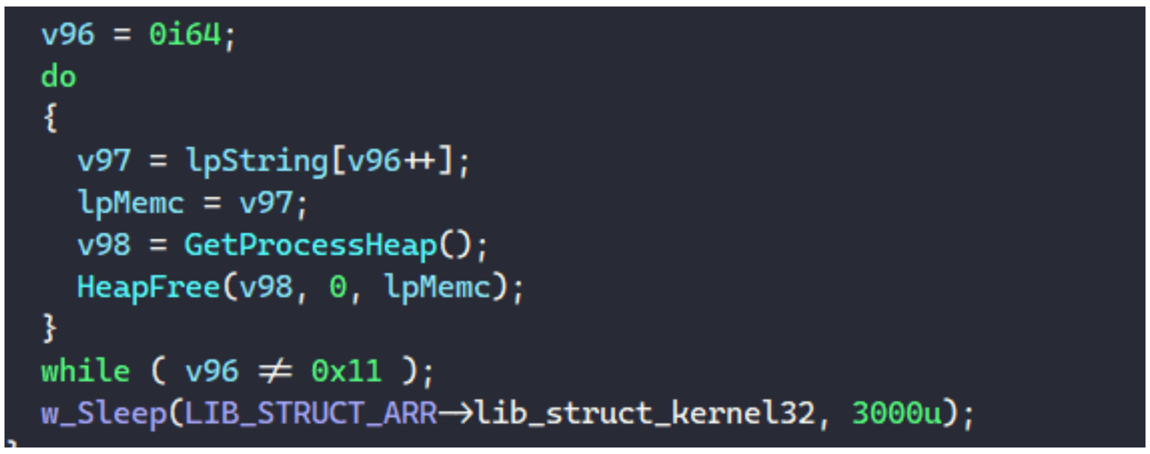
```
v2 = w_HeapAlloc_16_bytes();
set_up_lib_struct_kernel32(v2);
LIBRARY_STRUCT_LIST->lib_struct_kernel32 = v2;
v3 = w_HeapAlloc_16_bytes();
set_up_lib_struct_wininet(v3);
LIBRARY_STRUCT_LIST->lib_struct_wininet = v3;
v4 = w_HeapAlloc_16_bytes();
set_up_lib_struct_advapi32(v4);
LIBRARY_STRUCT_LIST->lib_struct_advapi32 = v4;
v5 = w_HeapAlloc_16_bytes();
set_up_lib_struct_ole32(v5);
LIBRARY_STRUCT_LIST->lib_struct_ole32 = v5;
v6 = w_HeapAlloc_16_bytes();
set_up_lib_struct_rpcrt4(v6);
LIBRARY_STRUCT_LIST->lib_struct_rpcrt4 = v6;
v7 = w_HeapAlloc_16_bytes();
set_up_lib_struct_shell32(v7);
LIBRARY_STRUCT_LIST->lib_struct_shell32 = v7;
v8 = w_HeapAlloc_16_bytes();
set_up_lib_struct_bcrypt(v8);
LIBRARY_STRUCT_LIST->lib_struct_bcrypt = v8;
v9 = w_HeapAlloc_16_bytes();
set_up_lib_struct_crypt32(v9);
LIBRARY_STRUCT_LIST->lib_struct_crypt32 = v9;
v10 = w_HeapAlloc_16_bytes();
set_up_lib_struct_dnsapi(v10);
LIBRARY_STRUCT_LIST->lib_struct_dnsapi = v10;
v11 = w_HeapAlloc_16_bytes();
set_up_lib_struct_netapi32(v11);
LIBRARY_STRUCT_LIST->lib_struct_netapi32 = v11;
v12 = w_HeapAlloc_16_bytes();
set_up_lib_struct_shlwapi(v12);
```

After this global list of LIBRARY_STRUCT is populated, an API can be called from a function taking in its corresponding library's LIBRARY_STRUCT structure and its parameters.

This function resolves the API name from a stack string, retrieves the API's address using the get_API_addr function from the library structure, and calls the API with its parameters.



```
__int64 __fastcall w_Sleep(LIBRARY_STRUCT *kernel32_lib_struct, unsigned int dwMilliseconds)
{
  __int64 i; // rcx
  __int64 (__fastcall *Sleep)(_QWORD); // rax
  char Sleep_str[16]; // [rsp+28h] [rbp-10h] BYREF

  strcpy(Sleep_str, "r-,,dT");
  for ( i = 0i64; i != 6; ++i )
    Sleep_str[i] = (7 * (Sleep_str[i] - 0x54) % 0x7F + 0x7F) % 0x7F;// "Sleep"
  Sleep = w_get_API_addr(kernel32_lib_struct, Sleep_str);
  return Sleep(dwMilliseconds);
}
```



```
v96 = 0i64;
do
{
  v97 = lpString[v96++];
  lpMemc = v97;
  v98 = GetProcessHeap();
  HeapFree(v98, 0, lpMemc);
}
while ( v96 ≠ 0x11 );
w_Sleep(LIB_STRUCT_ARR→lib_struct_kernel32, 3000u);
```

The way the wrapper function is setup to call the actual API is really intuitive, making the code simple to understand through static analysis. However, it’s a bit more difficult to automate the process since there is no API hashing involved.

For my analysis, I just manually decode the stack strings in my debugger and rename the wrapper function accordingly.

At this point, we have fully unpacked BAZARLOADER and understood how the malware obfuscates its strings and APIs to make analysis harder.

In the next blog post, we will fully analyze how the loader downloads and launches a Cobalt Strike beacon from its C2 servers!