

**Topic** :- How malware resolve API addresses dynamically

**Malware hashes** :-

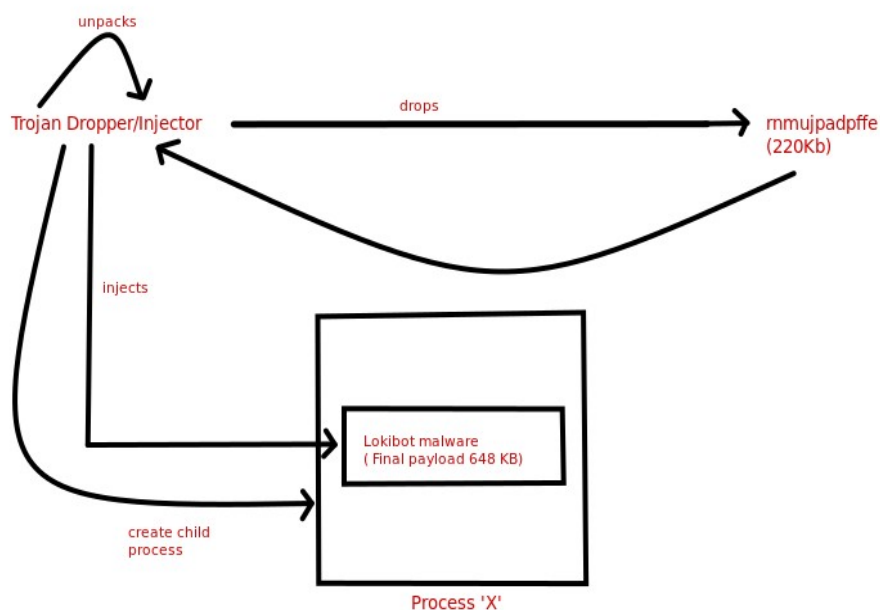
- **sha1** :- 70c34a5e1442816c23d78454edc2c7505f43f82b
- **sha256** :- 563818872af4977ebccd2bc8f97e968edeb6cce444c7a380b3c69e53fd317c2e

**Tools Used** :- Windbg

**Overview** :- Malware often try to hide their intentions and one simple trick is to hide the API's they use which can in most cases reveal their intention. They achieve this using Dynamic API address resolution which we will discuss in this article.

Whenever a malware analyst gets a malware file most of the time the first steps taken is to see the strings and imported API's by the malware. But this is hindered when the malware is packed and sometimes even after unpacking the malware we can see the strings but not always the imported API's. Which then requires the Analyst to resolve to Dynamic Analysis or code reversing of malware code. So, in this Article, I would like to present a clear picture as to how the malware goes about resolving API address dynamically using the Lokibot Malware sample.

So below image shows the injector part of the Loki Malware.



In order to achieve this task it has to use windows API's or system call interface which it can do in many ways. Some of the techniques are.

- It uses kernel32.dll to get the API address it wants.
- It uses the ntdll.dll to get the Syscall ID and uses sysenter to achieve the functionality required.

In this article we will focus on the first. I have already discussed the second technique in depth in a previous article which you can check out [here](#).

So we can see from the above image that it creates a new process from its own binary image. In order to do that it uses the particular windows API, in this case CreateProcessW. So lets see how it gets the address of that API.

It first gets the address of the PEB(Process Environment Block) at line 0x0168099f. Then it goes on to access the PEB\_LDR\_DATA structure field at line 0x016809a5. Then inside PEB\_LDR\_DATA structure it accesses the InLoadOrderModuleList structure. It points to a *Doubly linked* list. This list holds a structure of type LDR\_DATA\_TABLE\_ENTRY.

```
0:000> dt ntdll!_PEB 0x27e000
+0x000 InheritedAddressSpace : 0 ''
+0x001 ReadImageFileExecOptions : 0 ''
+0x002 BeingDebugged : 0 ''
+0x003 BitField : 0 ''
+0x003 ImageUsesLargePages : 0y0
+0x003 IsProtectedProcess : 0y0
+0x003 IsImageDynamicallyRelocated : 0y0
+0x003 SkipPatchingUser32Forwarders : 0y0
+0x003 IsPackagedProcess : 0y0
+0x003 IsAppContainer : 0y0
+0x003 IsProtectedProcessLight : 0y0
+0x003 IsLongPathAwareProcess : 0y0
+0x004 Mutant : 0xffffffff Void
+0x008 ImageBaseAddress : 0x00400000 Void
+0x00c Ldr : 0x77311c60 _PEB_LDR_DATA
+0x010 ProcessParameters : 0x009f18b0 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData : (null)
```

```
Address: @$scopeip
01680995 e00e jmp 01680995
01680985 8bca mov ecx, edx
01680987 d1e8 shr eax, 1
01680989 c1e107 shl ecx, 7
0168098c 46 inc esi
0168098d 0bc8 or ecx, eax
0168098f 03cf add ecx, edi
01680991 03d1 add edx, ecx
01680993 0fbe3e movsx edi, byte ptr [esi]
01680996 8bc2 mov eax, edx
01680998 85ff test edi, edi
0168099a 75e9 jne 01680985
0168099c 5f pop edi
0168099d 5e pop esi
0168099e c3 ret
0168099f 64a130000000 mov eax, dword ptr fs:[00000030h] fs:003b:00000030=0027e0
016809a5 8b400c mov eax, dword ptr [eax+0Ch]
016809a8 8b400c mov eax, dword ptr [eax+0Ch]
016809ab 8b00 mov eax, dword ptr [eax]
016809ad 8b00 mov eax, dword ptr [eax]
016809af 8b4018 mov eax, dword ptr [eax+18h]
016809b2 c3 ret
016809b3 55 push ebp
016809b4 8bec mov ebp, esp
016809b6 56 push esi
016809b7 8bf1 mov esi, ecx
```

Frame Index	Name
[0x0]	0x168099f
[0x1]	0x1680bd0
[0x2]	0x167028a

Name	Value
eax	0x00000000
ebx	0x000015de
ecx	0x00001ef3

```
0x470 LeapSecondData : 0x7ffa0000 _LEAP_SECOND_DATA
+0x474 LeapSecondFlags : 0
+0x474 SixtySecondEnabled : 0y0
+0x474 Reserved : 0y00000000000000000000000000000000 (0)
+0x478 NtGlobalFlag2 : 0
0:000> dt ntdll!_PEB_LDR_DATA 0x77311c60
+0x000 Length : 0x30
+0x004 Initialized : 0x1 ''
+0x008 SsHandle : (null)
+0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x9f20f0 - 0xb090f8 ]
+0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x9f20f8 - 0xb09100 ]
+0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x9f1ff8 - 0xb09108 ]
+0x024 EntryInProgress : (null)
+0x028 ShutdownInProgress : 0 ''
+0x02c ShutdownThreadId : (null)
```

```
01680985 e00e jmp 01680995
01680985 8bca mov ecx, edx
01680987 d1e8 shr eax, 1
01680989 c1e107 shl ecx, 7
0168098c 46 inc esi
0168098d 0bc8 or ecx, eax
0168098f 03cf add ecx, edi
01680991 03d1 add edx, ecx
01680993 0fbe3e movsx edi, byte ptr [esi]
01680996 8bc2 mov eax, edx
01680998 85ff test edi, edi
0168099a 75e9 jne 01680985
0168099c 5f pop edi
0168099d 5e pop esi
0168099e c3 ret
0168099f 64a130000000 mov eax, dword ptr fs:[00000030h] fs
016809a5 8b400c mov eax, dword ptr [eax+0Ch]
016809a8 8b400c mov eax, dword ptr [eax+0Ch]
016809ab 8b00 mov eax, dword ptr [eax]
016809ad 8b00 mov eax, dword ptr [eax]
016809af 8b4018 mov eax, dword ptr [eax+18h]
016809b2 c3 ret
016809b3 55 push ebp
016809b4 8bec mov ebp, esp
016809b6 56 push esi
016809b7 8bf1 mov esi, ecx
```

Name	Value
eax	0x00000000
ebx	0x000015de
ecx	0x00001ef3

The LDR\_DATA\_TABLE\_ENTRY structure contains various field as shown below. At offset 0x18 it contains the base of the dll, which is accessed at line 0x016809af.

The screenshot shows a debugger window with the following components:

- Command Window:**

```

0:000> dt ntdll!_LIST_ENTRY 0x9f20f0
[ 0x9f1fe8 - 0x77311c6c ]
+0x000 Flink : 0x009f1fe8 _LIST_ENTRY [ 0x9f24d0 - 0x9f20f0 ]
+0x004 Blink : 0x77311c6c _LIST_ENTRY [ 0x9f20f0 - 0xb090f8 ]
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY 0x9f1fe8
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x9f24d0 - 0x9f20f0 ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x9f24d8 - 0x9f20f8 ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x9f28b0 - 0x77311c7c ]
+0x018 DllBase : 0x771f0000 Void
+0x01c EntryPoint : (null)
+0x020 SizeOfImage : 0x19e000
+0x024 FullDllName : _UNICODE_STRING "C:\Windows\SYSTEM32\ntdll.dll"
+0x02c BaseDllName : _UNICODE_STRING "ntdll.dll"
+0x034 FlagGroup : [4] "???"
+0x034 Flags : 0xaac4
+0x034 PackagedBinary : 0y0
+0x034 MarkedForRemoval : 0y0
+0x034 ImageDll : 0y1

```
- Registers Window:**

Name	Value
User	
eax	0x00000000
- Assembly Window:**

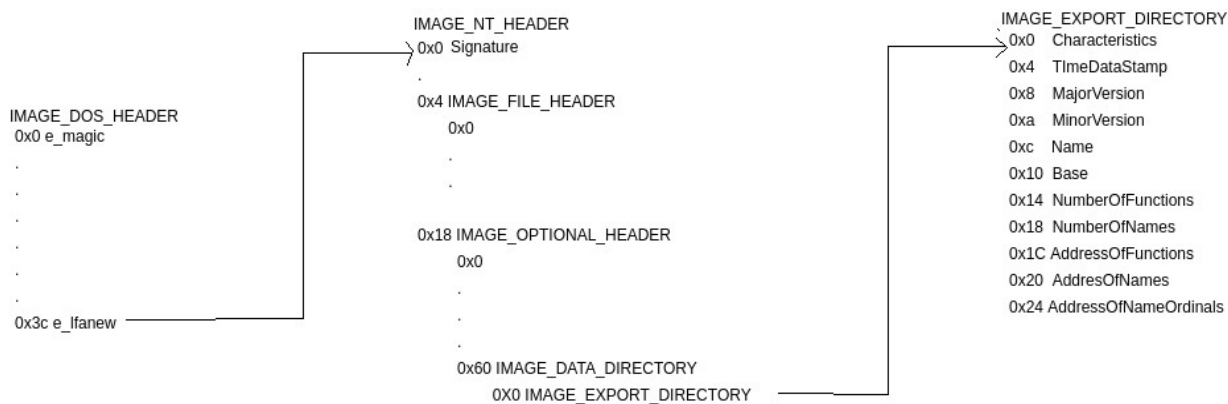
```

Address: 0x016809af
01680983 jmp 01680993
01680985 mov ecx, edx
01680987 shr eax, 1
01680989 shl ecx, 7
0168098c inc esi
0168098d or ecx, eax
0168098f add ecx, edi
01680991 add edx, ecx
01680993 movsx edi, byte ptr [esi]
01680996 mov eax, edx
01680998 test edi, edi
0168099a jne 01680985
0168099c pop edi
0168099d pop esi
0168099e ret
0168099f mov eax, dword ptr fs:[0000030h]
016809a5 mov eax, dword ptr [eax+0Ch]
016809a8 mov eax, dword ptr [eax+0Ch]
016809ab mov eax, dword ptr [eax]
016809ad mov eax, dword ptr [eax]
016809af mov eax, dword ptr [eax+18h]
016809b2 ret
016809b3 push ebp
016809b4 mov ebp, esp

```

An arrow points from the **DllBase** field in the structure to the instruction at address 0x016809af.

After getting the base address of the kernel32.dll, the malware traverses the dll to get the function address. In order to understand that we need to know the layout of the PE(Portable Executable) file format. Below is a high level overview of the PE format.



Below we can see how the malware implements the functionality of getting function address.

```

00> t
=76600000 ebx=000015de ecx=76600000 edx=ff7f721a esi=00000000 edi=76600000
=03150a5e esp=0099e8a8 ebp=0099e8c4 iopl=0         nv up ei pl zr na pe nc
001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
50a5e 8b473c      mov     eax,dword ptr [edi+3Ch] ds:0023:7660003c=000000f8

Registers
Name      Value
-----
eax       0x76600000
ebx       0x000015de
ecx       0x76600000
edx       0xff7f721a
esi       0x00000000
edi       0x76600000
esp       0x0099e8a8
ebp       0x0099e8c4
eip       0x03150a5e

03150a57 8bf9      mov     edi, ecx
03150a59 8955fc    mov     dword ptr [ebp-4], edx
03150a5c 33f6      xor     esi, esi
03150a5e 8b473c    mov     eax, dword ptr [edi+3Ch] ds:0023:7660003c=000000f8
03150a61 8b443878  mov     eax, dword ptr [eax+edi+78h]
03150a65 03c7      add     eax, edi
03150a67 8b5020    mov     edx, dword ptr [eax+20h]
03150a6a 8b581c    mov     ebx, dword ptr [eax+1Ch]
03150a6d 03d7      add     edx, edi
03150a6f 8b4824    mov     ecx, dword ptr [eax+24h]
03150a72 03df      add     ebx, edi
03150a74 8b4018    mov     eax, dword ptr [eax+18h]
03150a77 03cf      add     ecx, edi
03150a79 8955f4    mov     dword ptr [ebp-0Ch], edx
03150a7c 894df0    mov     dword ptr [ebp-10h], ecx
03150a7f 8945f8    mov     dword ptr [ebp-8], eax
03150a82 85c0      test    eax, eax
03150a84 7418      je      03150a9e
03150a86 8b0cb2    mov     ecx, dword ptr [edx+esi*4]
03150a89 03cf      add     ecx, edi

```

It first accesses the `e_lfanew` field of `IMAGE_DOS_HEADER` which points to `IMAGE_NT_HEADER`. Then at line `0x3150a61` it access the `IMAGE_EXPORT_DIRECTORY` Field. After getting the base of `IMAGE_EXPORT_DIRECTORY` structure, it access the fields in the Structure as shown above in the image.

- `IMAGE_EXPORT_DIRECTORY + 0x20 = AddressOfNames`
- `IMAGE_EXPORT_DIRECTORY + 0x1C = AddressOfFunctions`
- `IMAGE_EXPORT_DIRECTORY + 0x24 = AddressOfNameOrdinals`
- `IMAGE_EXPORT_DIRECTORY + 0x18 = NumberOfNames`

The `AddressOfNames` field points to an array of pointers, where each address points to the name of a particular function.

```

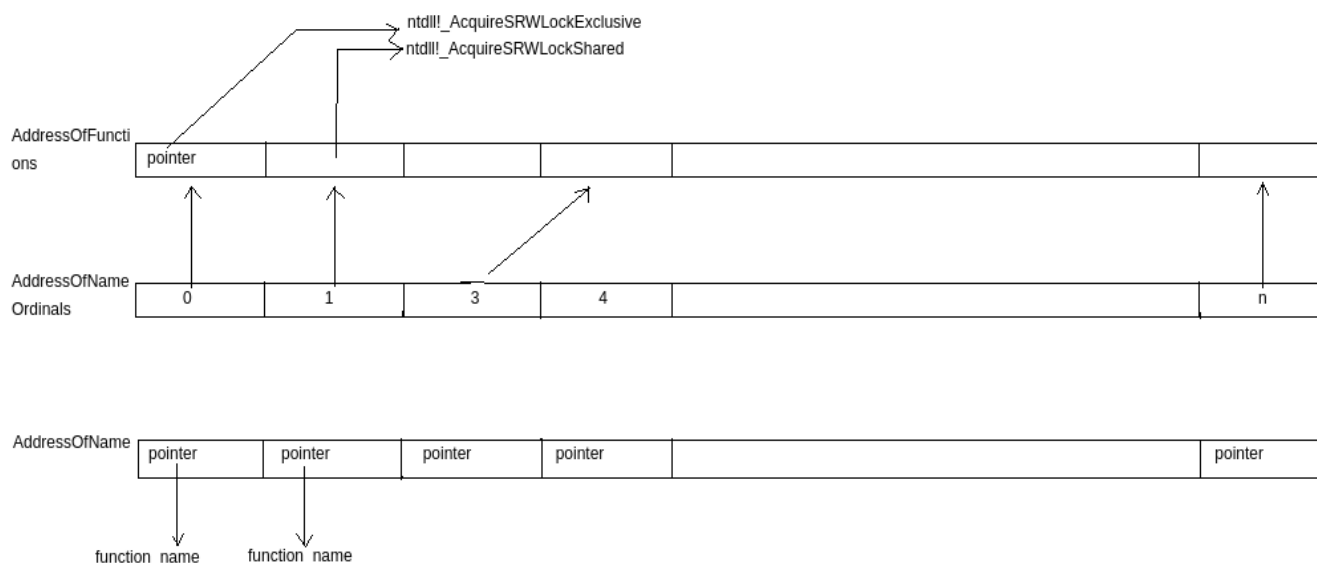
03150a72 03df      add     ebx, edi
0:000> dt combase!_IMAGE_EXPORT_DIRECTORY 0x76679590
+0x000 Characteristics : 0
+0x004 TimeDateStamp   : 0xcdcf12195
+0x008 MajorVersion    : 0
+0x00a MinorVersion    : 0
+0x00c Name            : 0x7d474
+0x010 Base            : 1
+0x014 NumberOfFunctions : 0x646
+0x018 NumberOfNames   : 0x646
+0x01c AddressOfFunctions : 0x795b8
+0x020 AddressOfNames   : 0x7aed0
+0x024 AddressOfNameOrdinals : 0x7c7e8
0:000> dd 7667aed0
7667aed0 0007d4d0 0007d509 0007d53c 0007d54b
7667aee0 0007d560 0007d569 0007d572 0007d583
7667aef0 0007d594 0007d5d9 0007d5ff 0007d61e
7667af00 0007d63d 0007d64a 0007d65d 0007d675
7667af10 0007d690 0007d6a5 0007d6c2 0007d701
7667af20 0007d742 0007d755 0007d762 0007d77c
7667af30 0007d79a 0007d7d1 0007d816 0007d861
7667af40 0007d8bc 0007d911 0007d964 0007d9b9

Address: @scopeip
03150a4c 50      pop     eax
03150a4d c3      ret
03150a4e 55      push    ebp
03150a4f 8bec    mov     ebp, esp
03150a51 83ec10  sub     esp, 10h
03150a54 53      push    ebx
03150a55 56      push    esi
03150a56 57      push    edi
03150a57 8bf9    mov     edi, ecx
03150a59 8955fc  mov     dword ptr [ebp-4], edx
03150a5c 33f6    xor     esi, esi
03150a5e 8b473c  mov     eax, dword ptr [edi+3Ch]
03150a61 8b443878  mov     eax, dword ptr [eax+edi+78h]
03150a65 03c7    add     eax, edi
03150a67 8b5020  mov     edx, dword ptr [eax+20h]
03150a6a 8b581c  mov     ebx, dword ptr [eax+1Ch]
03150a6d 03d7    add     edx, edi
03150a6f 8b4824  mov     ecx, dword ptr [eax+24h]
03150a72 03df    add     ebx, edi
03150a74 8b4018  mov     eax, dword ptr [eax+18h]
03150a77 03cf    add     ecx, edi
03150a79 8955f4  mov     dword ptr [ebp-0Ch], edx
03150a7c 894df0  mov     dword ptr [ebp-10h], ecx

```

+0x020	AddressOfNames	:	0x7aed0		03150a4c	50	pop	eax
+0x024	AddressOfNameOrdinals	:	0x7c7e8		03150a4d	c3	ret	
0:000>	dd	7667aed0			03150a4e	55	push	ebp
7667aed0	0007d4d0	0007d509	0007d53c	0007d54b	03150a4f	8bec	mov	ebp, esp
7667aee0	0007d560	0007d569	0007d572	0007d583	03150a51	83ec10	sub	esp, 10h
7667aef0	0007d594	0007d5d9	0007d5ff	0007d61e	03150a54	53	push	ebx
7667af00	0007d63d	0007d64a	0007d65d	0007d675	03150a55	56	push	esi
7667af10	0007d690	0007d6a5	0007d6c2	0007d701	03150a56	57	push	edi
7667af20	0007d742	0007d755	0007d762	0007d77c	03150a57	8bf9	mov	edi, ecx
7667af30	0007d79a	0007d7d1	0007d816	0007d861	03150a59	8955fc	mov	dword ptr [ebp-4], edx
7667af40	0007d8bc	0007d911	0007d964	0007d9b9	03150a5c	33f6	xor	esi, esi
0:000>	dc	7667d4d0			03150a5e	8b473c	mov	eax, dword ptr [edi+3Ch]
7667d4d0	75716341	53657269	6f4c5752	78456b63	03150a61	8b443878	mov	eax, dword ptr [eax+edi+78h]
7667d4e0	73756c63	00657669	4c44544e	74522e4c	03150a65	03c7	add	eax, edi
7667d4f0	7163416c	65726975	4c575253	456b636f	03150a67	8b5020	mov	edx, dword ptr [eax+20h]
7667d500	756c6378	65766973	71634100	65726975	03150a6a	8b581c	mov	ebx, dword ptr [eax+1Ch]
7667d510	4c575253	536b636f	65726168	544e0064	03150a6d	03d7	add	edx, edi
7667d520	2e4c4c44	416c7452	69757163	52536572	03150a6f	8b4824	mov	ecx, dword ptr [eax+24h]
7667d530	636f4c57	6168536b	00646572	69746341	03150a72	03df	add	ebx, edi
7667d540	65746176	43746341	41007874	76697463	03150a74	8b4018	mov	eax, dword ptr [eax+18h]
					03150a77	03cf	add	ecx, edi
					03150a79	8955f4	mov	dword ptr [ebp-0Ch], edx
					03150a7c	894df0	mov	dword ptr [ebp-10h], ecx

The malware tries to iterate through all the names of APIs to find the matching API it requires. Sometimes it will directly match the API name with a predefined string and some times it calculate's a Hash of the API name and matche's that with a predefined Hash . The offset at which the function name is found, the same offset is used to access the AddressOfNameOrdinals array. The AddressOfNamesOrdinals field also points to an array of number. These number are the mapping from AddressOfNames to AddressOfFunctions array.



The Address obtained at the end is used to call the particular function.

The screenshot displays a debugger interface with three main panels: Command, Disassembly, and Registers/Stack.

**Command Panel:** Shows assembly instructions at addresses 76170001 to 7617000c. The instruction at 7617000c is `CALL EBX`, which is highlighted with a red background. An arrow points from this instruction to the `CALL` instruction in the Disassembly panel.

**Disassembly Panel:** Shows assembly instructions starting at address 01681120. The instruction at 01681158 is `CALL EBX`, which is highlighted with a red background. An arrow points from this instruction to the `CALL` instruction in the Registers panel.

**Registers Panel:** Shows the current values of registers. The `eax` register is highlighted with a red background and contains the value `0x76197160`. An arrow points from this value to the `CALL` instruction in the Disassembly panel.

**Stack Panel:** Shows the current stack frame. The stack pointer `esp` is at address `0x76197160`. The instruction at `0x76197160` is `CALL EBX`, which is highlighted with a red background.

**Conclusion** :- Dynamic address resolution of APIs is used by almost all evasive malware. It hinders the static analysis of malware and can be used in creative ways to even avoid API Logging.