Dynamic Imports and Working Around Indirect Calls - Smokeloader Study Case

m.alvar.es/2019/10/dynamic-imports-and-working-around.html

When reversing malware it is common to find an injected payload loading references to external resources (DLL functions). This happens for two main reasons:

1. The hosting process does not have all resources necessary to the execution of the injected payload;
2. Making reversing engineering the malware trickier since the dumped segment will have all calls pointing to a meaningless address table.

This article explains how to revert this trick and get back *API* call names annotations in an *IDApro* database. A sample of Smokeloader was used for illustrating the ideas described in this post.

This article is divided in three main parts:

1. Explaining the observed technique;
2. How it works; and
3. How to circumventing it in order to facilitate reversing.

First of all, shout out to *Sergei Frankoff* from Open Analysis for this amazing video tutorial on this same topic which inspired me to write about my analyses. Regards also to Mark Lim who also wrote a very interesting article about labelling indirect calls in *2018*. His article uses *structures* instead of patching the code (which is also a good approach) but I think it lacks important details and I will try to cover these points in here.

Examples presented in this article were extracted from the following *Smokeloader* sample:

Filename: p0n36i2d.exe
MD5: a8cc396b6f5568e94f28ca3381c7f9df
SHA1: 12948e36584e1677e80f78b8cc5c20576024c13f
SHA256: 17b548f9c8077f8ba66b70d55c383f87f92676520e2749850e555abb4d5f80a5
Size: 215.5 KB (220672 bytes)
Type: PE32 executable for MS Windows (GUI) Intel 80386 32-bit

Explaining what is going on in the first stage (*packer*/*crypter*) is out of scope; this article focuses on characteristics found in the final payload. This sample injects the main payload in "*explorer.exe*" as it is possible to observe in this AnyRun sandbox analysis.

Figure 01 shows how the code looks immediately after the execution control passes to the injected code.

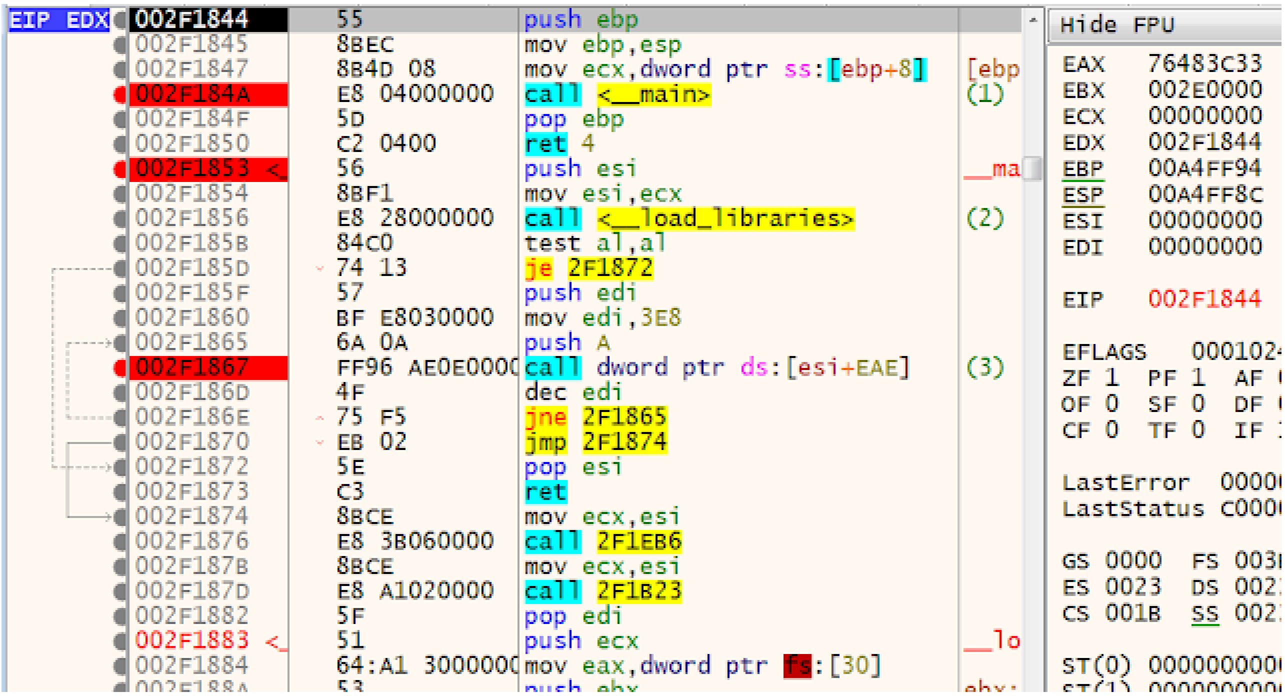


Figure 01 - Smokeloader's final payload.

Three points were marked in this code snip (**1**, **2** and **3**). The first point **(1)** is the call to the main function (located at *0x002F1853*). This function expects to receive an address through *ECX* register. This address points to a data segment where all temporary structures will be stored.

The third point **(3)** is an indirect call to an address stored in register *ESI* plus offset *0xEAE*. The debugger was not able to resolve this address since the "memory segment" pointed by *ESI* is not set at this point of the execution (*Instruction Pointer* pointing to *0x002F1844*). **This pattern usually is an indicator that this code will dynamically resolve and import external resources to a specific address table** (in this case stored in what we called "*data segment*"). This is an interesting technique because this table can be moved around by changing the address stored in *ESI* as long as offsets are preserved. In this code *ESI* is set to "*0x002E0000*" which is the address of a *read-and-write* memory segment created during the first stage. *Figure 02* shows the region pointed by the offset *0xEAE* which is *empty* at this point of the execution.

Figure 02 - Address pointed by the indirect call.

The second point **(2)** marks a function call immediately before the indirect call **(3)**. This is a strong indicator that the code for creating the address table must be somewhere inside this function. The address located in "*002E0EAE"* will be filled with pointers to the expected API function. *Figure 3* shows this same memory region after the "*__load_libraries*" function is executed.

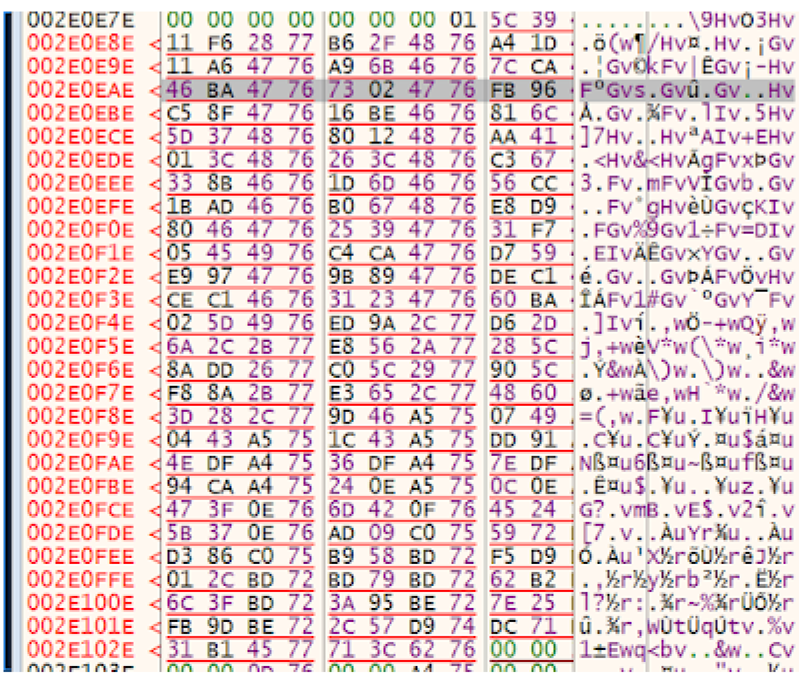
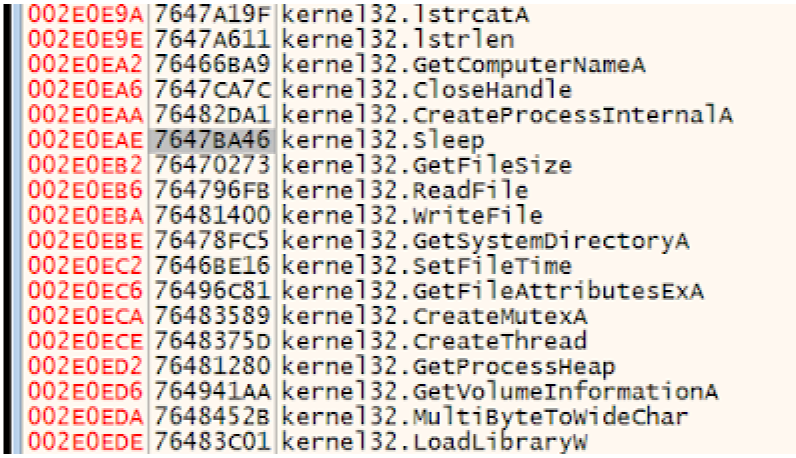


Figure 03 - Address pointed by the indirect call is filled after the "__load_libraries" function is called

x32dbg has a memory dump visualisation mode called "A*ddress*" which will list every function pointed to each address loaded in the call table we just described.



002E0E9A	7647A19F	kernel32.lstrcatA
002E0E9E	7647A611	kernel32.lstrlen
002E0EA2	76466BA9	kernel32.GetComputerNameA
002E0EA6	7647CA7C	kernel32.CloseHandle
002E0EAA	76482DA1	kernel32.CreateProcessInternalA
002E0EAE	7647BA46	kernel32.Sleep
002E0EB2	76470273	kernel32.GetFileSize
002E0EB6	764796FB	kernel32.ReadFile
002E0EBA	76481400	kernel32.WriteFile
002E0EBE	76478FC5	kernel32.GetSystemDirectoryA
002E0EC2	7646BE16	kernel32.SetFileTime
002E0EC6	76496C81	kernel32.GetFileAttributesExA
002E0ECA	76483589	kernel32.CreateMutexA
002E0ECE	7648375D	kernel32.CreateThread
002E0ED2	76481280	kernel32.GetProcessHeap
002E0ED6	764941AA	kernel32.GetVolumeInformationA
002E0EDA	7648452B	kernel32.MultiByteToWideChar
002E0EDE	76483C01	kernel32.LoadLibraryW

Figure 04 - Resolved address in call table

Figure 04 shows that the position pointed by the indirected call listed in point (3) points to function "*sleep*" inside "*kernel32.dll*". Basically this call table is an *Array* of unsigned integers (*4 bytes*) containing an address pointing to an API call in each position.

The "*__load_library*" function is responsible for creating this "*call table*" so the focus of this article will move to understand how it works.

--- ***End of part I*** ---

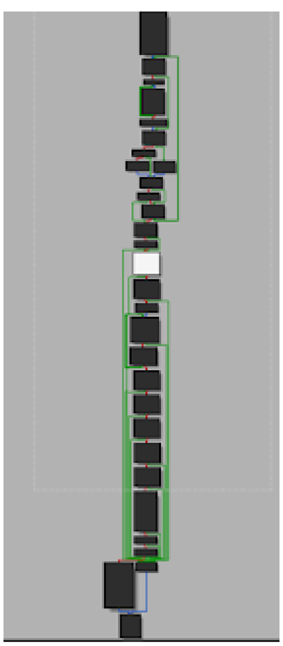
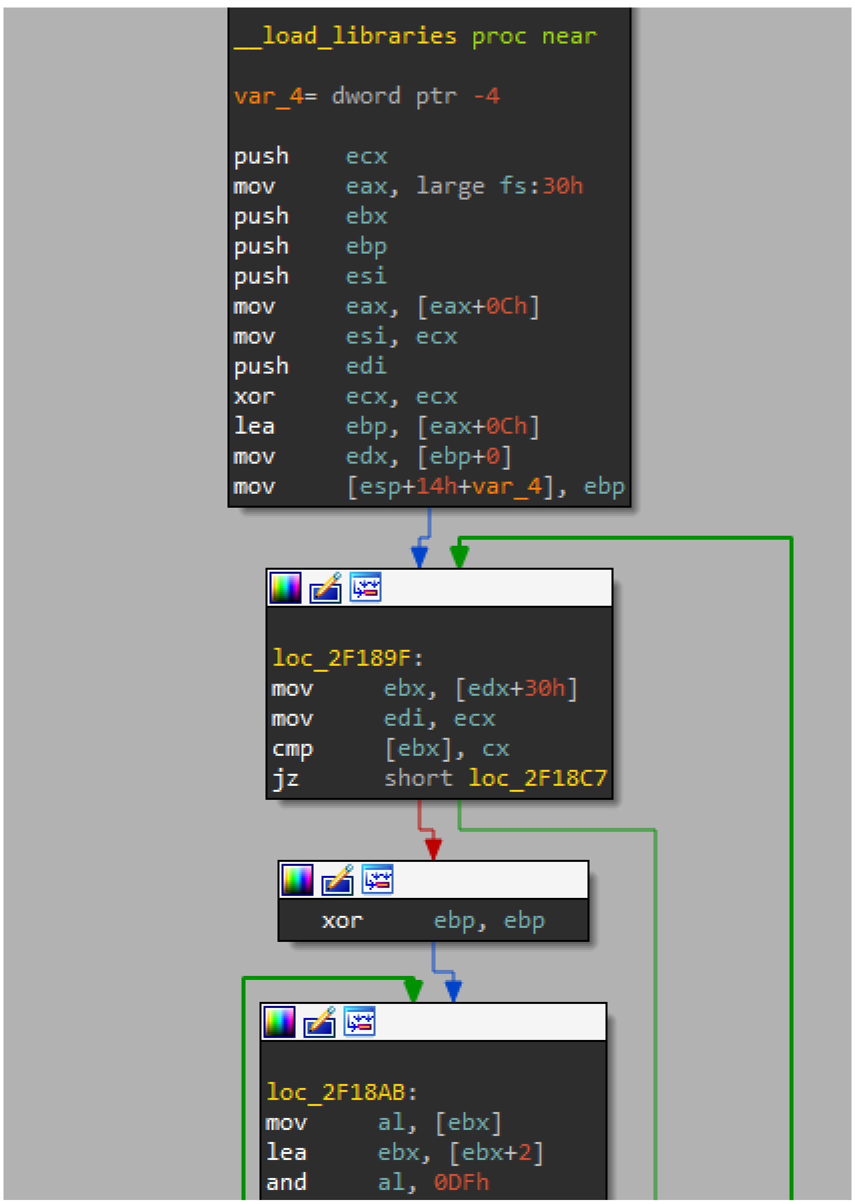


Figure 05 - "*__load_libraries*" zoomed out CFG representation.

Figure 05 shows an overview of the "*__load_library*" function created by *IDA*. This function is quite large and performs few connected steps which we need to go through in order to fully understanding its behaviour. This function can be divided in three main sections:

1. Code responsible for finding the base addresses for core libraries;
2. Code responsible for loading addresses for calls within code libraries;
3. The last section is responsible for loading other libraries necessary for executing the malware.

Figure 06 presents the first part of the "*__load_libraries*" function. In its preamble the code navigates through the *TEB* (*Thread Environment Block*) and loads *4 bytes* from offset *0x30* into register *EAX*. This address contains the address of the *PEB* (*Process Environment Block*). Next step is to get the location for the "*PEB_LDR_DATA*" structure which is located in offset *0xC*. This structure contains a linked list containing information about all modules (*DLLs*) loaded by a specific process.



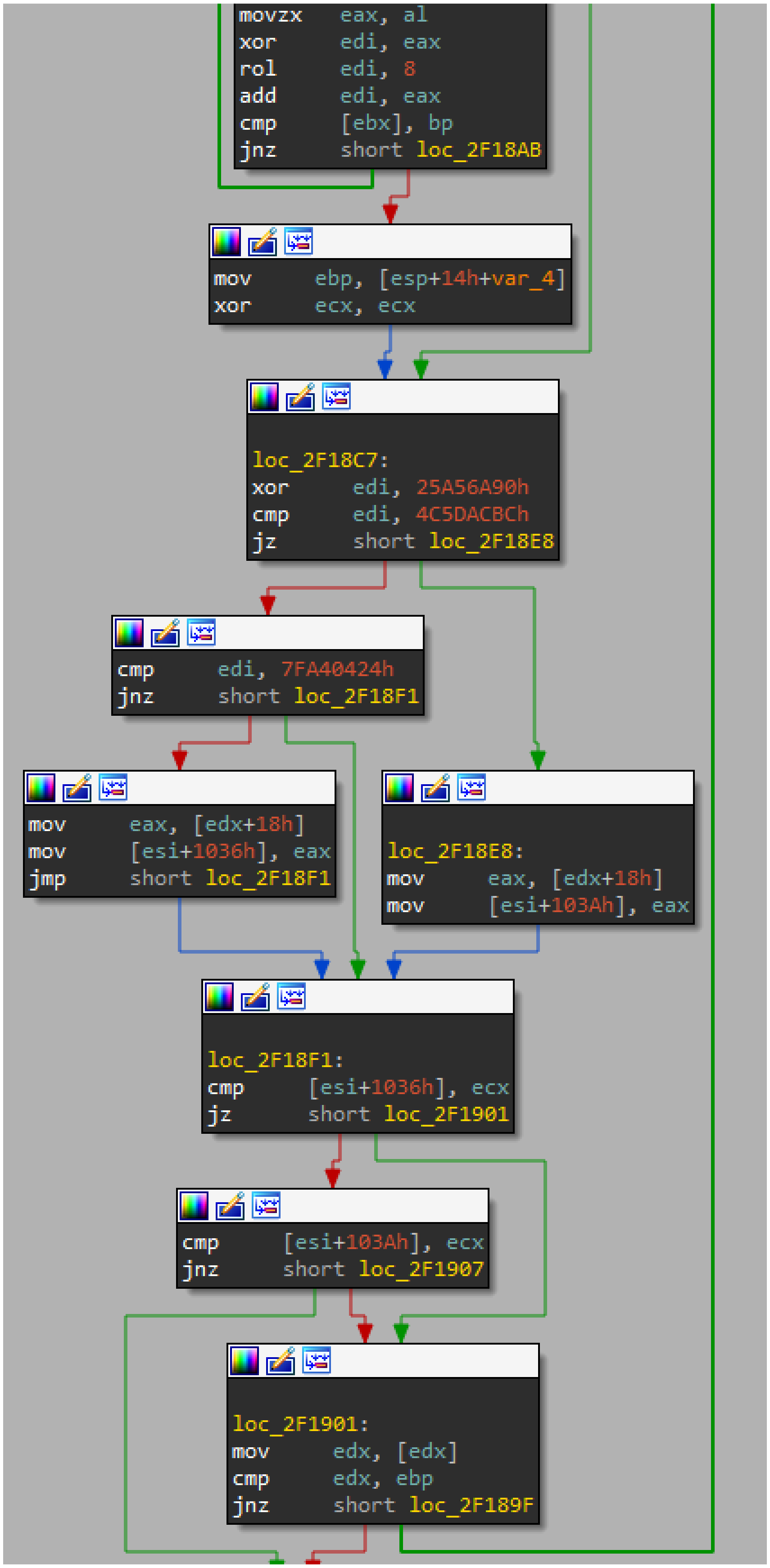


Figure 06 - first section of the "__load_libraries" function.

The code accesses the offset *0xC* in the "*PEB_LDR_DATA*" structure which contains the head element for the loaded modules in the order they were loaded by the process. Each element in this linked list is a combination of "*_LDR_DATA_TABLE_ENTRY*" and "*_LIST_ENTRY*" structures. This structure has an entry to the base name of the module in the offset *0x30*. Figure 07 summarises all this "structure maze" used in order to fetch loaded module names (e*xcusez-moi* for my paint brush skills :D).

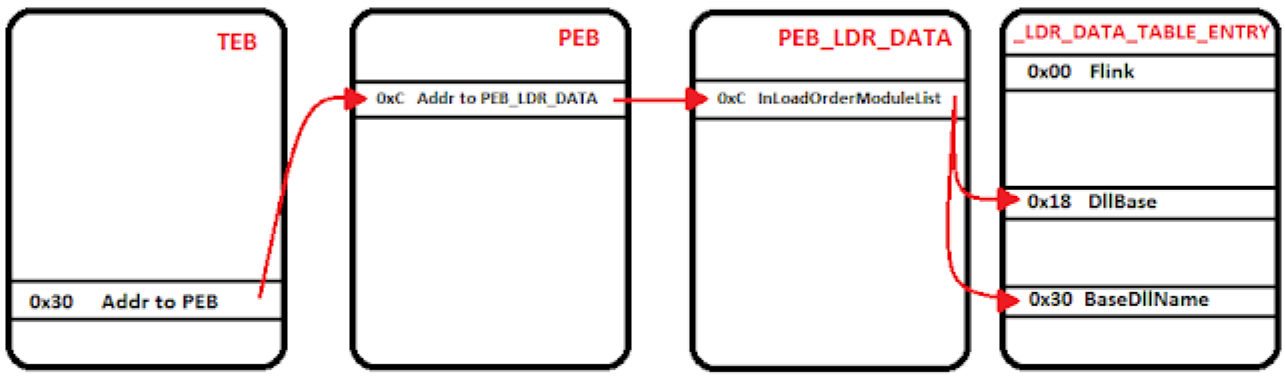


Figure 07 - Path through the process internal structures to get loaded DLL names and base addresses

The main loop, beginning at "*loc_2F189F*" (*Figure 06*), goes through all modules loaded by the "*explorer.exe"* process. This algorithm fetches the module name and calculates a hash out of it. The second smaller looping located at "*loc_2F18AB*" (*Figure 06*) is the part of the code responsible for calculating this hash. *Figure 08* shows the reversed code for this hashing algorithm.

```python
# al, [ebx]
# lea      ebx, [ebx+2]
# and      al, 0DFh
# movzx    eax, al
# xor      edi, eax
# rol      edi, 8
# add      edi, eax
# cmp      [ebx], bp

module_name = b'tinype.exe' # EDI = 611AC587

rol = lambda val, r_bits, max_bits=32: \
    (val << r_bits%max_bits) & (2**max_bits-1) | \
    ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))

final = 0
for c in module_name:
    t1 = (c & 0xDF) & 0xff
    final = final ^ t1
    final = rol(final, 8)
    final += t1
    final &= 0xffffffff

print(hex(final))
```

Figure 08 - Reversed hashing algorithm used in the first part of the analysed code

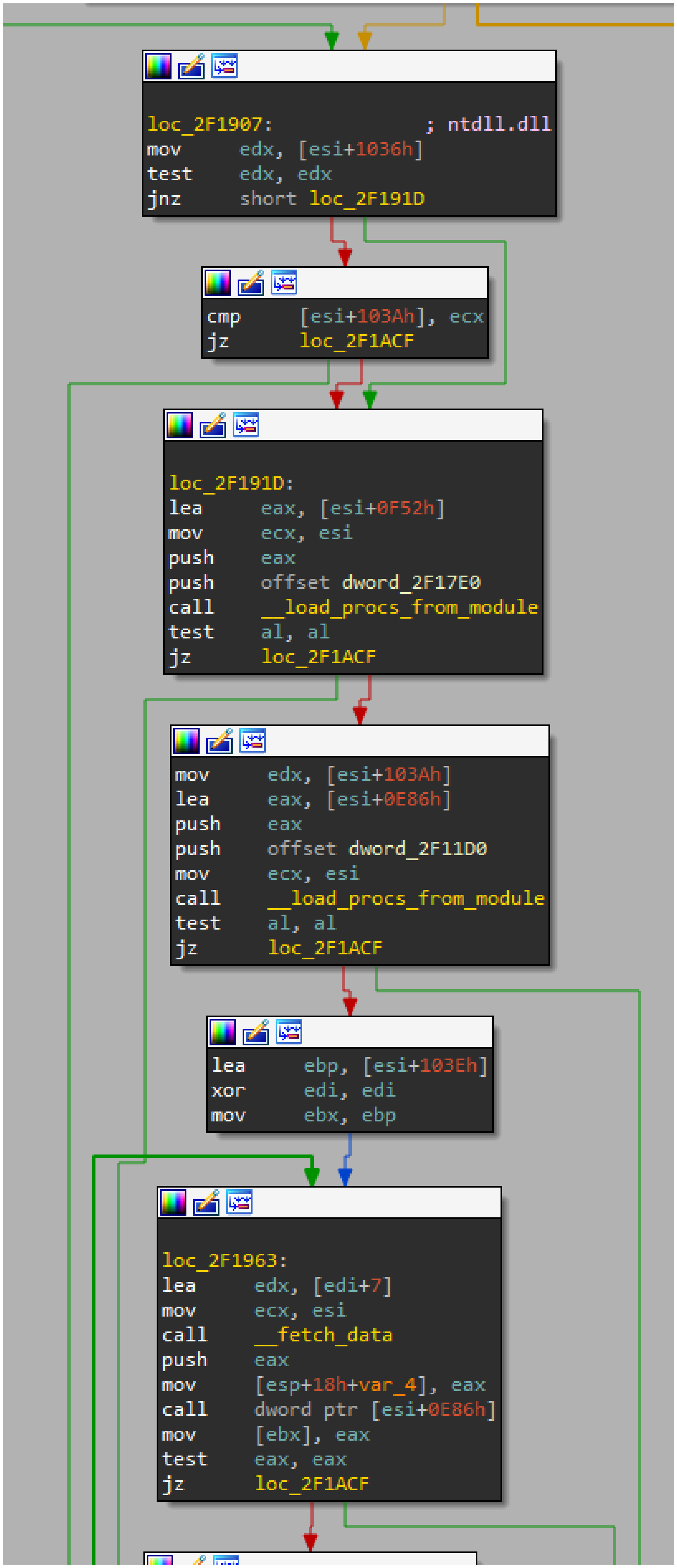
Moving forward, after calculating a *hash* the algorithm does a XOR operation with a hardcoded value *0x25A56A90* and this value is compared with two hardcoded hashes: *0x4C5DACBC* (***kernel32.dll***) and *0x7FA40424 (**ntdll.dll**)*. The base addresses of each *DLL* are stored in two global variables located in the following addresses *[ESI+0x1036]* and *[ESI+0x103A]*.

Bonus: these hardcoded hashes can be used for detecting this specific version of *Smokeloader*.

Summarising, this first part of the code is responsible by finding the base address of two core *libraries* in MS Windows ("*ntdll.dll*" and "*kernel32.dll*"). These addresses will be used for fetching resources necessary for loading all other libraries required by the malware.

Figure 09 shows the second section of "*__load_libraries*". This figure shows the code with some functions names already figured out in order to make it more didactic.

```
jnz     short loc_2F189F
```


loc_2F1907: ; ntdll.dll
mov edx, [esi+1036h]
test edx, edx
jnz short loc_2F191D
cmp [esi+103Ah], ecx
jz loc_2F1ACF
loc_2F191D:
lea eax, [esi+0F52h]
mov ecx, esi
push eax
push offset dword_2F17E0
call __load_procs_from_module
test al, al
jz loc_2F1ACF
mov edx, [esi+103Ah]
lea eax, [esi+0E86h]
push eax
push offset dword_2F11D0
mov ecx, esi
call __load_procs_from_module
test al, al
jz loc_2F1ACF
lea ebp, [esi+103Eh]
xor edi, edi
mov ebx, ebp
loc_2F1963:
lea edx, [edi+7]
mov ecx, esi
call __fetch_data
push eax
mov [esp+18h+var_4], eax
call dword ptr [esi+0E86h]
mov [ebx], eax
test eax, eax
jz loc_2F1ACF

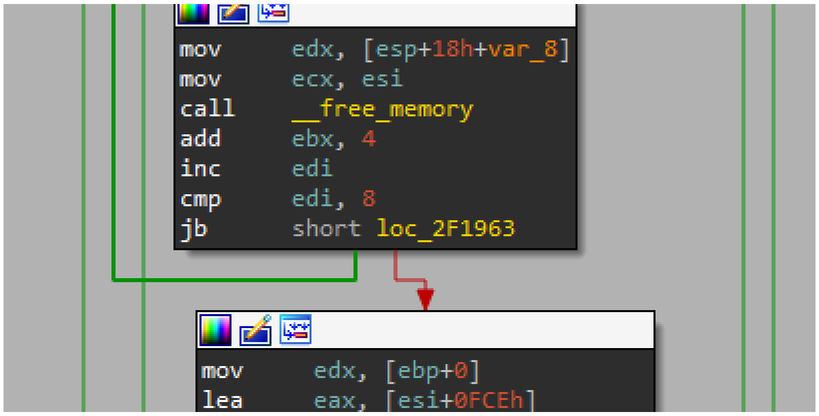


Figure 09 - second section of the "__load_libraries" function.

The first two basic blocks checks if the function was able to find "*ntdll.dll*" and "*kernel32.dll*" base addresses. If these modules are available then the "*__load_procs_from_module*" function is invoked for filling the call table. This function receives *4* parameters and does not follow the standard C calling convention. Two parameters are passed through the stack and the other two through registers (*ECX* and *EDX*). This function expects a *DLL* base address in *EDX*, the data segment in *ECX*, an address to a list of *unsigned ints* (api calls hashes) and a destination address (where the calls addresses will be stored). The last two parameters are pushed in the stack.

Figure 10 shows the *hardcoded* hashes passed as parameter to "*__load_procs_from_module*" function. This list will be used to determine which procedures will be loaded in the call table.

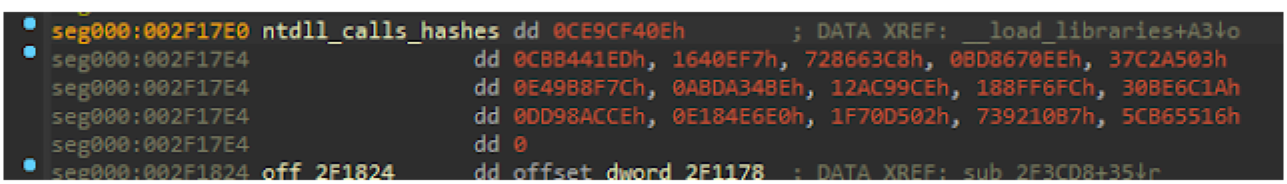
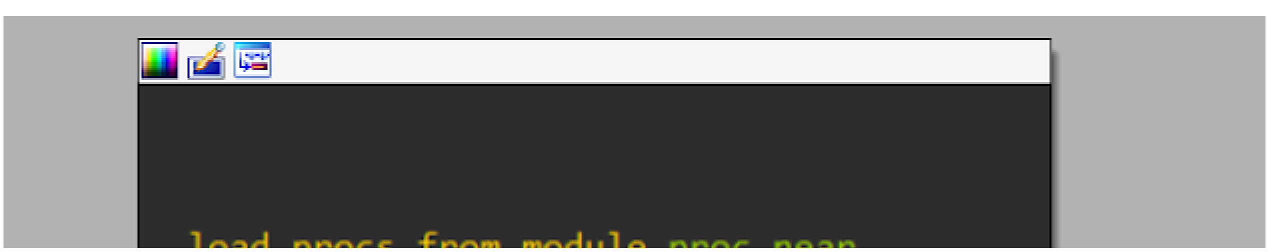
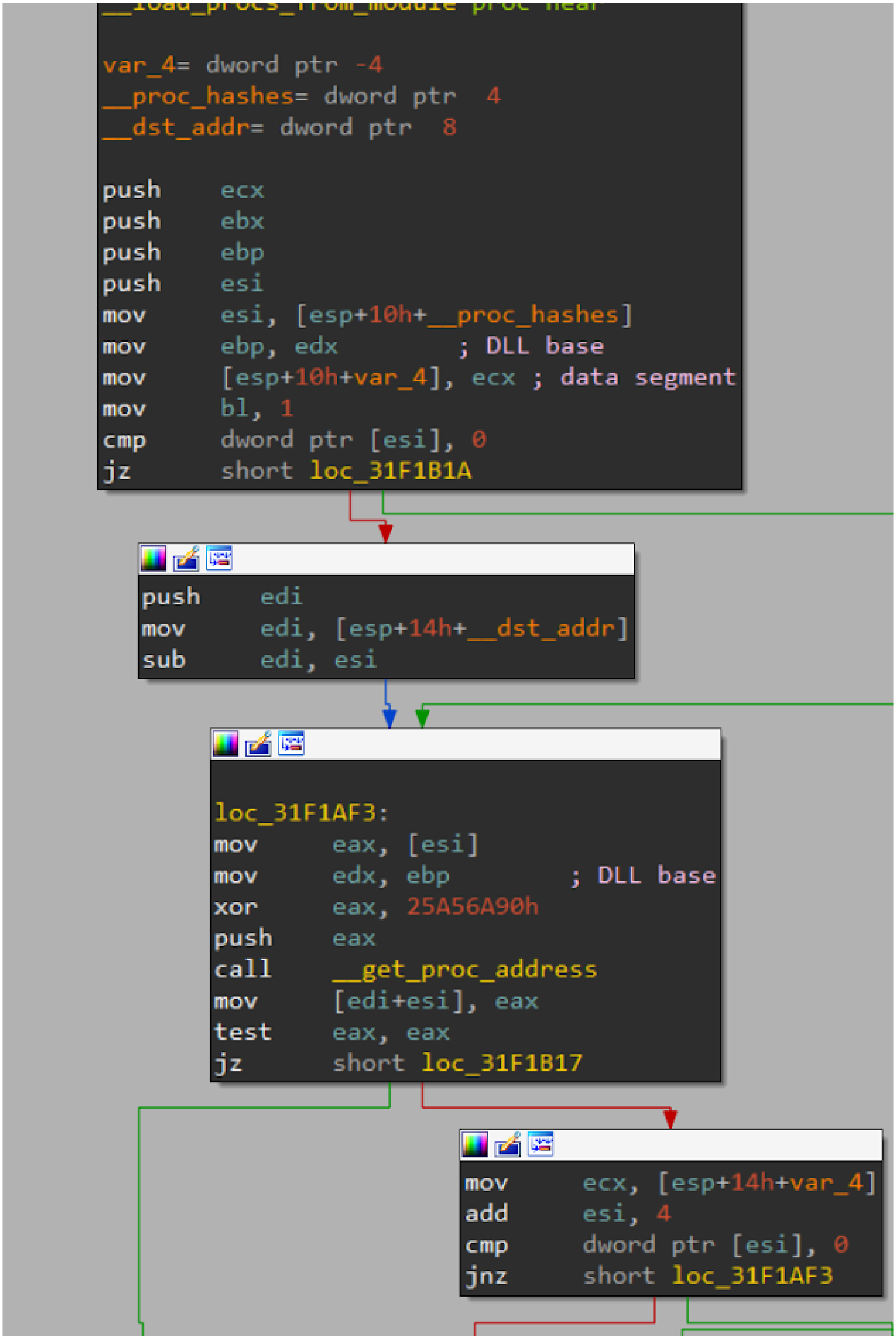


Figure 10 - Array of hashes of "ntdll.dll" function names

Next step is to take a look inside "*__load_procs_from_module*" function. *Figure 11* shows the code for this function. Parameters and functions were named to facilitate the understanding of this code.





var_4= dword ptr -4
__proc_hashes= dword ptr 4
__dst_addr= dword ptr 8
push ecx
push ebx
push ebp
push esi
mov esi, [esp+10h+__proc_hashes]
mov ebp, edx ; DLL base
mov [esp+10h+var_4], ecx ; data segment
mov bl, 1
cmp dword ptr [esi], 0
jz short loc_31F1B1A
push edi
mov edi, [esp+14h+__dst_addr]
sub edi, esi
loc_31F1AF3:
mov eax, [esi]
mov edx, ebp ; DLL base
xor eax, 25A56A90h
push eax
call __get_proc_address
mov [edi+esi], eax
test eax, eax
jz short loc_31F1B17
mov ecx, [esp+14h+var_4]
add esi, 4
cmp dword ptr [esi], 0
jnz short loc_31F1AF3

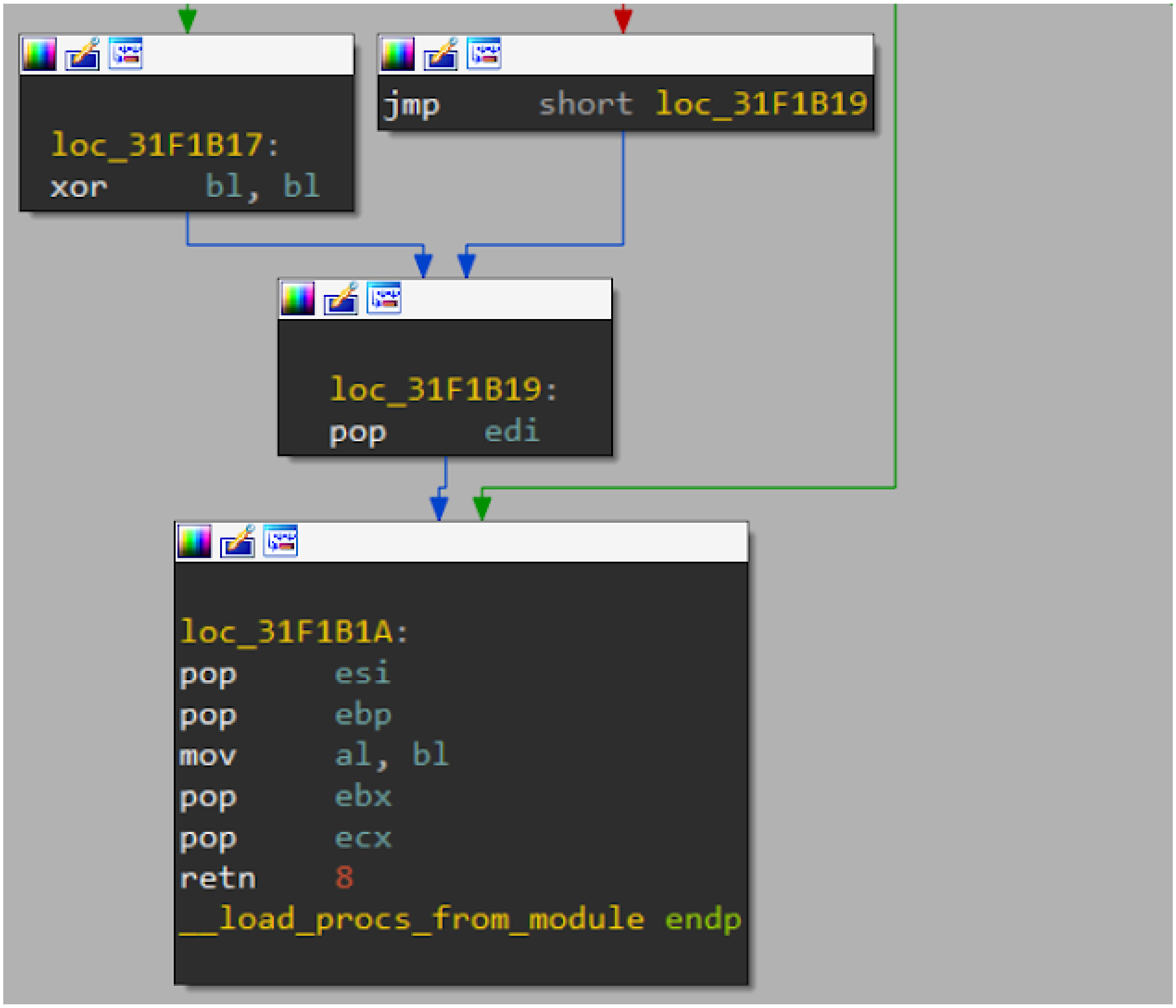
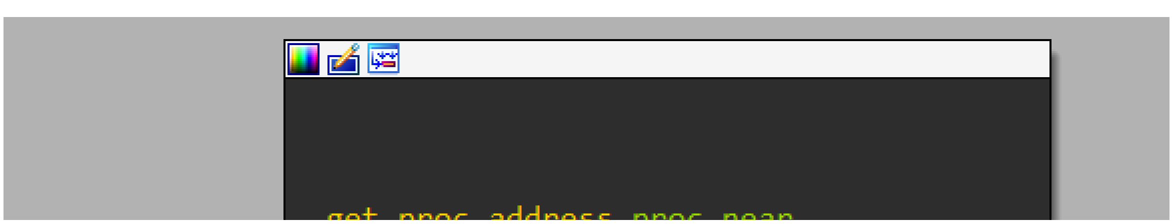
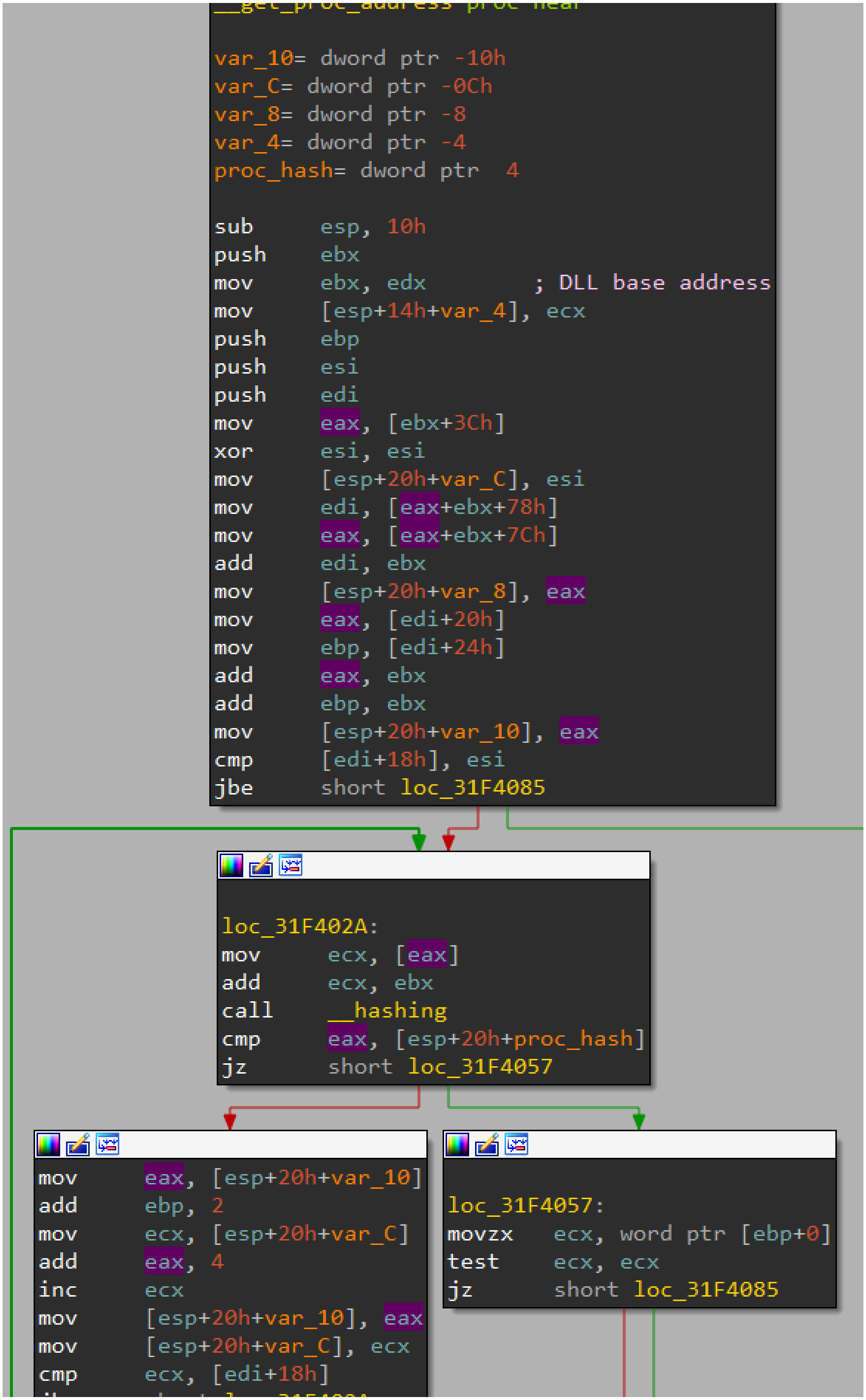


Figure 11 - Code for "*__load_procs_from_modules*" function

This function iterates over a list of *4 bytes* hashes received as parameter. Each element is *XORed* with a hardcoded value (*0x25A56A90)* and passed to the function "*__get_proc_address*" together with a base address of a library. This function iterates over all procedures names exported by a *DLL*, calculates a hash and compares it with the hash received as parameter. If it finds a match, "*__get_proc_address*" returns an address for the specific function.

Lets take a closer look inside "*__get_proc_address*" to figure out how it navigates through the loaded *DLL*. *Figure 12* shows a snip of the code for this function.




var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
proc_hash= dword ptr 4
sub esp, 10h
push ebx
mov ebx, edx ; DLL base address
mov [esp+14h+var_4], ecx
push ebp
push esi
push edi
mov eax, [ebx+3Ch]
xor esi, esi
mov [esp+20h+var_C], esi
mov edi, [eax+ebx+78h]
mov eax, [eax+ebx+7Ch]
add edi, ebx
mov [esp+20h+var_8], eax
mov eax, [edi+20h]
mov ebp, [edi+24h]
add eax, ebx
add ebp, ebx
mov [esp+20h+var_10], eax
cmp [edi+18h], esi
jbe short loc_31F4085
loc_31F402A:
mov ecx, [eax]
add ecx, ebx
call __hashing
cmp eax, [esp+20h+proc_hash]
jz short loc_31F4057
mov eax, [esp+20h+var_10]
add ebp, 2
mov ecx, [esp+20h+var_C]
add eax, 4
inc ecx
mov [esp+20h+var_10], eax
mov [esp+20h+var_C], ecx
cmp ecx, [edi+18h]
loc_31F4057:
movzx ecx, word ptr [ebp+0]
test ecx, ecx
jz short loc_31F4085

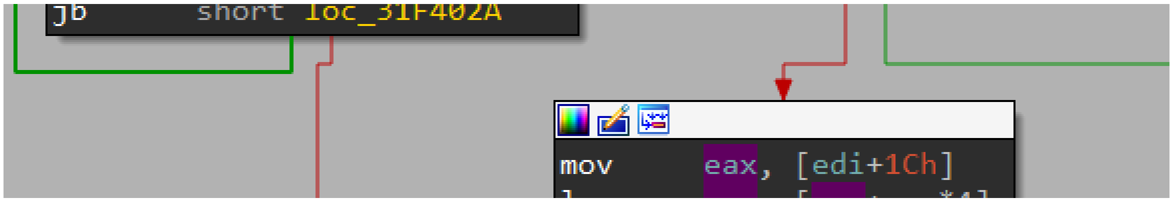


Figure 12 - Code for "*__get_proc_address*" function.

The preamble of the function fetches the address for the *PE* header by accessing offset *0x3C* in the DLL base address. Next step it fetches the relative virtual address (*RVA*) for the export directory at offset *0x78* of the *PE header*. From the Export Directory structure this function fetches the following fields: *NumberOfNames* (offset *0x18*), *AddressOfNames* (offset *0x20*) and *AddressOfNameOrdinals* (offset *0x24*). References for all these structures can be found in the Corkami Windows Executable format overview.

After loading information about the exports the code will iterates through the list of function names and calculates a *4 bytes* hash by calling the "*__hashing*" function (same algorithm described in *Figure 08*). If the output of the "*__hashing*" function matches the hardcoded hash then the ordinal for that function is saved and the address related to that ordinal is returned.

Figure 13 shows a code in Python that reproduces the above mentioned comparison algorithm using hardcoded hashes extracted from memory (*Figure 10*) and all function names exported by ntdll.dll.

```python
ntdll_function_hashes = [
    0x0CE9CF40E, 0x0CBB441ED, 0x1640EF7, 0x728663C8,
    0x0BD8670EE, 0x37C2A503, 0x0E49B8F7C, 0x0ABDA34BE,
    0x12AC99CE, 0x188FF6FC, 0x30BE6C1A, 0x0DD98ACCE,
    0x0E184E6E0, 0x1F70D502, 0x739210B7, 0x5CB65516
]

rol = lambda val, r_bits, max_bits=32: \
    (val << r_bits%max_bits) & (2**max_bits-1) | \
    ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))

def __hashing(arg):
    final = 0
    for c in arg:
        t1 = (c & 0xDF) & 0xff
        final = final ^ t1
        final = rol(final, 8)
        final += t1
        final &= 0xffffffff
    return final

fd = open('ntdll_exports.txt', 'r')
for h in ntdll_function_hashes:
    h2 = h ^ 0x25A56A90
    for c in fd:
        c = c.replace('\n', '')
        if h2 == __hashing(bytes(c, 'utf-8')):
            print('[+] 0x{:x} -> {}'.format(h, c))
    fd.seek(0)
```

Figure 13 - Reversing outcome for code responsible by resolving "*ntdll.dll*" hardcoded hashes

This code produces the following output:

```
[+] 0xce9cf40e -> RtlGetLastWin32Error
[+] 0xcbb441ed -> RtlAllocateHeap
[+] 0x1640ef7 -> RtlReAllocateHeap
[+] 0x728663c8 -> RtlFreeHeap
[+] 0xbd8670ee -> ZwCreateSection
[+] 0x37c2a503 -> ZwMapViewOfSection
[+] 0xe49b8f7c -> ZwUnmapViewOfSection
[+] 0xabda34be -> RtlComputeCrc32
[+] 0x12ac99ce -> RtlMoveMemory
[+] 0x188ff6fc -> RtlZeroMemory
[+] 0x30be6c1a -> atoi
[+] 0xdd98acce -> LdrGetDllHandle
[+] 0xe184e6e0 -> RtlGetVersion
[+] 0x1f70d502 -> ZwQueryInformationProcess
[+] 0x739210b7 -> LdrProcessRelocationBlock
[+] 0x5cb65516 -> RtlRandomEx
```

Finally, these addresses are used for filling the call table which will be referenced by indirect calls in the main payload. It is possible to confirm that what was described so far is true by observing the function addresses written in the data segment after executing the second section of "*__load_libraries*". *Figure 14* shows the part of the call table filled so far with the expected "*ntdll.dll"* calls.

Address	Value	Comments
002E0F52	772C9AED	ntdll.RtlGetLastWin32Error
002E0F56	772B2DD6	ntdll.RtlAllocateHeap
002E0F5A	772CFF51	ntdll.RtlReAllocateHeap
002E0F5E	772B2C6A	ntdll.RtlFreeHeap
002E0F62	772A56E8	ntdll.ZwCreateSection
002E0F66	772A5C28	ntdll.NtMapViewOfSection
002E0F6A	772A69B8	ntdll.NtUnmapViewOfSection
002E0F6E	7726DD8A	ntdll.RtlComputeCrc32
002E0F72	77295CC0	ntdll.RtlMoveMemory
002E0F76	77295C90	ntdll.RtlZeroMemory
002E0F7A	77269D0D	ntdll.atoi
002E0F7E	772B8AF8	ntdll.LdrGetDllHandle
002E0F82	772C65E3	ntdll.RtlGetVersion
002E0F86	772A6048	ntdll.NtQueryInformationProcess
002E0F8A	77262F1D	ntdll.LdrProcessRelocationBlock
002E0F8E	772C283D	ntdll.RtlRandomEx
002E0F92	00000000	
002E0F96	00000000	

Figure 14 - Segment of Smokeloader's dynamically generated call table

The last segment of the "*__load_libraries*" function de-obfuscates the remain libraries names and load them by using the same resources used for loading "*ntdll"* and "*kernel32*". The libraries loaded by *Smokeloader* are: "*user32*", "*advapi32*", "*urlmon*", "*ole32*", "*winhttp*", "*ws2_32*", "*dnsapi*" and "*shell32*".

Now that the whole process of creating the *call table* used by the indirect calls is described, next step will get into fixing the memory containing the main payload by using *IDA Python*.

--- End of part II ---

When the main payload of *Smokeloader* is imported into *IDApro* it is possible to see code containing indirect calls which uses a base address stored in a register plus an offset. *Figure 15* presents a snip of the main payload containing such indirect calls.

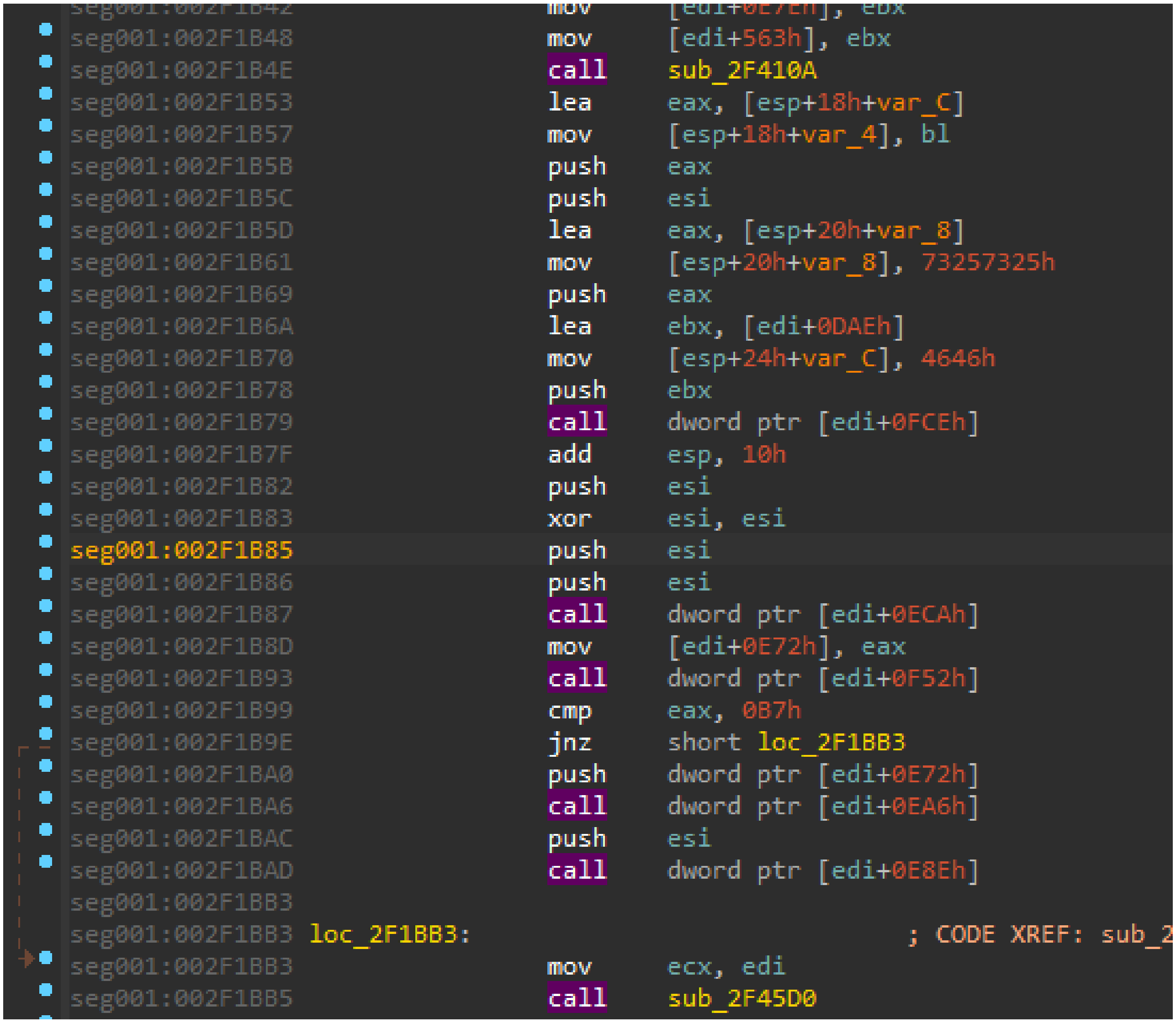


Figure 15 - Indirect calls calling functions pointed at the dynamic generated calls table.

This characteristic makes the processing of reversing this code harder since the interaction with other resources in the Operating System is not clear as all external calls is not explicit. The goal in this part of the article is to patch these calls for pointing to addresses we going to map and label (using *IDA Python*). The code below implements the change we want.
This code performs the following actions into our *IDB*:

1. Reads a memory dump of the data segment of an executing *Smokeloader* binary (line *106*);
2. Creates a *DATA* segment mapped into *0x00000000* (line *107*).
3. Loads the dumped data segment from the running sample into this new segment (line *35*);
4. Imports API names extracted from *x32dbg* to specific positions in the new data segment (line *112*);
5. Patches all indirect call instructions (opcode *55 9X*) to direct call instructions (line *51*).

Figure 16 shows the code listed after executing the script above. As we can see, all indirect calls were translated to direct calls to a labeled table located in the freshly created data segment starting at address *0x00000000*.

```
seg001:002F1B48                 mov     [edi+563h], ebx
seg001:002F1B4E                 call    sub_2F410A
seg001:002F1B53                 lea     eax, [esp+18h+var_C]
seg001:002F1B57                 mov     [esp+18h+var_4], bl
seg001:002F1B5B                 push    eax
seg001:002F1B5C                 push    esi
seg001:002F1B5D                 lea     eax, [esp+20h+var_8]
seg001:002F1B61                 mov     [esp+20h+var_8], 73257325h
seg001:002F1B69                 push    eax
seg001:002F1B6A                 lea     ebx, [edi+0DAEh]
seg001:002F1B70                 mov     [esp+24h+var_C], 4646h
seg001:002F1B78                 push    ebx
seg001:002F1B79                 call    user32_wsprintfA
seg001:002F1B7F                 add     esp, 10h
seg001:002F1B82                 push    esi
seg001:002F1B83                 xor     esi, esi
seg001:002F1B85                 push    esi
seg001:002F1B86                 push    esi
seg001:002F1B87                 call    kernel32_CreateMutexA
seg001:002F1B8D                 mov     [edi+0E72h], eax
seg001:002F1B93                 call    ntdll_RtlGetLastWin32Error
seg001:002F1B99                 cmp     eax, 0B7h
seg001:002F1B9E                 jnz     short loc_2F1BB3
seg001:002F1BA0                 push    dword ptr [edi+0E72h]
seg001:002F1BA6                 call    kernel32_CloseHandle
seg001:002F1BAC                 push    esi
seg001:002F1BAD                 call    ntdll_RtlExitUserThread
seg001:002F1BB3
seg001:002F1BB3 loc_2F1BB3:                             ; CODE XREF: sub
seg001:002F1BB3                 mov     ecx, edi
seg001:002F1BB5                 call    sub_2F45D0
```

Figure 16 - Patched code with calls containing meaningful labels.

Just heads up for preventing people against messing up research IDBs: for obvious reasons (different instruction sets) the script above **can not** be used for patching *64 bits Smokeloader* IDBs but it could be easily adapted to do the same task.

--- End of part III ---

That's all folks!

The ideas described in this article can be extended and used to analyse any other malware families dynamically importing libraries and using indirect calls. Another thing cool for experimenting in future would be write a script which loads *DLLs* and extracts labels statically by using the reversed "__*hashing*" function and native functionalities in *IDA* for mapping DLLs in the process address space.