Taking a deep dive into SmokeLoader

farghlymal.github.io/SmokeLoader-Analysis/

March 1, 2024

Aziz Farghly

Malware Reverse Engineer

16 minute read

Smoke Loader Analysis

Smoke Loader, software introduced in 2011, is primarily utilized for loading subsequent stages of malware onto systems, particularly information stealers designed to extract credentials through various means.

Its widespread acclaim can be attributed to its advanced Anti-Analysis and Anti-debugging techniques, along with its stealthy behavior, which poses challenges for detection. Notably, Smoke Loader employs consistent efforts to obfuscate its Command and Control (**C2**) operations by simulating communication requests that resemble legitimate traffic patterns to well-known websites, including microsoft.com, bing.com, adobe.com, and others.

Originally marketed under the name SmokeLdr on dark-web platforms, Smoke Loader has been exclusively available to threat actors based in Russia since 2014.

Smoke Loader is typically disseminated through malicious documents, primarily Word or PDF files, often distributed via spam emails or targeted spear-phishing campaigns. The malware is activated upon interaction with such malicious documents, initiating its deployment onto the system. Subsequently, Smoke Loader injects malicious code into compromised system processes, such as explorer.exe, thereby initiating its malicious operations while masquerading as a normal process.

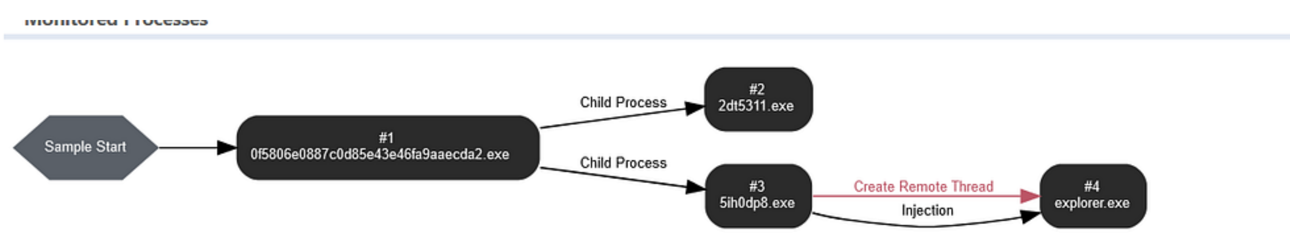


Figure 1. File analysis on **VMRay** platform

Technical Analysis

The sample we have today is compiled in May/2023 so not that old.

> ***sha1: C6BA6E91D40AA1507775077F9662ECB25C9F0943***

Smoke loader in this campaign comes packaged with Wextract which is a Win32 Cabinet Self-Extractor, understanding Cabinet structure is not hard we need to explore file resources and determine which file will be extracted by this extractor and then extract it statically without the need to run the extractor.

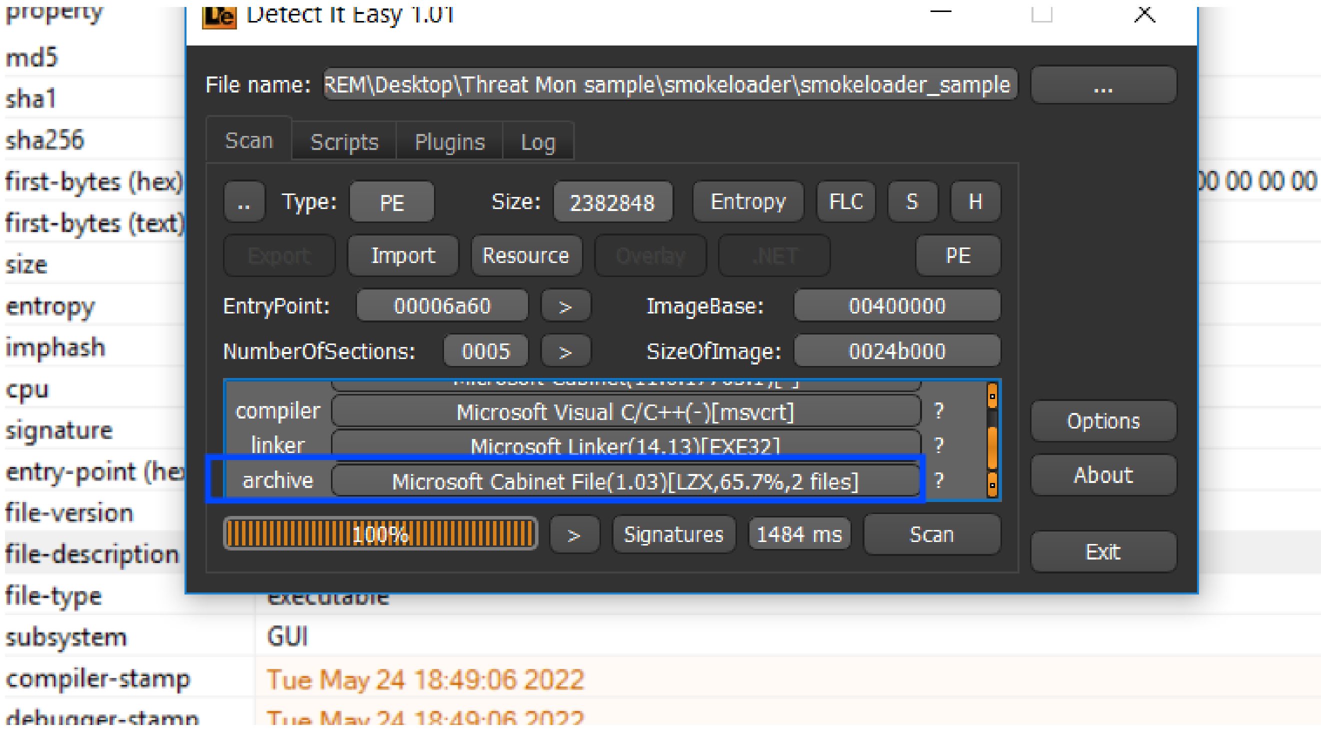
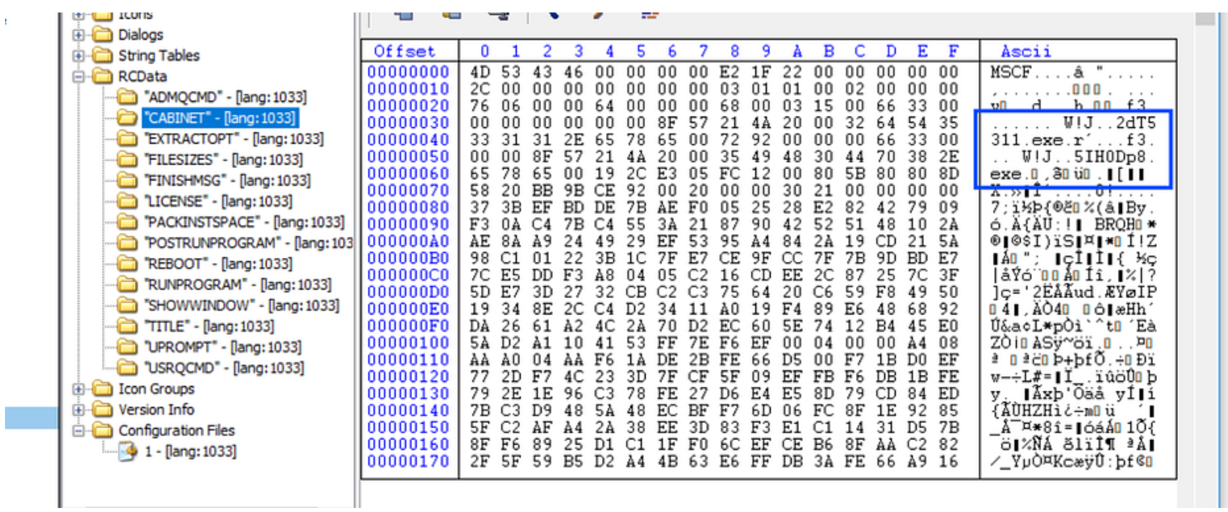
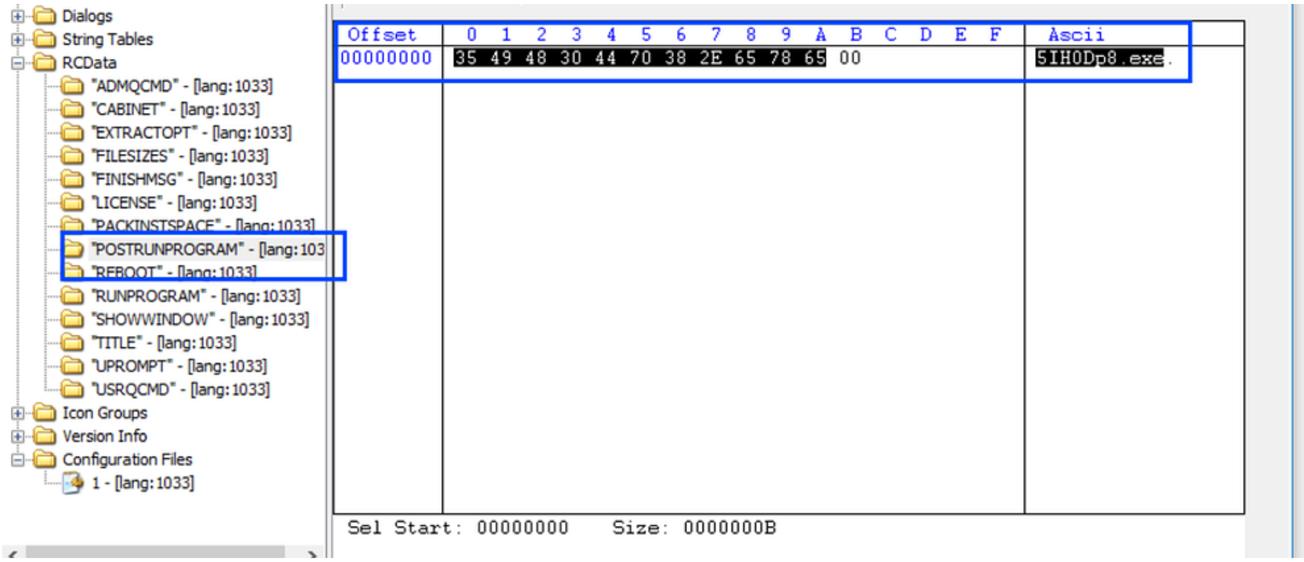


Figure 2. Viewing file type on DIE tool

navigating the **resource** section, **RCData** path, and “**CABINET**” icon, we find a reference to **exe files**.



then going to “**POSTRUNPROGRAM**” I found a mention of **5IH0Dp8.exe**

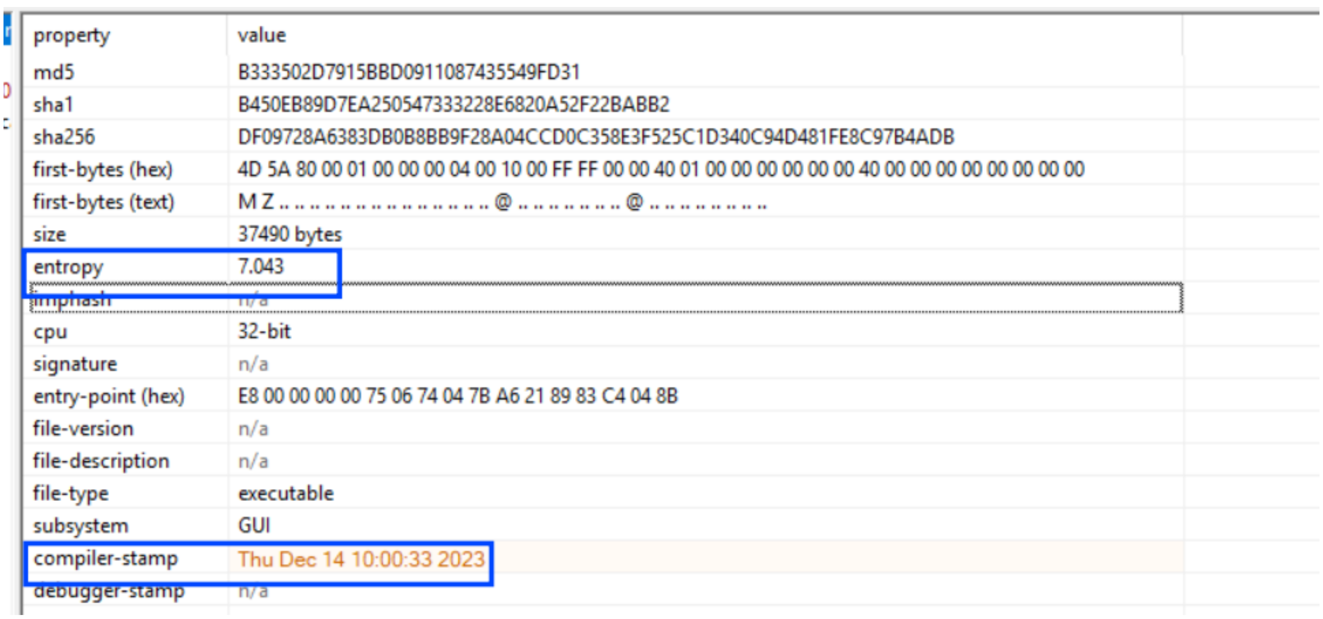


Extracting the executables embedded in this file, especially my focus will go on the sample mentioned in “**POSTRUNPROGRAM”** element.

Stage 2

the sample is an x86 Pe file with high entropy that indicates a decryption or packing stream.

> ***sha1:B450EB89D7EA250547333228E6820A52F22BABB2***



property	value
md5	B333502D7915BBD0911087435549FD31
sha1	B450EB89D7EA250547333228E6820A52F22BABB2
sha256	DF09728A6383DB0B8BB9F28A04CCD0C358E3F525C1D340C94D481FE8C97B4ADB
first-bytes (hex)	4D 5A 80 00 01 00 00 00 04 00 10 00 FF FF 00 00 40 01 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
first-bytes (text)	M Z .. .. @ .. .. @ .. ..
size	37490 bytes
entropy	7.043
imphash	n/a
cpu	32-bit
signature	n/a
entry-point (hex)	E8 00 00 00 00 75 06 74 04 7B A6 21 89 83 C4 04 8B
file-version	n/a
file-description	n/a
file-type	executable
subsystem	GUI
compiler-stamp	Thu Dec 14 10:00:33 2023
debugger-stamp	n/a

Figure 3. Getting File Entropy and and compilation time

the sample also has no imports and strings and got flagged as smoke loader by 60 AV engine through VT API used in PE-Studio software which ensures our predication that this sample is the 2 Stage of Smoke loader campaign and the other one maybe acts as a decoy.

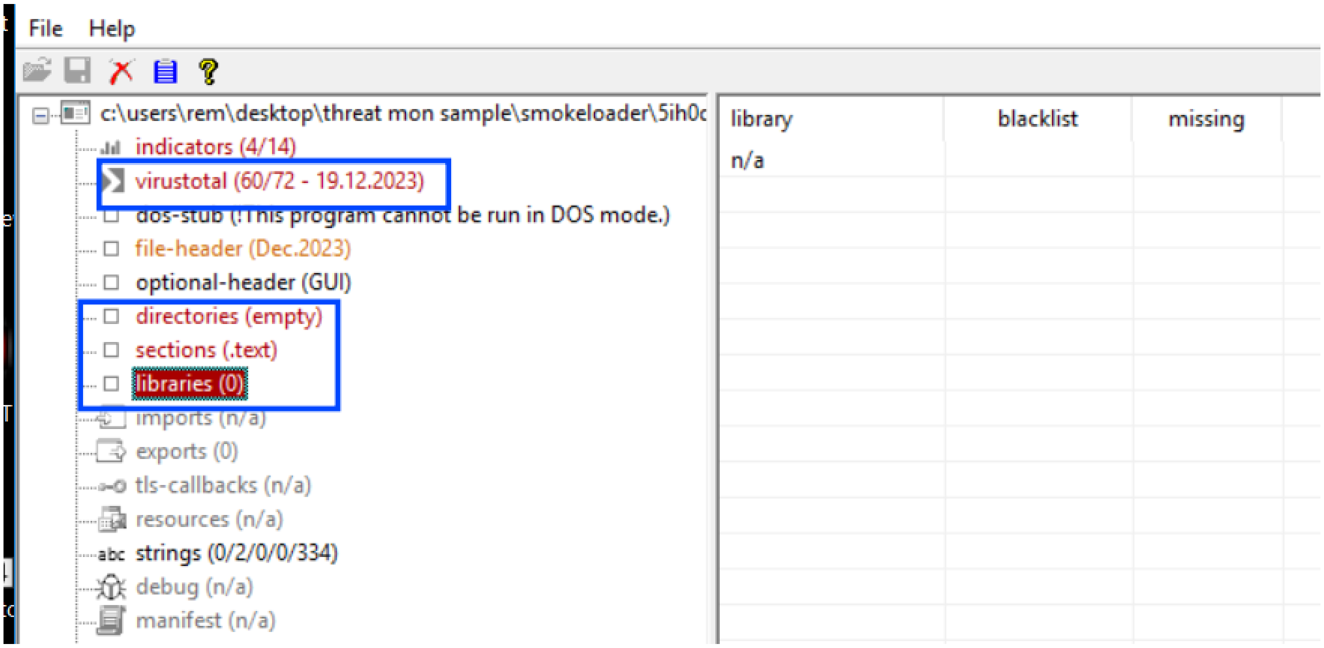


Figure 4. Sample Flaged by VT, No Imports found

I tried to execute the malware using some monitoring tools but found that the process was terminated Immediately there is the basic usage of Windows APIs, which guides us to VM detection or Analysis detection mechanism, so in the next part, I will discuss how Smoke loaders work and how are modules works.

Code Analysis

the next phase of this article will involve static code analysis to get more info about smoke loader functionality.

Opaque Predicates

Smoke Loader welcomed us with anti-analysis techniques called Opaque Predicates, which trick the disassembler engines into producing a wrong code, also the technique acts as control flow obfuscation that makes the analysis process harder and more confusing due to the high usage of garbage code and the use of **jnz/jz** instructions which are pointing for the same address, the figure below will demonstrate more.

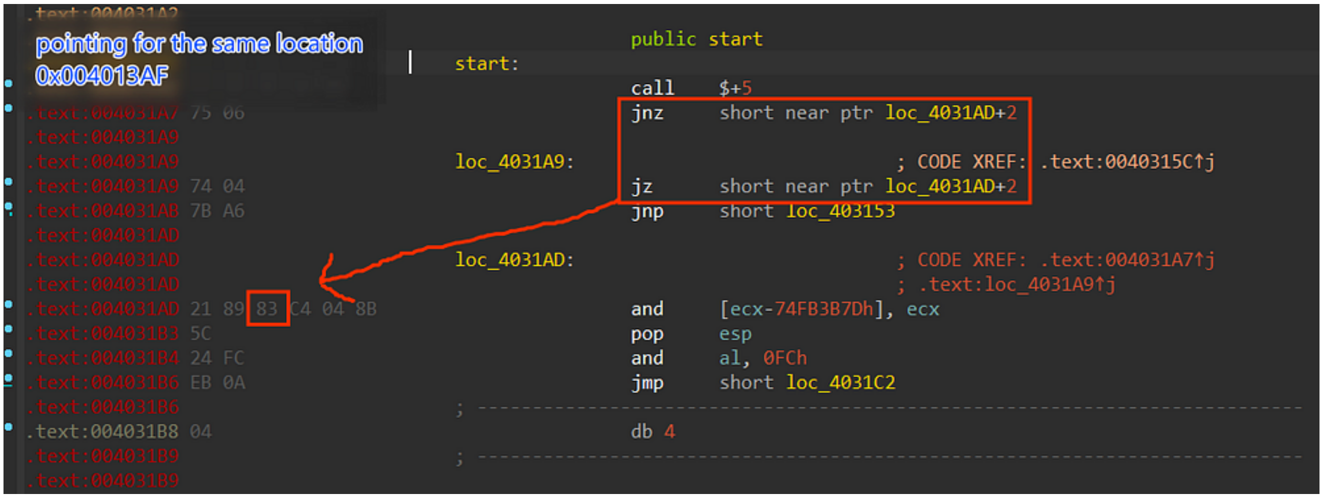


Figure 5. Tricking Disassembler using Opaque Predicates

to make it easier we need to patch this code and fix this junk of jumps by replacing **jz/jnz** with unconditional jump **** using a simple Python code that uses IDA python to fix it

this code belongs to **n1ght-w0lf**, big Thanks to him.

```
import idc
ea = 0
while True:
 ea = min(idc.find_binary(ea, idc.SEARCH_NEXT | idc.SEARCH_DOWN, "74 ? 75 ?"), # JZ /
JNZ
 idc.find_binary(ea, idc.SEARCH_NEXT | idc.SEARCH_DOWN, "75 ? 74 ?")) # JNZ / JZ
 if ea == idc.BADADDR:
 break
 idc.patch_byte(ea, 0xEB) # JMP
 idc.patch_byte(ea+2, 0x90) # NOP
 idc.patch_byte(ea+3, 0x90) # NOP
```

the result was good enough to make the code more readable, the conditional jumps converted into non-conditional jumps, and **nopping** the bytes of the original jumps

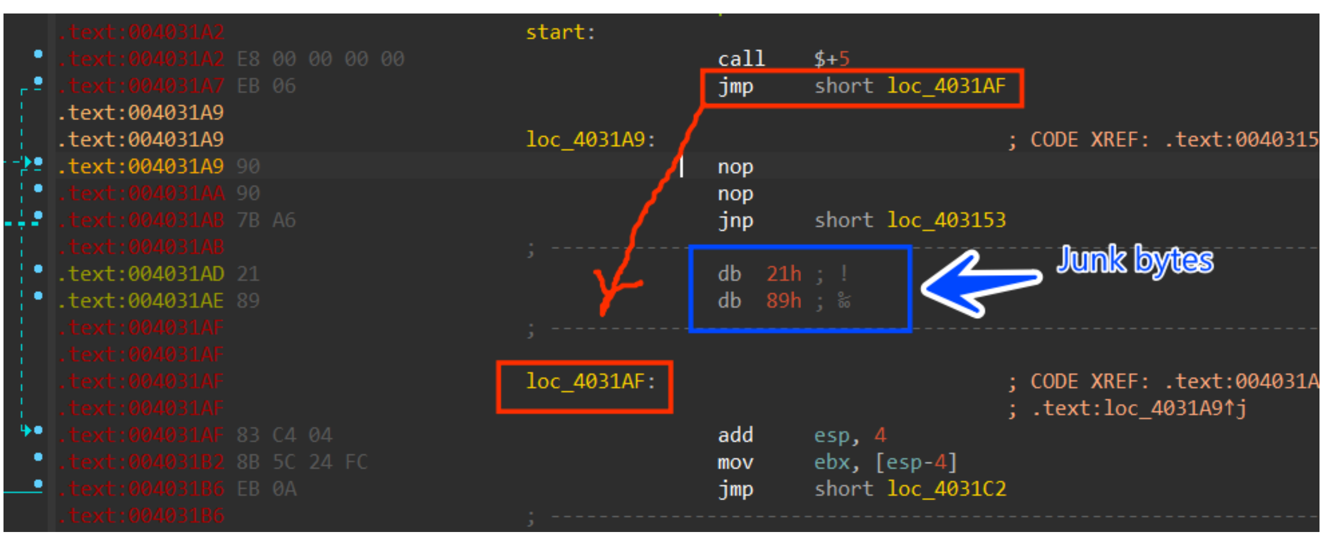


Figure 6. After fixing JMPs using the above script

smoke loader code is so obfuscated that we need to go step by step in the code to identify where the 3 stages will be dropped or downloaded by the Smoke loader, so we still need to fix all of this, by using Python code to fix it and convert all these junk bytes into a nop byte to be able to create a function in IDA pro.

Anti-Debugging

after trying to fix the code we finally got a regular function, smoke reads the **PEB** structure to obtain access to the element placed at **0xA4** which points to **OSMajorVersion** which classifies Windows version, if it's less than 6 which means it's running in an old windows version **[XP or W server 2003]**

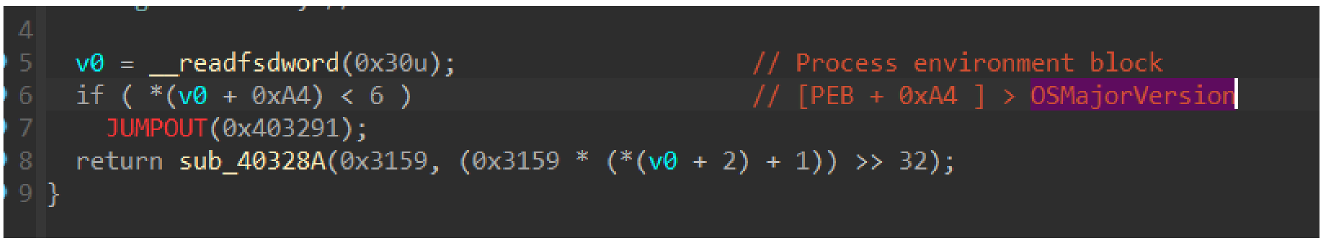


Figure 7. Getting Windows Version through PEB Structure

Transferring Control Flow

after that, Smokeloader does not use normal calls or jumps, instead, it uses the **[push-ret] or [mov [esp] , value]** method cause when the **ret** instruction is executed it pops the top of the stack into **EIP** or instruction pointer

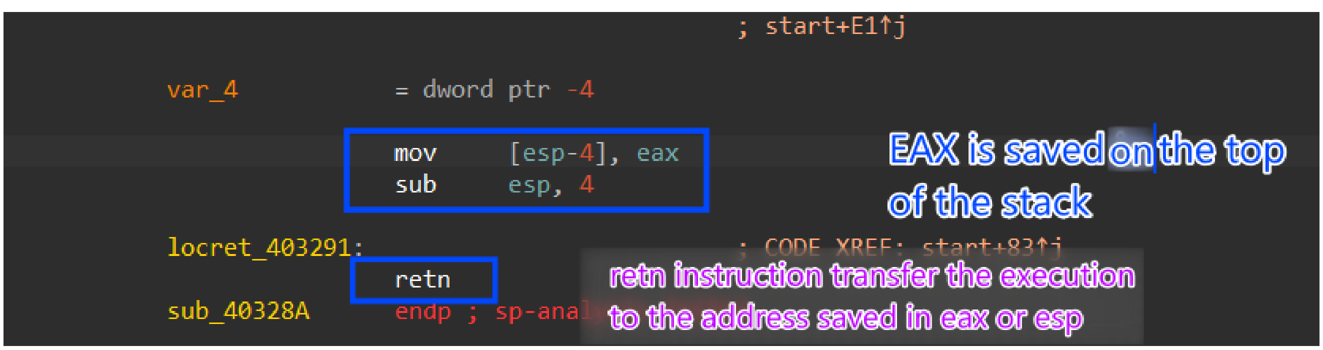


Figure 8. Transferring execution using [push ret]

so the sample here will not provide us with the address to jump to, we need to identify it manually, the address is being saved into **ecx**, and using **mul** instruction the value is moved to **eax** and then adding the value in **eax** to the image base **(ebx value)** which in our case is **0x400000** so the next jump will point to **0x403159**

```
loc_403267:                             ; CODE XREF: st
                mov     ecx, 3159h
                jmp     short loc_403273
; ---------------------------------------------------
```

Figure 9. Moving 0x3159h to **ecx** register

```
loc_403276:                             ; CODE XREF: start:
                mul     ecx
                jmp     short loc_40327F
; ---------------------------------------------------
```

Figure 10. Multiplying by ecx will move part of the result to eax

```
add     eax, ebx        ; eax = 0x3159 and ebx = 0x400000
                        ; so eax = 0x403159
```

Figure 11. Constructing the final address by adding it to the base address in **ebx**

Decrypt on-demand

after some reversing and following the malware jumps which were so confusing and made me stuck, I found that Smoke is decrypting the function that will be executed and after executing it re-encrypt it again to stay as stealthy and evasive as it can, the malware saves the offset of the address of the function to be decrypted for further execution and then re-encryption on **eax** register and the length is saved on **ecx** register and the Xor decryption key is saved on **edx** register before calling the decryption routine which also acts as encryption routine after executing the decrypted function

```
loc_401194:                             ; CODE XREF: .text:loc_4
                push    11CCh           ; address = 0x4011CC
                mov     eax, [esp]
                add     esp, 4
                jmp     short loc_4011A6
; ---------------------------------------------------
```

Figure 12. saving the address of the function to be decrypted on eax

```
loc_4011AB:                             ; CODE XREF: .text:loc_4011B2↓j
                push    5Fh ; '_'
                pop     ecx
                jmp     short loc_4011B5
; ---------------------------------------------------
```

Figure 13. The size of the function is saved on ecx

The Xor Key which is specified for this function is saved on edx, every function has its own decryption key.

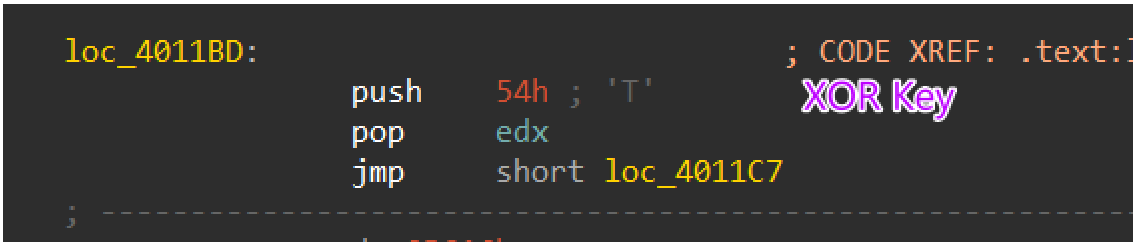


Figure 14. The Xor Key is saved on edx

and here is the part responsible for applying Xoring.

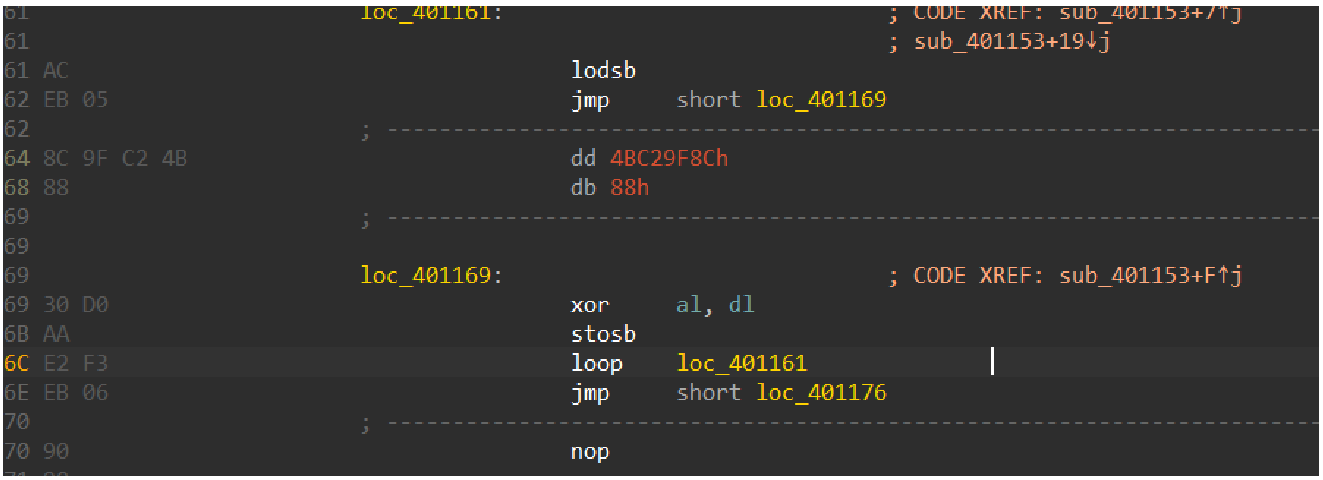


Figure 15. Xoring Blob

the decryption routine has been called many times and each time it encrypts the address after the call instruction

which is the first call or first function to be decrypted and then executed and then re-encrypted is **0x4011CC**

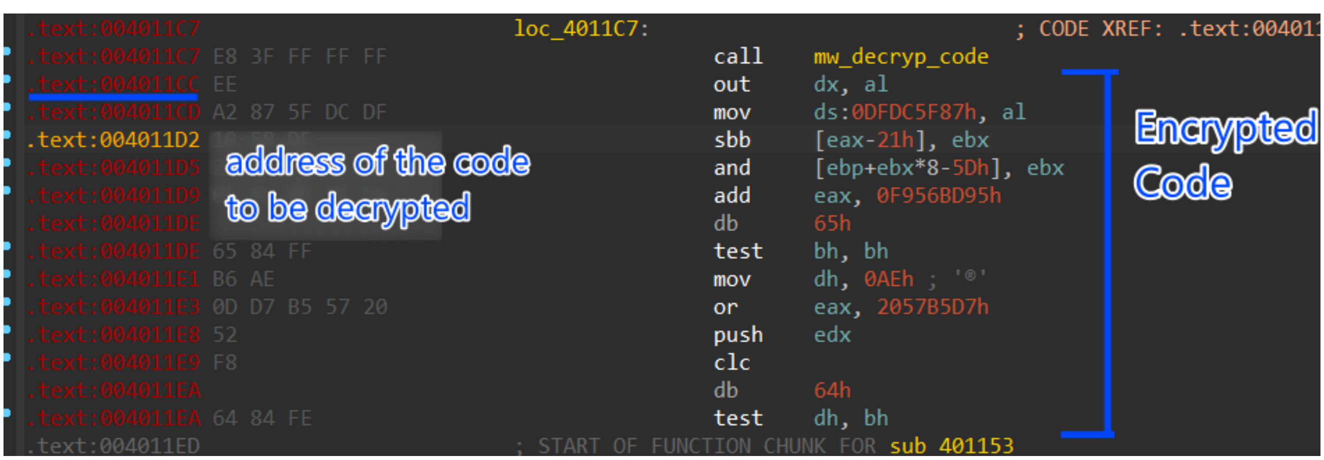


Figure 16. First encrypted function

so to fix this we need to simulate the decryption process and patch the bytes, and because there is not a static pattern Smoke uses it to push arguments to the decryption **function(offset,size,xor_key)** so I found that there is a **20** function call to **mw_decrypt_code()** which is responsible for decrypting the code, so I go through all of them manually using a simple P**ython** code to xor and patch the bytes using **IDA python**

```
def xor_chunk(offset, size,xor_key):
 ea = 0x400000 + offset
 for i in range(size):
 byte = ord(idc.get_bytes(ea+i, 1))
 byte ^= xor_key
 idc.patch_byte(ea+i, byte)
```

and here is how the code of 0x4011CC after decryption, looks normal and clean.

Figure 17. After Decrypting the code at address 0x4011CC

and here is how the function **0x4011CC** will re-encrypt itself after executing its content

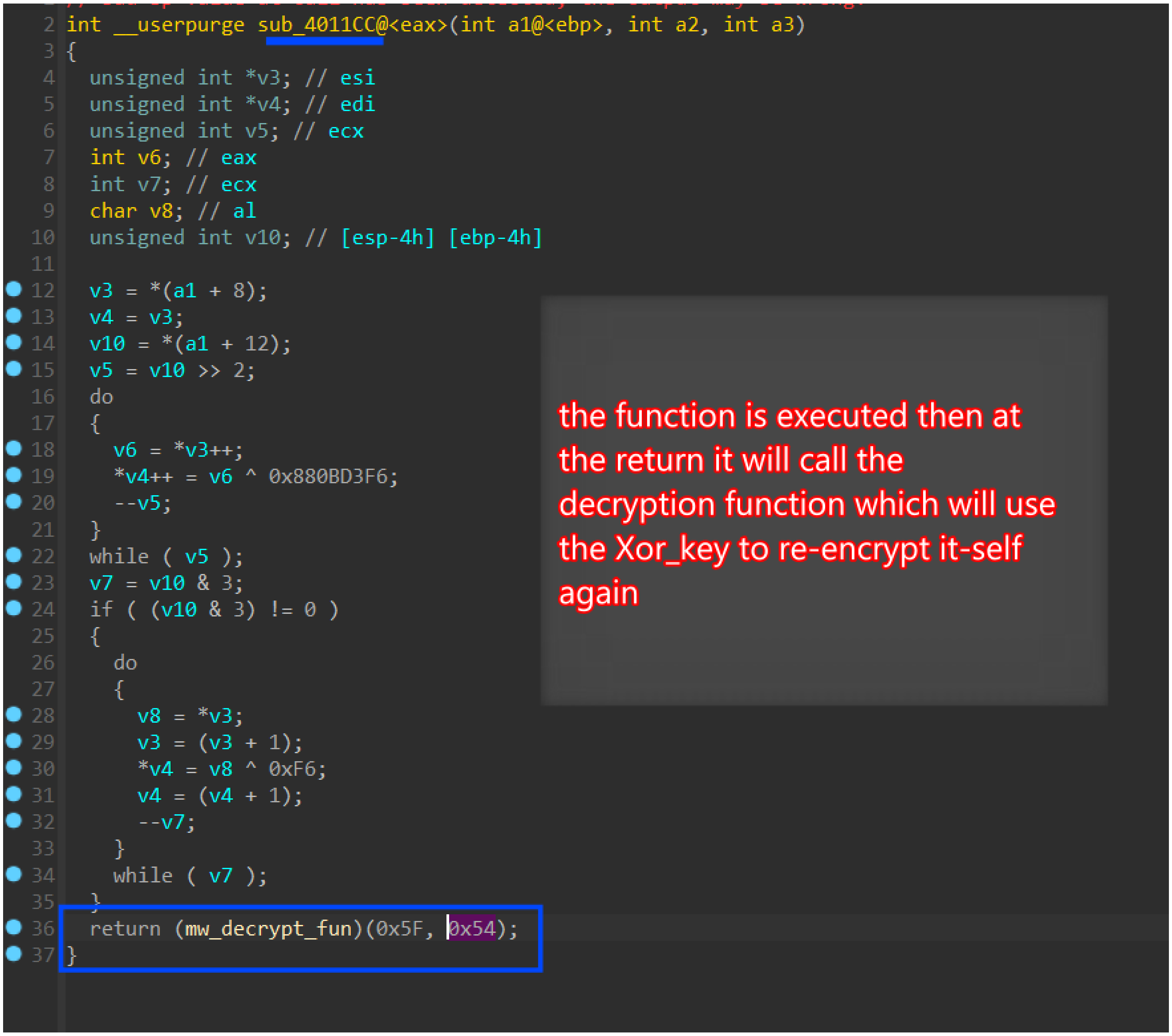


Figure 18. The function re-encrypts itself again after execution

using the code above I went through all the encrypted functions and decrypted them one by one and commented in every call to identify what address was being decrypted or encrypted, as you will see in the figure below.

Direction	Type	Address	Text
Do...	p	sub_401231:loc_40127A	call mw_decrypt_fun; for decrypting 0x40127F function
Do...	p	sub_40127F:loc_40137F	call mw_decrypt_fun; for re-encrypting 0x40127F
Do...	p	sub_40138E:loc_4013DA	call mw_decrypt_fun; for decrypting 0x4013DF function
Do...	p	sub_4013DF:loc_401484	call mw_decrypt_fun; for re-encrypting 0x4013DF function
Do...	p	.text:loc_4014D8	call mw_decrypt_fun; for decrypting 0x4014DD function
Do...	p	.text:loc_401866	call mw_decrypt_fun; for re encryption 0x4014DD function not for decryption stuff
Do...	p	sub_401872:loc_4018B5	call mw_decrypt_fun; for decrypting 0x004018BA function
Do...	p	sub_4018BA:loc_401923	call mw_decrypt_fun; for re-encrypting 0x4018BA function
Do...	p	.text:loc_401979	call mw_decrypt_fun; for decrypting 0x40197E function
Do...	p	sub_401AED:loc_401B41	call mw_decrypt_fun; for re-encrypting 0x40197E function
Do...	p	sub_401B4D:loc_401B92	call mw_decrypt_fun; for decrypting 0x401B97 function
Do...	p	sub_401B97:loc_401BF8	call mw_decrypt_fun; for re-encrypting 0x401B97 function
Do...	p	.text:loc_401C54	call mw_decrypt_fun; for decrypting 0x401C59 function
Do...	p	sub_401CCC:loc_401D2A	call mw_decrypt_fun; for re-encrypting 0x401C59 function
Do...	p	sub_401D39:loc_401D7D	call mw_decrypt_fun; for decrypting 0x401D82 function
Do...	p	sub_401D82:loc_401E69	call mw_decrypt_fun; for re-encrypting 0x401D82 function

Figure 19. Decryption and Re-Encryption for every function

API Hashing

After decrypting and patching All functions and trying to push comments in assembly view to make it easier to track function calls and control flow, the first decrypted function here is 0x4011CC this function decrypts a small punch of data, using a different XOR key [**0x0x880BD3F6**]

```
v3 = *(a1 + 8);
v4 = v3;
size = *(a1 + 0xC);
counter = size >> 2;                        // counter  = 0xC2
do
{
  Data = *v3++;
  *v4++ = Data ^ 0x880BD3F6;
  --counter;
}
while ( counter );
v7 = size & 3;
if ( (size & 3) != 0 )
{
  do
```

Figure 20. Sub_4011CC applies decryption stuff

first, this decrypted data did not make sense to me cause I found it useless but then after starting again from the start function, after fixing some of the obfuscation, I found that the malware tried to get the address of **ntdll.dll** in memory which absolutely will use it to resolve needed APIs via hashing

```
pop     ebx
sub     ebx, 2B63h      ; ebx = image base
mov     eax, esi        ; eax --> PEB
mov     eax, [eax+0Ch]  ; PEB_LDR_DATA *Ldr;
mov     eax, [eax+1Ch]  ; InInitializationOrderModuleList
mov     eax, [eax+8]    ; Ntdll Address in memory
test    eax, eax
jz      locret_402EE5
mov     [ebp-18h], eax
lea     eax, [ebx+3292h]
mov     [ebp-4], eax
push    dword ptr [ebp-4]
push    dword ptr [ebp-18h] ; Ntdll Address in memory
call    mw_Build_IAT_0  ; build for 0x403292  --> first APIs to be resolved
test    eax, eax
jz      locret_402EE5
jmp     short loc_402BB2
```

Figure 21. Getting **Ntdll.dll** address using **PEB**

getting into **mw_Build_IAT_0()** function reveals some secrets about the hashing algorithm used by Smokeloader.

Encrypted Hashes

the below code decrypts hashes and patches them in IDA pro

```
def xor_chunk_API(offset, n, key, is_big_endian=False):
 ea = 0x400000 + offset
 for i in range(0, (n//4)*4, 4):

 chunk = idc.get_bytes(ea + i, 4)

 if is_big_endian:
 chunk = chunk[::-1]

 value = int.from_bytes(chunk, byteorder='little')

 xor_result = value ^ key

 xor_bytes = xor_result.to_bytes(4, byteorder='little')

 idc.patch_bytes(ea + i, xor_bytes)
```

here is the hashing routine which is called **djb2**

Figure 22. API hashing routine

```
def hash_djb2(API_Name):
hash = 0x1505
for x in API_Name:
hash = (( hash << 5) + hash) + x
return hash & 0xFFFFFFFF
```

using **HashDb** to resolve these APIs

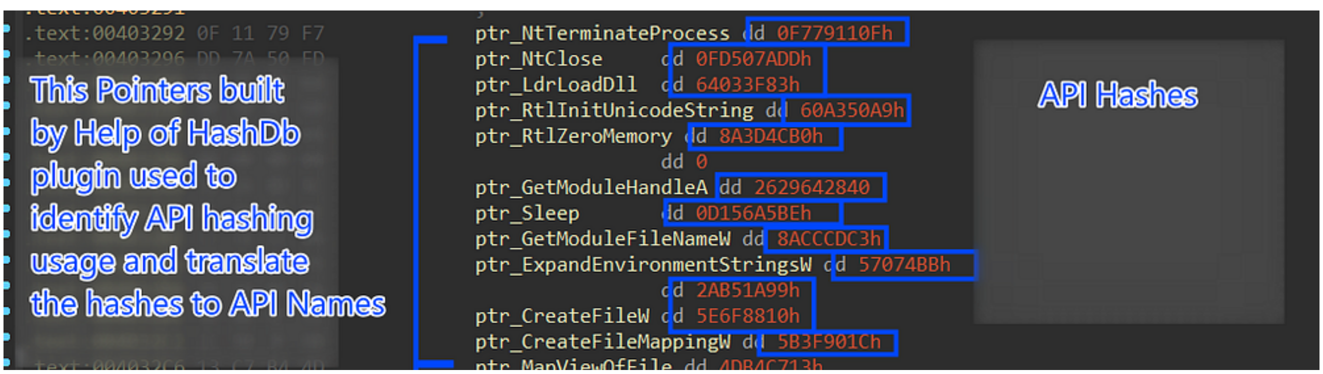


Figure 23. Replacing Hashs with names using **HashDB**

so after resolving All APIs, which is more than 40 APIs Now we need to go through the malware to identify its behavior.

Skip infection

after API building it will check the location of the current machine via keyboard language, which will be used to avoid infecting some countries **(Russia, Ukraine)**, It will get the keyboard language list and then compare it to constants that refer to the language of Russia and Ukraine

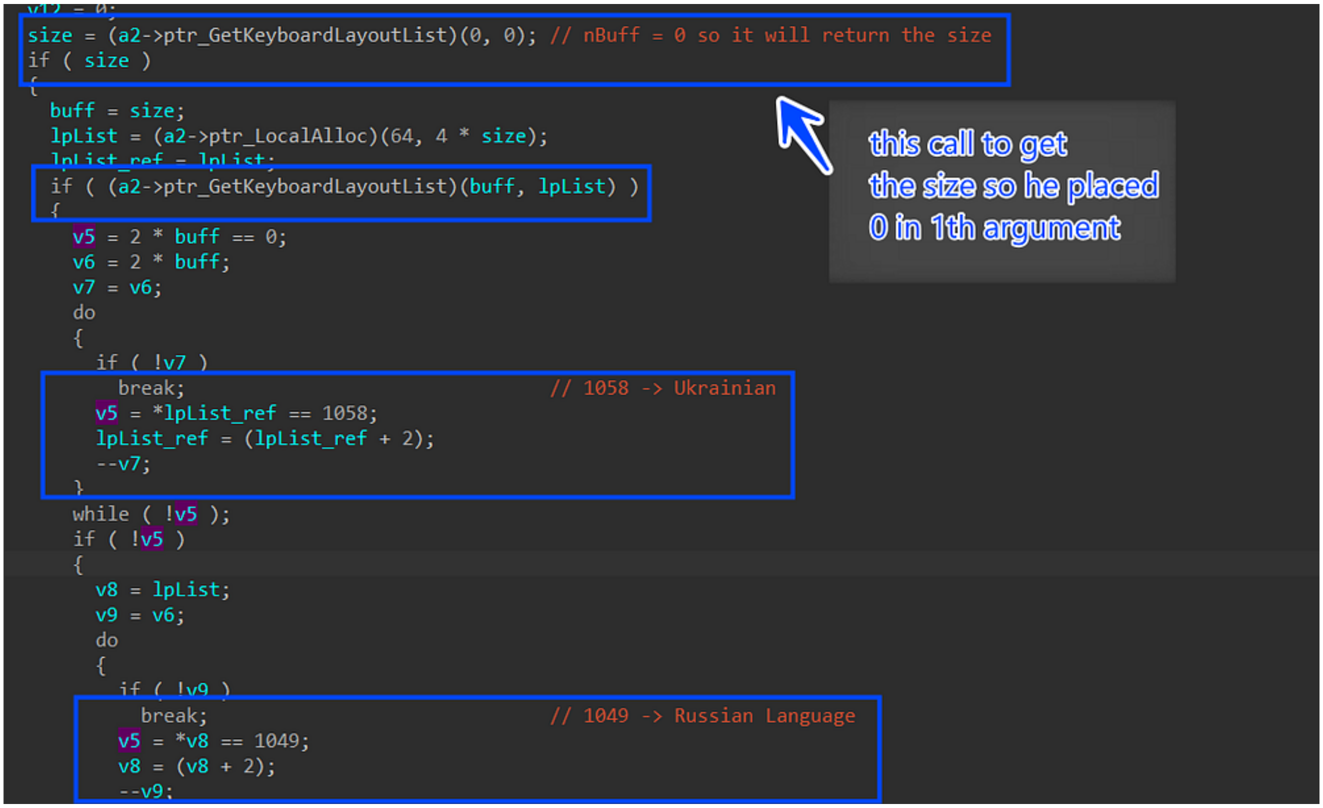


Figure 24. Skip infecting Russia and Ukraine

Check Privilege

after that, it will get the process token via OpenProcessToken API and then try to query [**TokenIntegrityLevel**] and check if it is less than **0x2000** which means that the malware with a Low integrity level

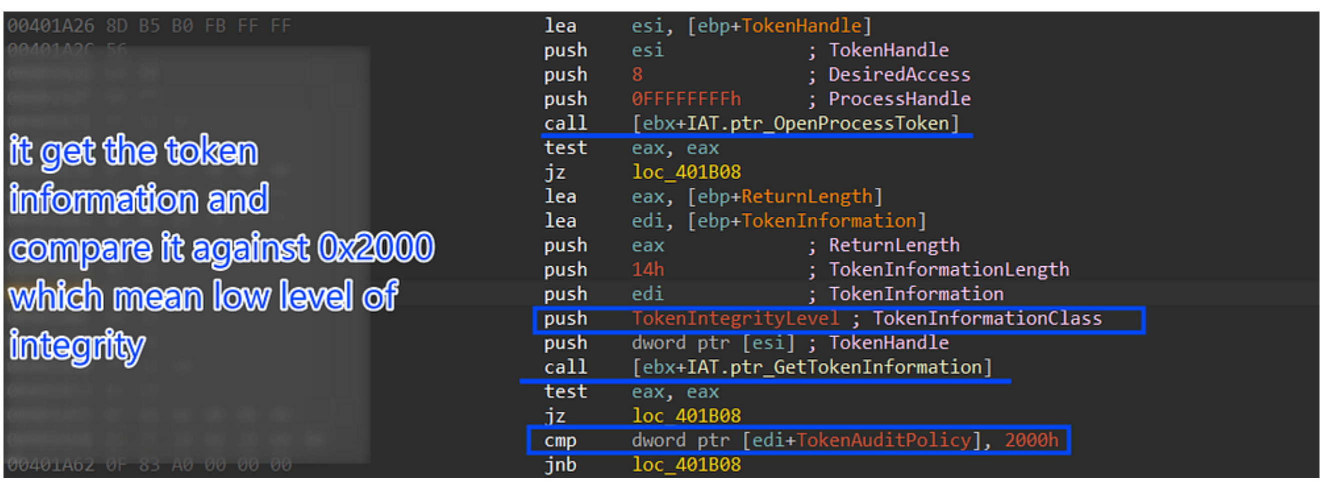


Figure 25. Getting Process Privalage

and if its integrity is under **0x2000** it will execute a command using ShellExecuteExW to run malware again under the Windows Management Instrumentation Command-line (***WMIC***)

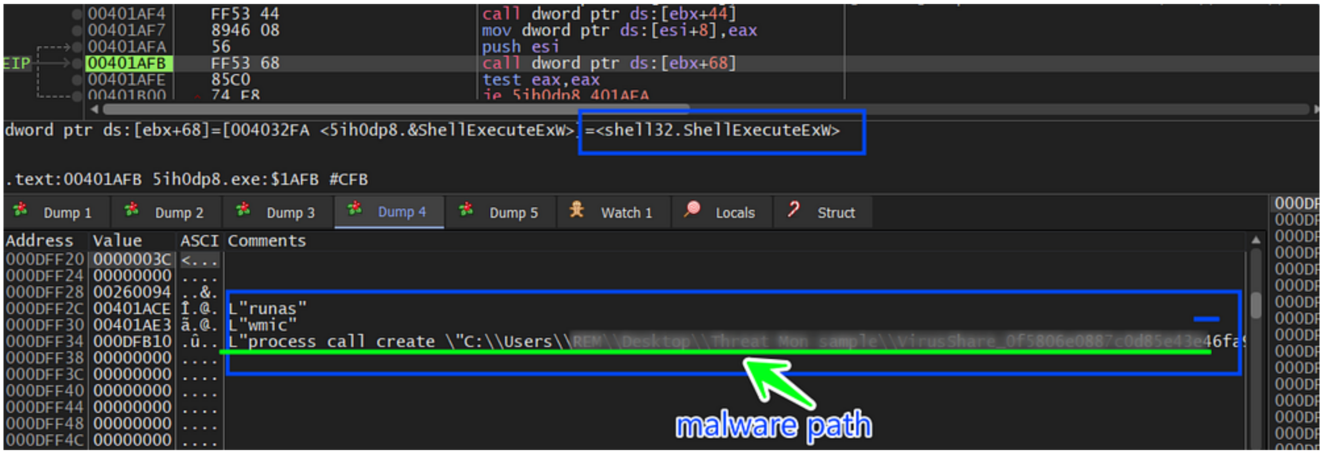


Figure 26. Executing Malware under WMIC

Anti-Debugging

then Smoke will use native **APIs** to check if it's being debugged but this time it will not do it through PEB or using APIs like check **Isdebuggerpresent**(), instead it will execute a call to **NtQueryInformationProcess**() using **ProcessDebugPort =7 as** an information class that Retrieves a **DWORD_PTR** value that is the port number of the debugger for the process. A nonzero value indicates that the process is being run under the control of a **ring 3** debugger.

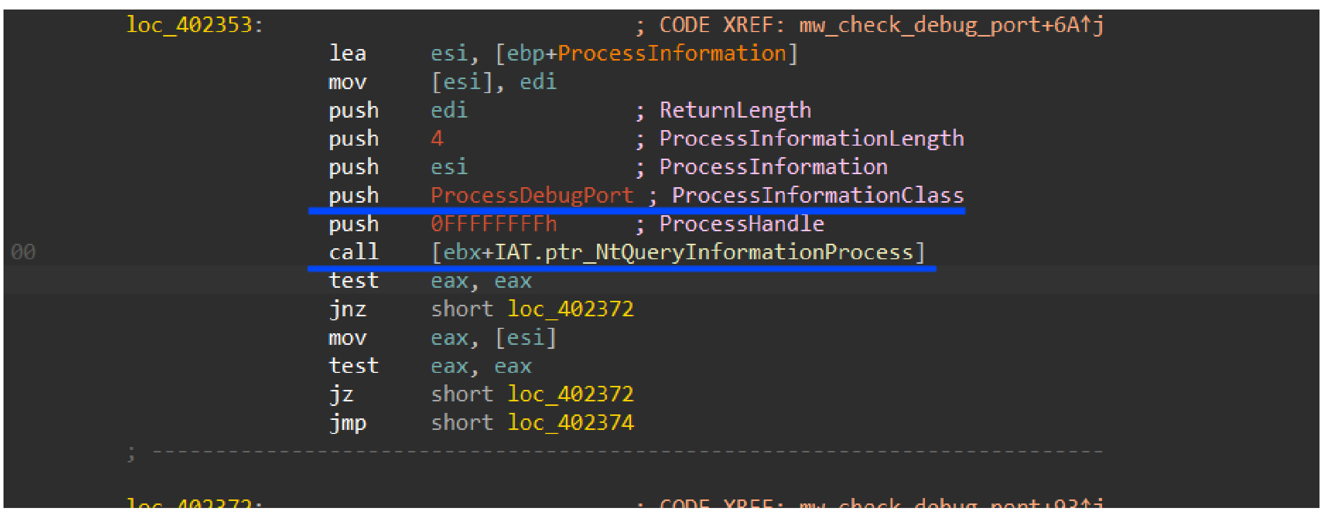


Figure 27. Checking Debugger existence using native API

if it finds that the malware is being debugged it will terminate the process.

> **note*** as I said before the malware decrypts the code and then re-encrypts it again, but sometimes it embeds some strings inside the decrypted code which prevents IDA from identifying this code as a separate function, imagine that instructions then strings then instructions in the same blob, to summarize that the strings exist in the text section inside the encrypted code and Smoke got access to it by calling the next instruction below the strings which places the address of the string in the top of the stack.

Check AVs & Virtualization

Smoke will go through all loaded modules in the victim machine and and for every module it will compare its name against some of the modules used by famous Anti-virus solutions

> ***sbiedll → Sandboxie Environment***
> ***aswhook → Avast Anti-virus***
> ***snxhk → Avast Anti-virus***

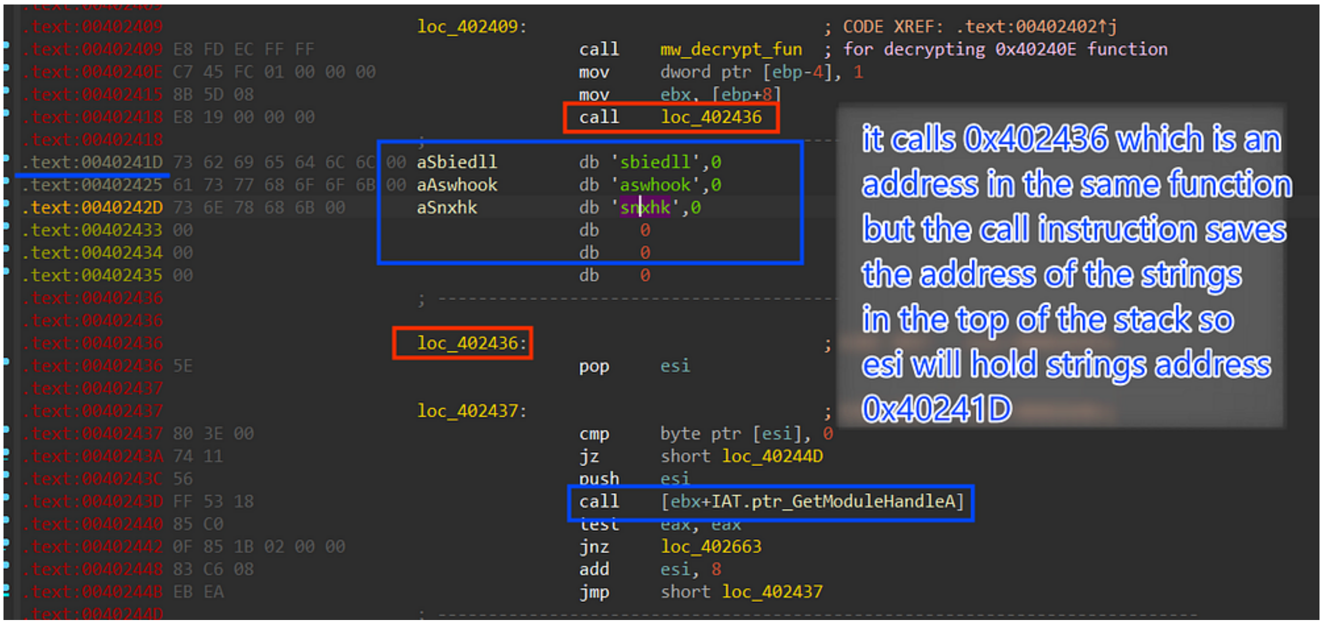


Figure 28. Comparing Modules Names to check AVs Existence

then it will enumerate all subkeys under these two keys which are related to disk drivers in a virtual environment

> **Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\SCSI**
> **Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\IDE**

it will search for some strings inside its subkeys.

values to look for → [qemu , virtio, vmware , vbox , xen]
These strings are related to the emulation of drivers in sandboxes and virtualization environment

```
loc_401C80:                             ; CODE XREF: .text:00401C70↑j
                call    sub_401CCC
; ---------------------------------------------------------------------------
aQemu:
00+             text "UTF-16LE", 'qemu',0
                db    0
                db    0
                db    0
                db    0
aVirtio:
00+             text "UTF-16LE", 'virtio'
                db    0
                db    0
aVmware:
00+             text "UTF-16LE", 'vmware'
                db    0
                db    0
aVbox:
00+             text "UTF-16LE", 'vbox',0
                db    0
                db    0
                db    0
                db    0
aXen:
00              text "UTF-16LE", 'xen',0
                db    0
                db    0
                db    0
                db    0
                db    0
                db    0
                db    0
```

Figure 30. embedded Disk Driver names related to VM emulation

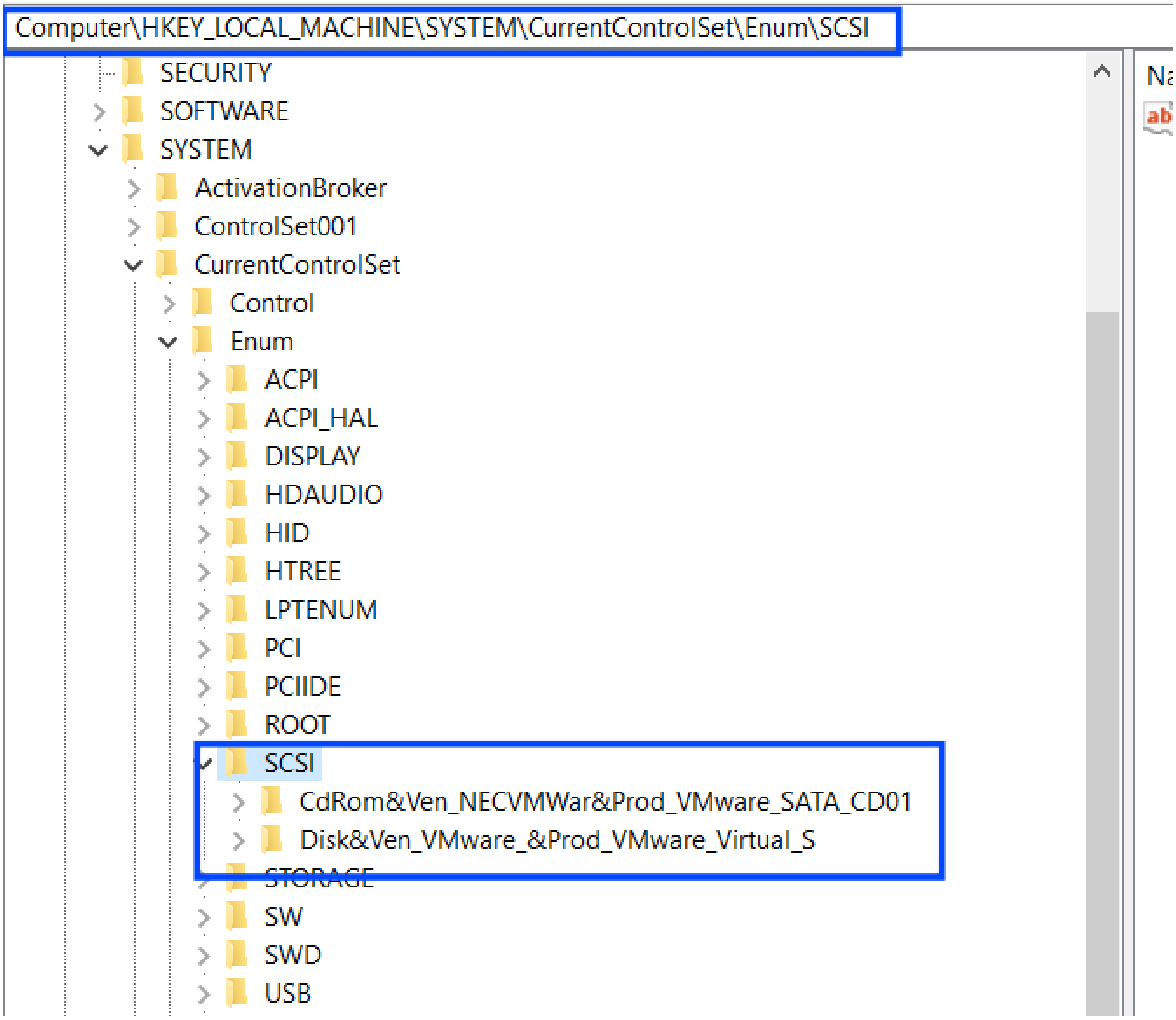


Figure 31. keys to search within

using **NtQuerySystemInformation()** API and placing SystemProcessInformation as a class information type it will Return an array of **SYSTEM_PROCESS_INFORMATION** structures, one for each process running in the system.

```
                mov     [ebp-0Ch], eax
                lea     edx, [ebp-8]
                push    edx
                push    dword ptr [ebp-8]
                push    dword ptr [ebp-0Ch]
                push    SystemProcessInformation
00              call    [ebx+IAT.ptr_NtQuerySystemInformation]
                test    eax, eax
00              jnz     loc_402201
                mov     edi, [ebp-0Ch]

loc_4020D1:                             ; CODE XREF: .text:004021FC↓j
```

Figure 32. Retrieving process name using **NtQuerySystemInformation**

then it will compare process names against some of the background processes used by **Qemu, Vmware, and Virtualbox** environments

```
qemu-ga.exe → Qemu
qga.exe → Qemu
windanr.exe
vboxservice.exe →Vbox
vboxtray.exe →Vbox
vmtoolsd.exe →Vmware
prl_tools.exe →System Explorer
```

then it will give a call to the same **API** but with **SystemModuleInformation** as an information class which returns **RTL_PROCESS_MODULES** structure that stores information about **loaded drivers**, so it compares driver name against some embedded drivers names that exist in virtual environments

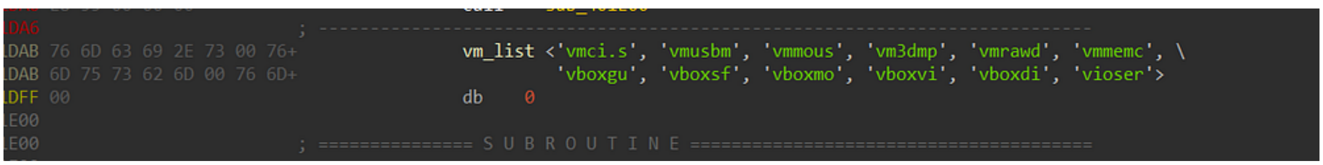


Figure 31. embedded drivers names

```
vmci.s     vmmemc    vboxvi
vmusbm     vboxgu    vboxdi
vmmous     vboxsf    viose
vm3dmp     vboxmo    vmrawd
```

Stage 3 Decryption

After passing all checks, Smoke will start loading the third stage.

it first will check the **Architecture** of the victim machine to determine the appropriate payload, there are 2 payloads one for **x86** and the other for **x64,** so it checks the value of **GS** or **Segment Register** which will be 0 if the process is running in **x86** pc but in **x64** system it will contain a positive value.

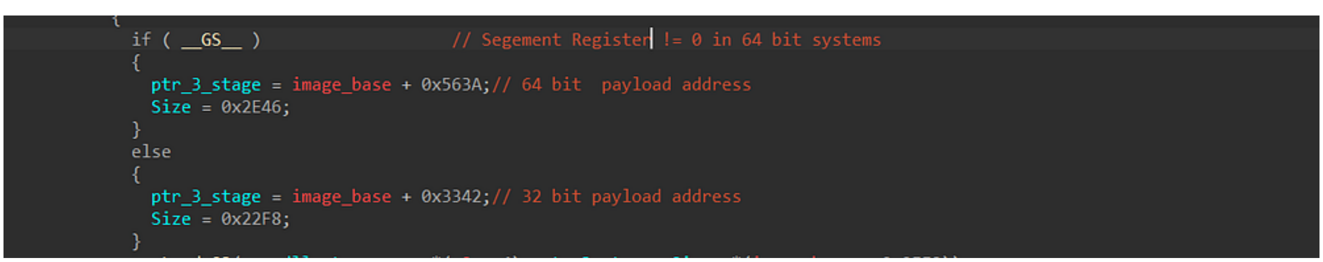


Figure 34. Checking windows Architecture

then it will decrypt the payload at the chosen address using the same decryption routine used for **hashes decryption** but with simple additions this time because it is using the Dword value as **Xor key**, he needs to decrypt the payload **dword by dword**, but what if the payload size is not a multiple of 4 (Dword size = 4 bytes) so it will result in a wrong decrypted value at the last (3 or 2 or 1) bytes, to fix this it will get the reminder value after decrypting with a dword value as xor key and then decrypt the reminder bytes with 1 byte as a xor key

```
ptr_payload_encrypted = a2;
ptr_payload_decrypted = a2;
counter = size >> 2;
do
{
  paylaod_Dword_value = *ptr_payload_encrypted;// adding 4 which is the size of DWORD
  ptr_payload_encrypted += 4;
  *ptr_payload_decrypted++ = paylaod_Dword_value ^ 0x880BD3F6;// deecrypting using Dword xor Key
  --counter;
}
while ( counter );
i = size & 3;                              // getting the last bytes which will be 1,2 or 3 bytes
if ( (size & 3) != 0 )
{
  do
  {
    payload_byte_value = *ptr_payload_encrypted++;
    *ptr_payload_decrypted = payload_byte_value ^ 0xF6;// decrypting using 1 Byte as Xor key
    ptr_payload_decrypted = (ptr_payload_decrypted + 1);
    --i;
  }
  while ( i );
}
```

Figure 35. Decrypting the payload with attention to its size

we do it statically by writing a script to decrypt this payload u can check it here

```
def xor_chunk_s3( data, dword_key, b_key):
 decrypted=b''

 for i in range(0,(len(data)//4)*4,4):

 _4_bytes= struct.unpack("<I",data[i:i+4])[0]

 xor_result = _4_bytes ^ dword_key

 decrypted+=struct.pack("<I",xor_result)

 last_bytes_len = len(data)%4

 if last_bytes_len > 0:

 last_decrypted=[]

 for byte in data[-last_bytes_len:]:

 last_decrypted.append(byte ^ b_key)

 print(last_decrypted)

 decrypted+=bytes(last_decrypted)
 return decrypted
```

Stage 3 Decompression

after decrypting the payload it will use the first 4 bytes as size that is used on **NtAllocateVirtualMemory**() API with **read_write** permission, then the pointer to the allocated memory and the decrypted payload are pushed to another anonymous function which after some research for this function using some const assembly instruction to identify it because it was not a decryption routine or whatever and also something that proves that this function is responsible for decompression is that the allocated size is larger than the decrypted data size which paves the way for a decompression operation that will happen in the allocated region

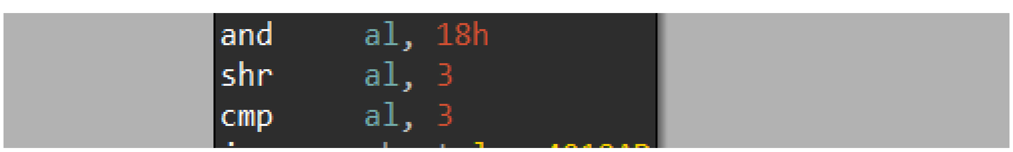


Figure 36. Code Chunk for Decompression routine

these assembly instructions give me a hint about the used algorithm which is LZSA2, an old compression algorithm used for old CPUs according to this **Blog**

```
mw_xor_decrypt(decrypted_payload, decrypted_payload, size);
decompressed_payload_size = *decrypted_payload;
RegionSize = decompressed_payload_size;
if ( (a2->ptr_NtAllocateVirtualMemory)(0xFFFFFFFF, &BaseAddress, 0, &RegionSize, 0x3000u, PAGE_READWRITE)
  || (mw_LZSA2_decompression(decrypted_payload + 4, decrypted_payload + 4, BaseAddress),
      v5 != decompressed_payload_size) )
{
  BaseAddress = 0;
}
```

Figure 37. The function responsible for LZSA2 decompression

so from another [GitHub repo](), we found a C implementation for this algorithm, cloned it, and then built the project

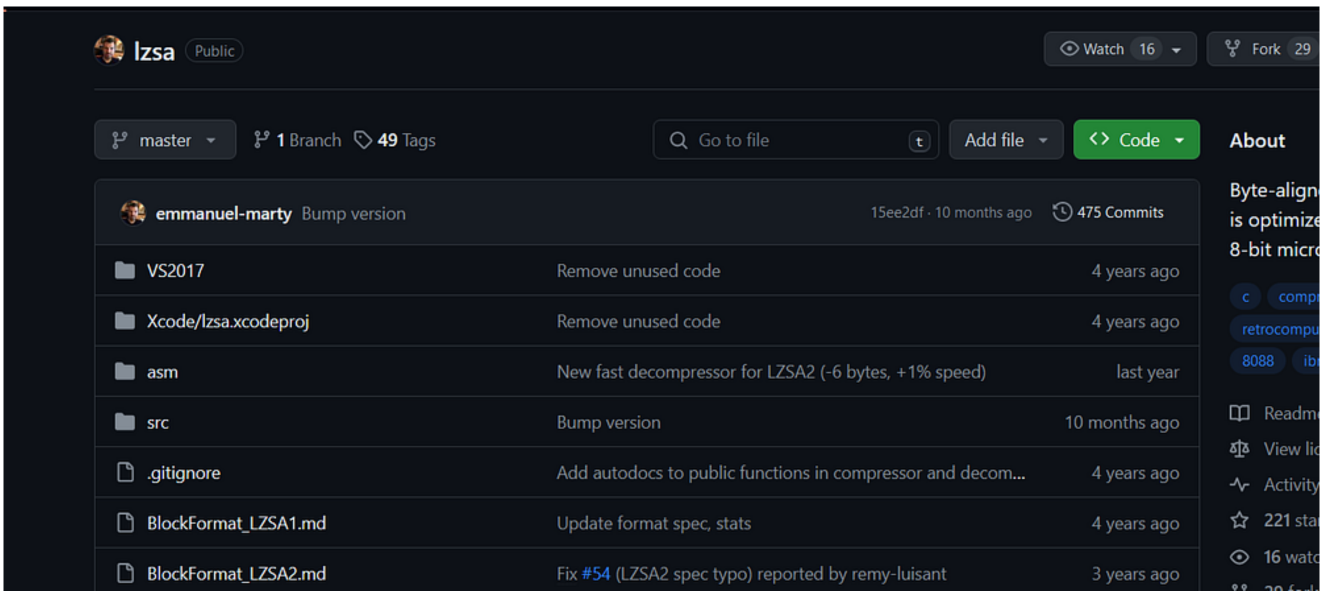


Figure 38. LZSA repo, Big Thanks to him

and here is the used command to decompress the decrypted payload

lzsa_debug.exe -d -r -f 2 decrypted_payload.bin decrypted_decompress.bin

Stage 3 Injection:

then after decompression, Smokeloader will start injecting this destroyed stage cause we got a PE file without headers, so to do it in Regular steps

1- It gets a handle for Explorer.exe by executing a call to **GetShellWindow()**Retrieves a handle to Shell's desktop window, in our case it's Explorer.exe, and then it gets a handle to this process using **GetWindowThreadProcessId()**

```
window_handle =GetShellWindow)();
 if ( window_handle )
 break;
 (a2->ptr_Sleep)(0x3E8u);

dwProcessId[1] = window_handle;
 v6 = dwProcessId;
 dwProcessId[0] = 0;
 (GetWindowThreadProcessId)(window_handle, dwProcessId);
```

2- it then gets a token handle to **explorer.exe** using **NtOpenProcess()** and duplicates this handle to use it later

3-It then creates a section with **PAGE_READWRITE** permission and then maps this section to the current **malware process** and **Explorer.exe** process using **NtCreateSection()** and **NtMapViewOfSection()** APIs

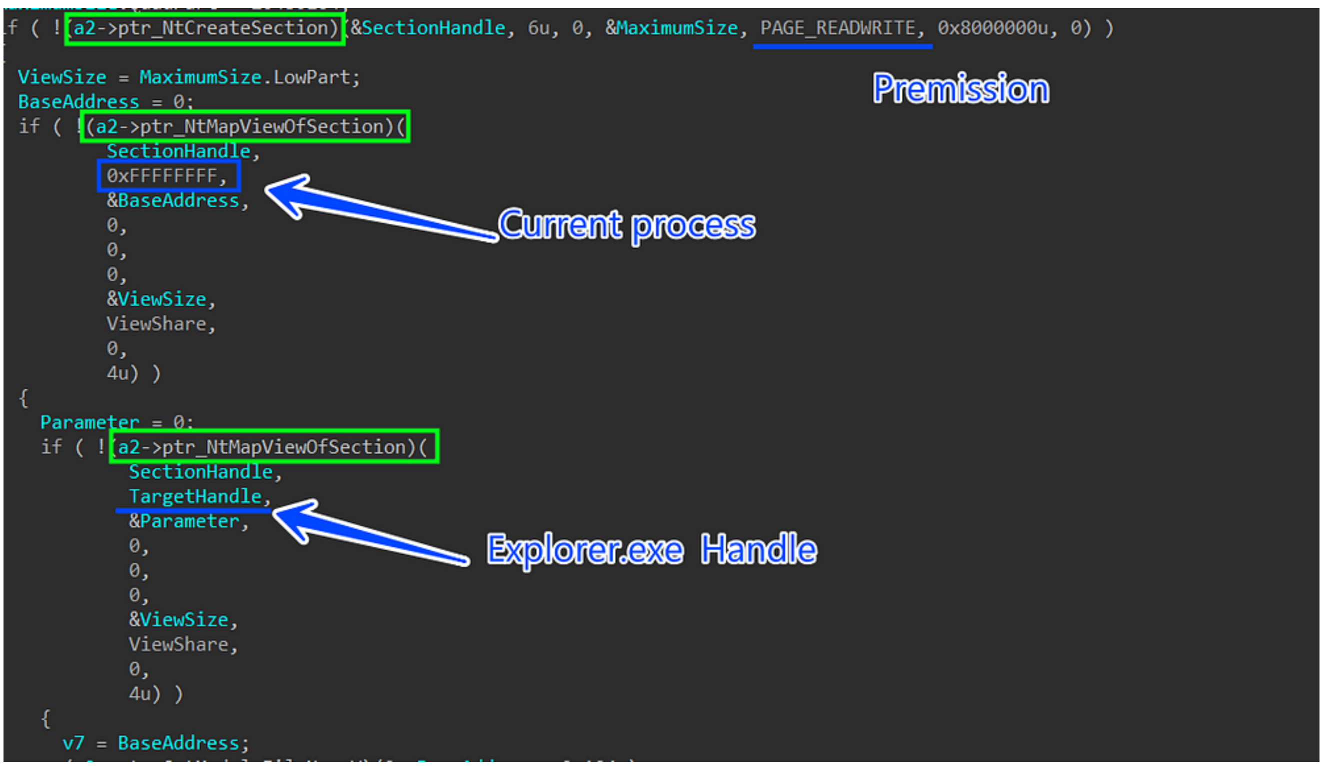


Figure 39. Creating and Mapping sections

4- Create another section but this time with a different permission **PAGE_EXECUTE_READWRITE,** and map this section to the current process and **explorer.exe.**

```
v6 = &section_handle;
if ( !(a2->ptr_NtCreateSection)(&section_handle, 0xEu, 0, &MaximumSize, PAGE_EXECUTE_READWRITE, 0x8000000u, 0) )
{
  if ( v39 )
  {
    ViewSize = MaximumSize.LowPart;
    v29 = 0;
    if ( !(a2->ptr_NtMapViewOfSection)(section_handle, 0xFFFFFFFF, &v29, 0, 0, 0, &ViewSize, ViewShare, 0, 4u) )
    {
      v31 = 0;
      if ( !(a2->ptr_NtMapViewOfSection)(
              section_handle,
              TargetHandle,                  // Explorer.exe Handle
              &v31,
              0,
              0,
              0,
              &ViewSize,
              ViewShare,
              0,
              0x20u) )
      {
```

Figure 40. Mapping sections to explorer.exe

5- it then hashes the encrypted payload not the decompressed only to check integrity but it is worth mentioning.

```
encrypted_payload = &byte_40563A;
 payload_size = 0x2E46;
 hash_value = 0x2260;
 do
 {
 v11 = *encrypted_payload++;
 hash_value = v11 + 33 * hash_value;
 --payload_size;
 }
 while ( payload_size );
```

6- it next copies the decompressed payload into the mapped section and then builds **IAT** for this payload, then it creates a new thread into **Explorer.exe** using **RtlCreateUserThread()** and **pushes** the address of payload in **explorer.exe** memory as a **StartAddress** argument for this API call.

```
        }
        v6 = v24;
        StartAddress = (v31 + *(v24 + 10));
        ThreadHandle = 0;
        (a2->ptr_RtlCreateUserThread)(TargetHandle, 0, 0, 0, 0, 0, StartAddress, Parameter, &ThreadHandle, 0);
      }
    }
  }
```

Figure 41. Creating a threat into **explorer.exe** with payload address as its entry point

Stage 3 configuration:

After extracting the third stage file which is a destroyed **PE** file without headers, this time I have 2 options

1. fixing the file, I found a good walkthrough to do in this blog, or

2. analyzing the binary inside explorer process which was very annoying cause explorer.exe handles many things and debugging it may force something to crash

so I decompressed the file as I said before and found that, malware configuration is saved in a string table, encrypted using **RC4**

and smoke is saving it like a key and then the length of the next string and then the length of the next string, etc…until the end of the encrypted data,

so I have written a simple script that can handle this and give us the decrypted config

```
dump= binascii.unhexlify(dump)
index = 0
key =0x246FC425
while index < len(dump):
 enc_length = str_data[index]
 x = rc4crypt(dump[index+1:index+1+enc_length], struct.pack('<I',key))
 print(x.replace(b'\x00',b''))
 index = index+1+enc_length
```

and here is a list of the encrypted strings in my GitHub

C&C

Malware Command and control hosts are also RC4 encrypted so it decrypts in a similar way as the configuration,

```
struct Command_n_control
{
 Byte Data_length;
 DWORD XOR_Key;
 char Data[Data_length];
};
```

and here is the decrypted C2

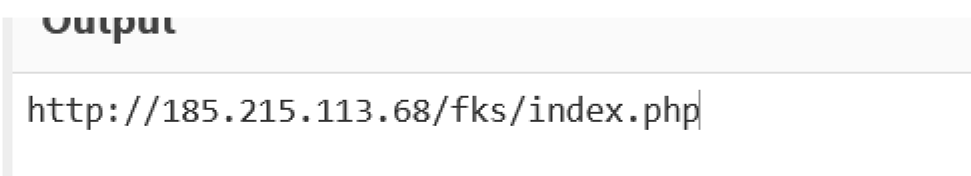


Figure 42. decrypted C2 address

the C2 is down so we don't know the next stage.

IOCs:

```
File:
Wextract file: C6BA6E91D40AA1507775077F9662ECB25C9F0943
dropped sample :B450EB89D7EA250547333228E6820A52F22BABB2

Other Hashes :
4cd9af3b630e3e06728b335c2a3a5c48297a4f36fb52b765209e12421a620fc8
daa69519885c0f9f4947c4e6f82a0375656630e0abf55a345a536361f986252e
8ecd99368b83efde6f0d0d538e135394c5aec47faf430e86c5d9449eb0c9f770
ab2c8fb5e140567a6e8e55c89138d5faa0ef5e6f2731be3c30561a8ce9e43d29
60c65307f80b12d2a8d8820756e90021419a1fcfcda18cdbee3a25974235ac

CnC:
hxxp://185.215.113.68/fks/index.php
hxxp://rixoxeu9.top/game.exe
hxxp://planilhasvbap.com.br/wp-admin/js/k/index.php
hxxp://telegatt.top/agrybirdsgamerept
hxxp://95.217.43.206/
```

you can find the full repo that contains all scripts ***here***

References

Deep Analysis of SmokeLoader
_SmokeLoader is a well known bot that is been around since 2011. It's mainly used to drop other malware families…_n1ght-w0lf.github.io

Windows Process Injection: PROPagate
_Introduction In October 2017, Adam at Hexacorn published details of a process injection technique called PROPagate. In…_modexp.wordpress.com

SmokeLoader Triage
_Taking a look how Smoke Loader works_research.openanalysis.net

SmokeLoader | dcd883af6eb9
_This feature requires an online-connection to the VMRay backend. An offline version with limited functionality is also…_www.vmray.com