An In-Depth analysis of the new Taurus Stealer

May 7, 2021

Research & Threat Intel 07 May 2021

Written By Alberto Marin Blueliv's Malware Sandbox Lead

Introduction

Taurus Stealer, also known as Taurus or Taurus Project, is a C/C++ information stealing malware that has been in the wild since April 2020. The initial attack vector usually starts with a malspam campaign that distributes a malicious attachment, although it has also been seen being delivered by the Fallout Exploit Kit. It has many similarities with Predator The Thief at different levels (load of initial configuration, similar obfuscation techniques, functionalities, overall execution flow, etc.) and this is why this threat is sometimes misclassified by Sandboxes and security products. However, it is worth mentioning that Taurus Stealer has gone through **multiple updates** in a short period and is actively being used in the wild. Most of the changes from earlier Taurus Stealer versions are related to the **networking** functionality of the malware, although other changes in the obfuscation methods have been made. In the following pages, we will **analyze in-depth** how this new Taurus Stealer version works and compare its main changes with previous implementations of the malware.

Underground information

The malware appears to have been developed by the author that created **Predator The Thief,** “Alexuiop1337”, as it was promoted on their Telegram channel and Russian-language underground forums, though they claimed it has no connection to Taurus. Taurus Stealer is advertised by the threat actor “**Taurus Seller**” (sometimes under the alias “Taurus_Seller”), who has a presence on a variety of Russian-language underground forums where this threat is primarily sold. The following figure shows an example of this threat actor in their post on one of the said forums:

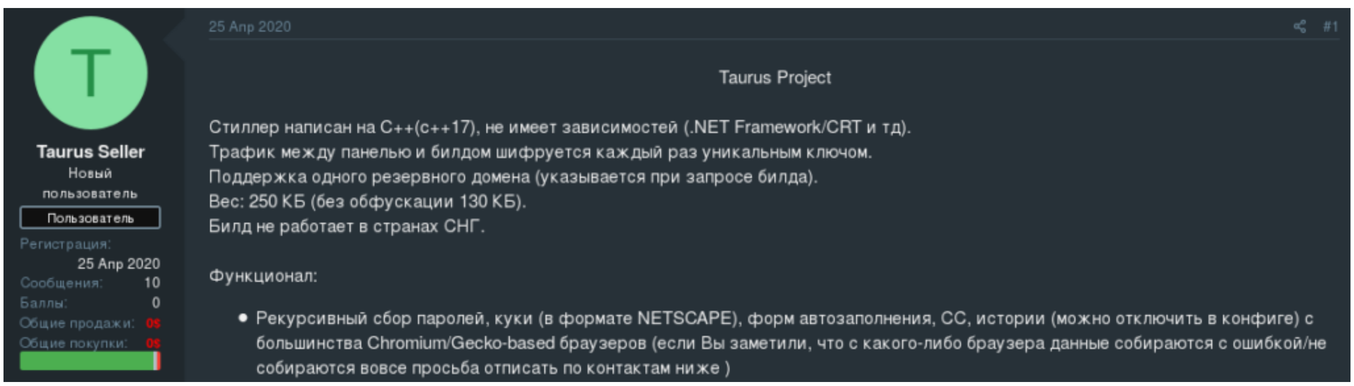

Figure 1. Taurus Seller post in underground forums selling Taurus Stealer

The initial description of the ad (translated by Google) says:

Stiller is written in C ++ (c ++ 17), has no dependencies (.NET Framework / CRT, etc.).

The traffic between the panel and the build is encrypted each time with a unique key.

Support for one backup domain (specified when requesting a build).

Weight: 250 KB (without obfuscation 130 KB).

The build does not work in the CIS countries.

Taurus Stealer sales began in April 2020. The malware is inexpensive and easily acquirable. Its price has fluctuated somewhat since its debut. It also offers temporal discounts (20% discount on the eve of the new year 2021, for example). At the time of writing this analysis, the prices are:

Concept	Price
License Cost – (lifetime)	150 $
Upgrade Cost	0 $

Table 1. Taurus Stealer prices at the time writing this analysis

The group has on at least one occasion given prior clients the upgraded version of the malware for free. As of January 21, 2021, the group only accepts payment in the privacy-centric cryptocurrency **Monero**. The seller also explains that the license will be lost **forever** if any of these rules are violated (ad translated by Google):

- It is forbidden to scan the build on VirusTotal and similar merging scanners
- It is forbidden to distribute and test a build without a **crypt**
- It is forbidden to transfer project files to third parties
- It is forbidden to insult the project, customers, seller, coder

This explains why most of Taurus Stealer samples found come packed.

Packer

The malware that is going to be analyzed during these lines comes from the packed sample 2fae828f5ad2d703f5adfacde1d21a1693510754e5871768aea159bbc6ad9775, which we had successfully detected and classified as Taurus Stealer. However, it showed some different behavior and networking activity, which suggested a new version of the malware had been developed. The first component of the sample is the **Packer**. This is the outer layer of Taurus Stealer and its goal is to hide the malicious payload and transfer execution to it in runtime. In this case, it will accomplish its purpose **without** the need to create another process in the system. The packer is written in C++ and its architecture consists of **3 different layers**, we will describe here the steps the malware takes to execute the payload through these different stages and the techniques used to and slow-down analysis.

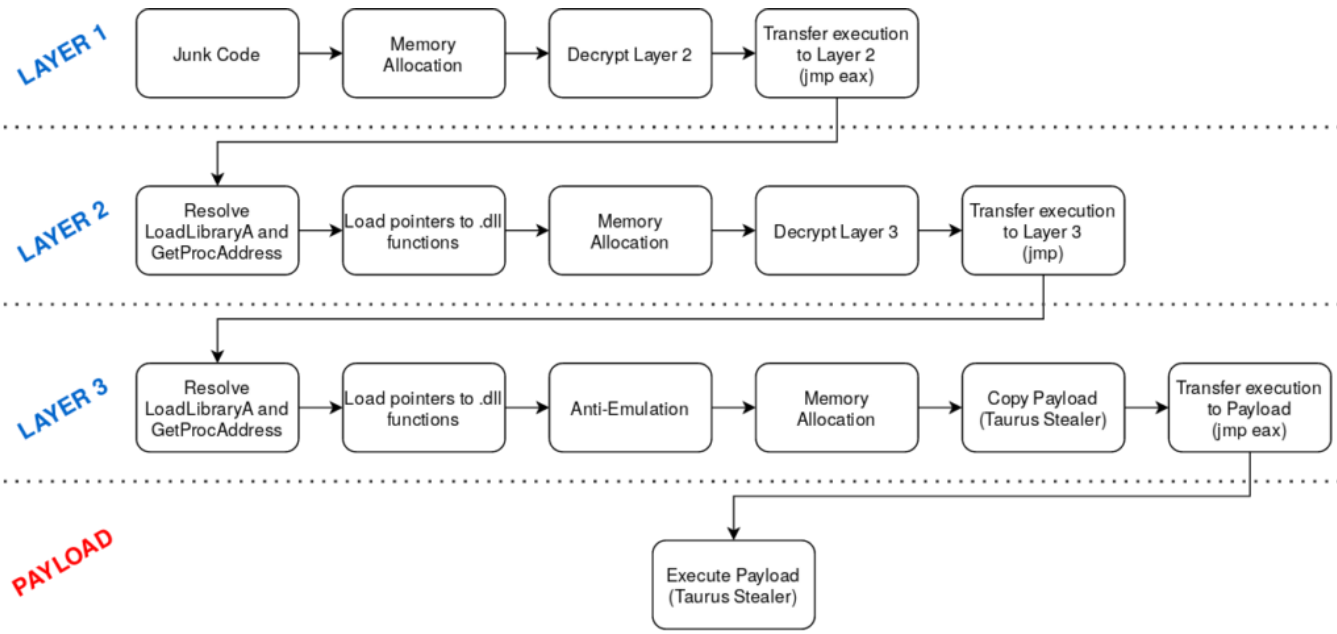

Figure 2. 2fae828f5ad2d703f5adfacde1d21a1693510754e5871768aea159bbc6ad9775 Packer layers

Layer 1 The first layer of the Packer makes use of junk code and useless loops to avoid analysis and prevent detonation in automated analysis systems. In the end, it will be responsible for executing the following essential tasks:

1. Allocating space for the Shellcode in the process's address space
2. Writing the encrypted Shellcode in this newly allocated space.
3. Decrypting the Shellcode
4. Transferring execution to the Shellcode

The initial ***WinMain()*** method acts as a wrapper using junk code to finally call the actual "main" procedure. Memory for the Shellcode is reserved using *VirtualAlloc* and its size appears hardcoded and obfuscated using an *ADD* instruction. The pages are reserved with read, write and execute permissions (*PAGE_EXECUTE_READWRITE*).

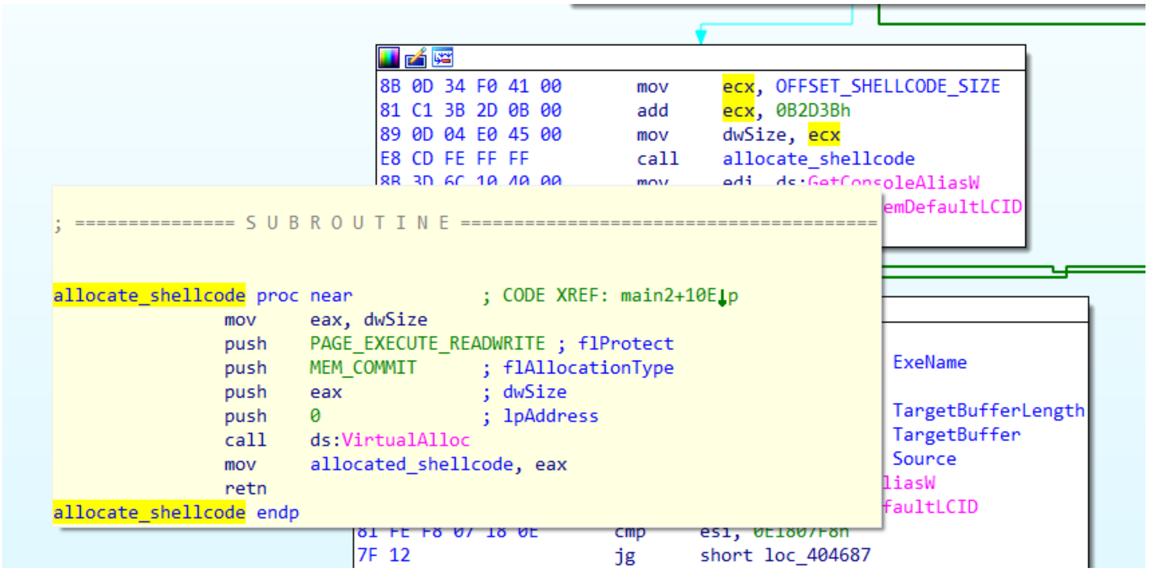

Figure 3. Memory allocation for the Shellcode

We can find the use of junk code almost anywhere in this first layer, as well as useless long loops that may prevent the sample from detonating if it is being emulated or analyzed in simple dynamic analysis Sandboxes. The next step is to load the Shellcode in the allocated space. The packer also has some hardcoded offsets pointing to the encrypted Shellcode and copies it in a loop, byte for byte. The following figure shows the core logic of this layer. The red boxes show junk code whilst the green boxes show the main functionality to get to the next layer.

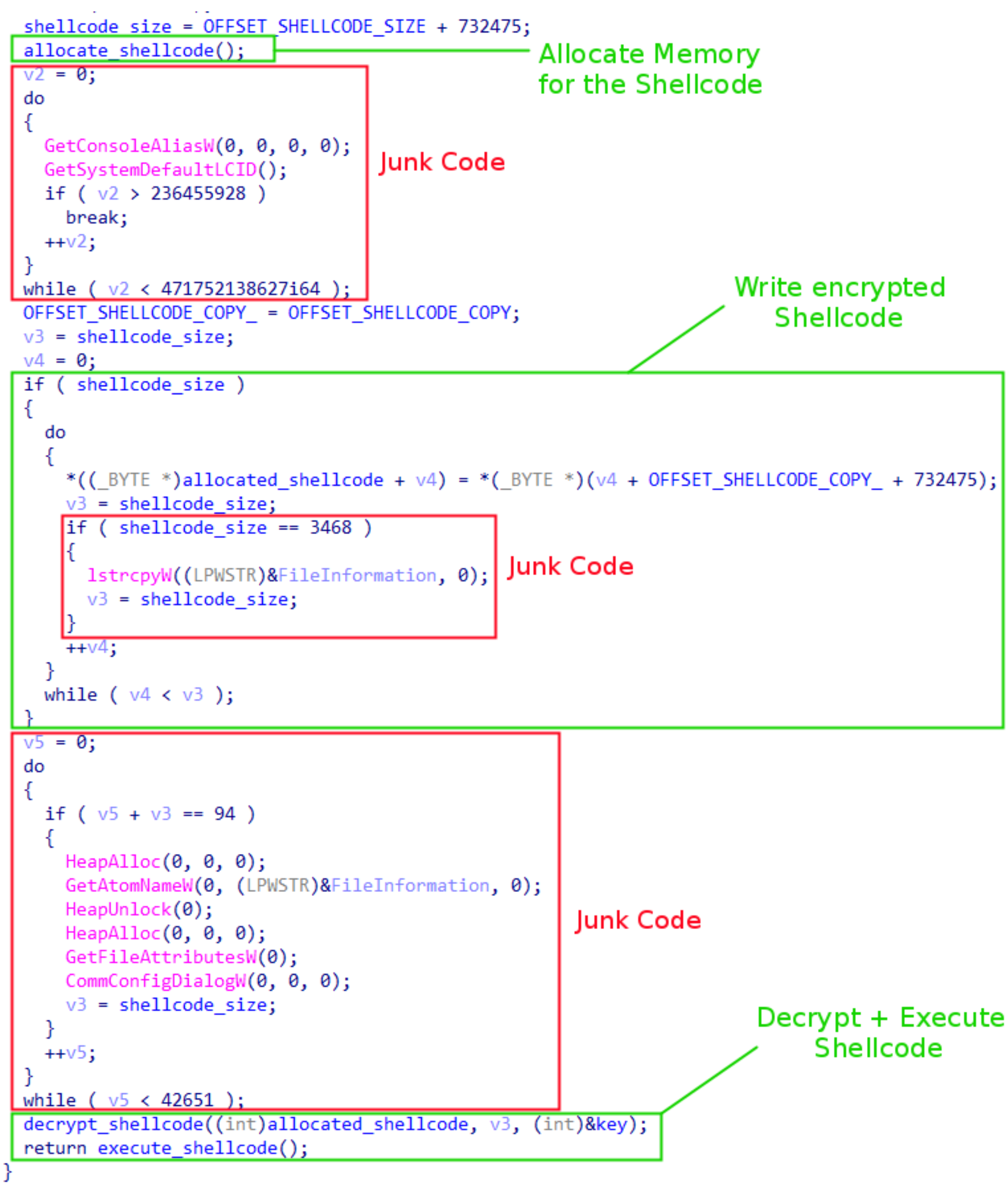

Figure 4. Core functionality of the first layer

The Shellcode is decrypted using a 32 byte key in blocks of 8 bytes. The decryption algorithm uses this key and the encrypted block to perform arithmetic and byte-shift operations using *XOR*, *ADD*, *SUB*, *SHL* and *SHR*. Once the Shellcode is ready, it transfers the execution to it using *JMP EAX*, which leads us to the second layer.

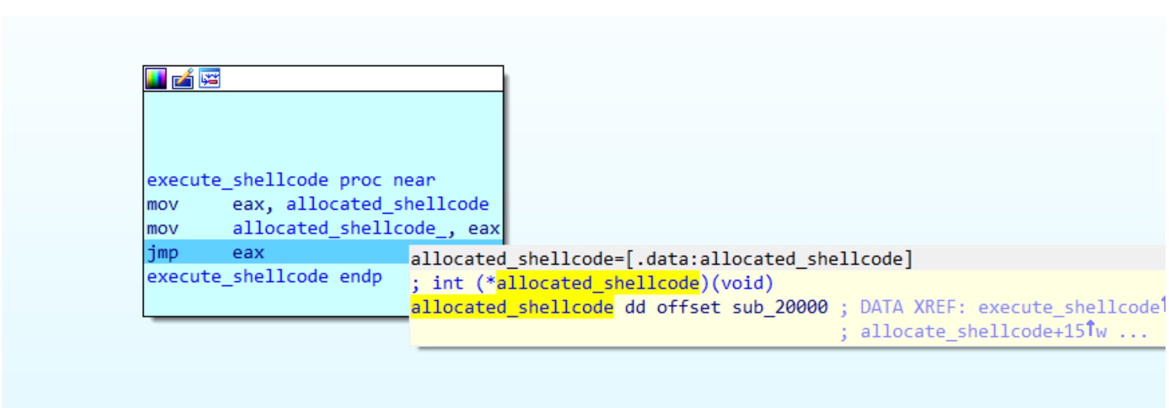

Figure 5. Layer 1 transferring execution to next layer

Layer 2 Layer 2 is a Shellcode with the ultimate task of decrypting another layer. This is not a straightforward process, an overview of which can be summarized in the following points:

1. Shellcode starts in a wrapper function that calls the **main** procedure.
2. Resolve *LoadLibraryA* and *GetProcAddress* from *kernel32.dll*
3. Load pointers to **.dll** functions
4. **Decrypt** layer 3
5. Allocate decrypted layer
6. Transfer execution using *JMP*

Finding DLLs and Functions This layer will use the TIB (Thread Information Block) to find the PEB (Process Environment Block) structure, which holds a pointer to a *PEB_LDR_DATA* structure. This structure contains information about all the loaded modules in the current process. More precisely, it traverses the *InLoadOrderModuleList* and gets the *BaseDllName* from every loaded module, **hashes** it with a custom hashing function and compares it with the respective "kernel32.dll" hash.

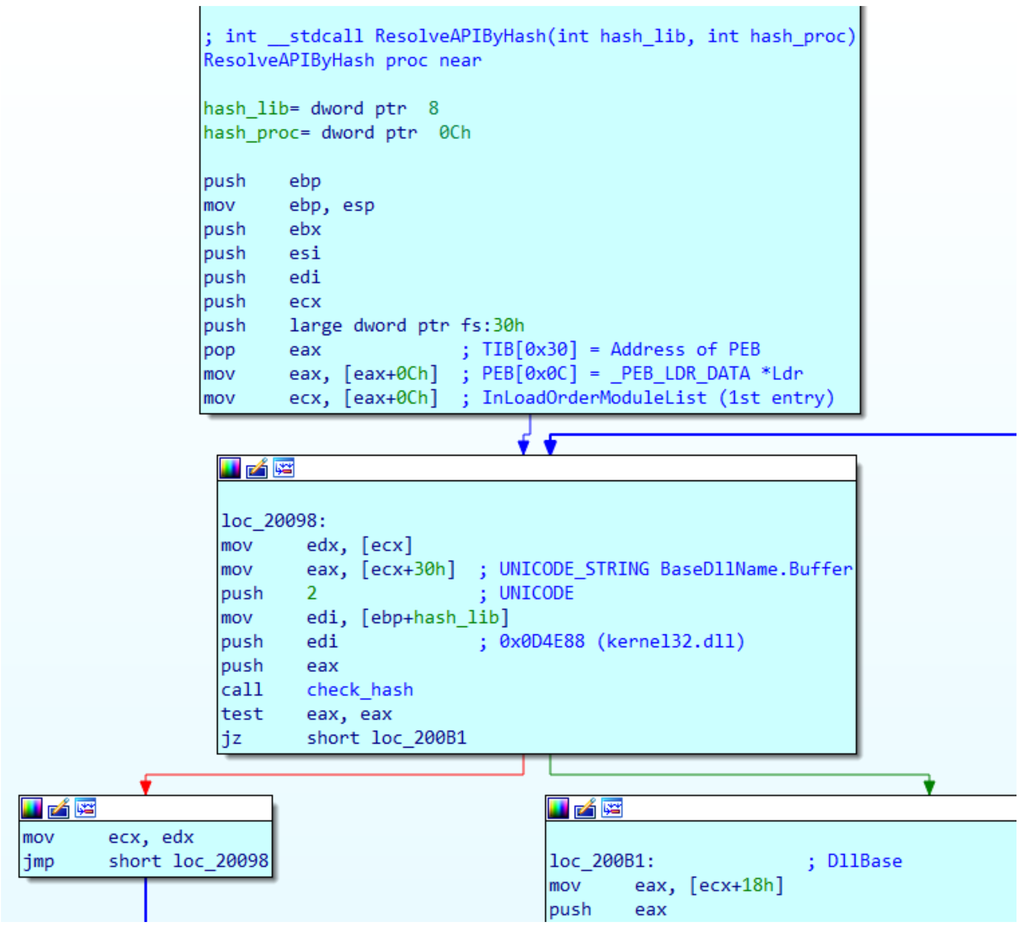

Figure 6. Traversing InLoadOrderModuleList and hashing BaseDllName.Buffer to find kernel32.dll

Once it finds "kernel32.dll" in this doubly linked list, it gets its *DllBase* address and loads the Export Table. It will then use the *AddressOfNames* and *AddressOfNameOrdinals lists* to find the procedure it needs. It uses the same technique by checking for the respective "*LoadLibraryA*" and "*GetProcAddress*" hashes. Once it finds the ordinal that refers to the function, it uses this index to get the address of the function using *AddressOfFunctions* list.

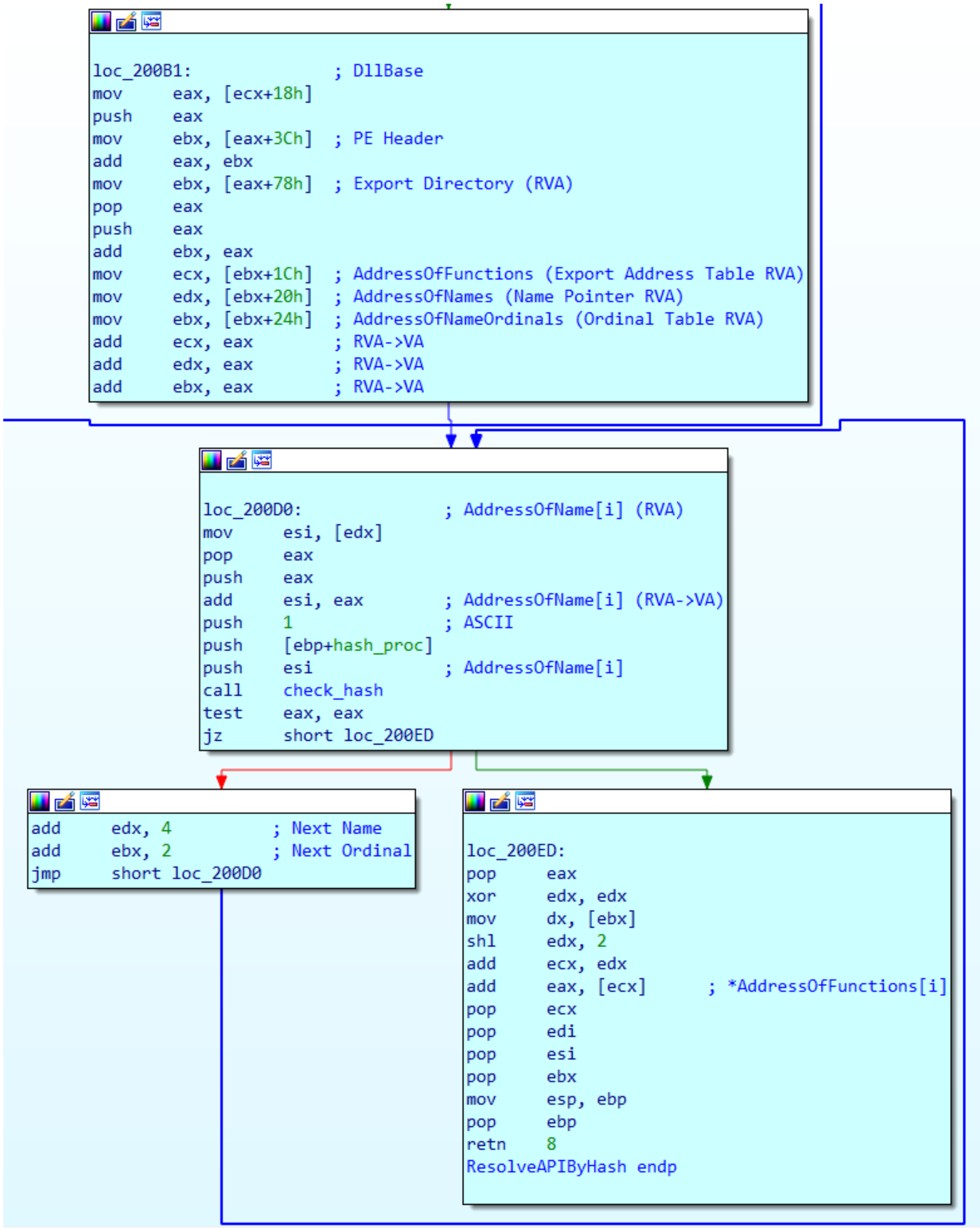

Figure 7. Resolving function address using the ordinal as an index to AddressOfFunctions list

The hashing function being used to identify the library and function names is custom and uses a parameter that makes it support both **ASCII** and **UNICODE** names. It will first use UNICODE hashing when parsing *InLoadOrderModuleList* (as it loads *UNICODE_STRING DllBase*) and ASCII when accessing the *AddressOfNames* list from the Export Directory.

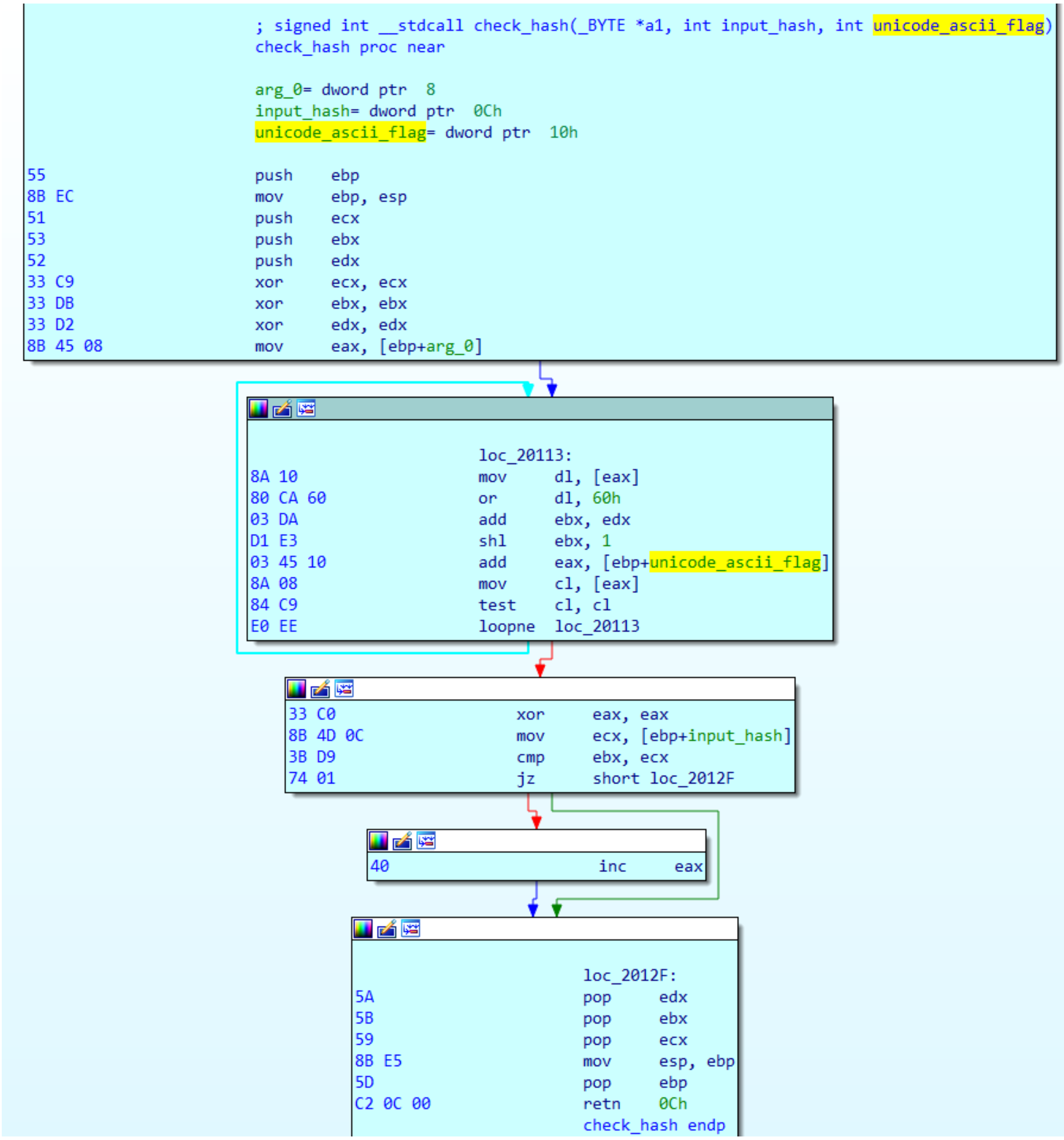

Figure 8. Custom hashing function from Layer 2 supporting both ASCII and UNICODE encodings

Once the malware has resolved *LoadLibraryA* and *GetProcAddress* from *kernel32.dll*, it will then use these functions to resolve more necessary APIs and save them in a “Function Table”. To resolve them, it relies on loading strings in the **stack** before the call to *GetProcAddress*. The API calls being resolved are:

- GlobalAlloc
- GetLastError
- Sleep
- VirtualAlloc
- CreateToolhelp32Snapshot
- Module32First
- CloseHandle

```
LoadLibraryA = ResolveAPIByHash(0xD4E88, 0xD5786);// Resolve kernel32.dll!LoadLibraryA
GetProcAddress = ResolveAPIByHash(0xD4E88, 0x348BFA);// Resolve kernel32.dll!GetProcAddress
*(_DWORD *)((char *)&loc_20048 + a1 - 131128) = LoadLibraryA;
*(_DWORD *)((char *)&loc_2004E + a1 - 131130) = GetProcAddress;
hKernel32 = 0;
v3 = 'nrek';
v4 = '23le';
v5 = 'lld.';
v6 = 0;                                       // LoadLibraryA("kernel32.dll")
hKernel32 = (*(int (__stdcall **)(int *))((char *)&loc_20048 + a1 - 131128))(&v3);
v3 = 'bolG';
v4 = 'lAla';
v5 = 'col';
v6 = 0;                                       // GetProcAddress(hkernel32, "GlobalAlloc")
*(_DWORD *)(a1 + 24) = (*(int (__stdcall **)(int, int *))((char *)&loc_2004E + a1 - 131130))(hKernel32, &v3);
v3 = 'LteG';
v4 = 'Etsa';
v5 = 'rorr';
v6 = 0;                                       // GetProcAddress(hkernel32, "GetLastError")
*(_DWORD *)(a1 + 28) = (*(int (__stdcall **)(int, int *))((char *)&loc_2004E + a1 - 131130))(hKernel32, &v3);
v3 = 'eelS';
v4 = 'p';
LOBYTE(v5) = 0;                               // GetProcAddress(hkernel32, "Sleep")
*(_DWORD *)(a1 + 32) = (*(int (__stdcall **)(int, int *))((char *)&loc_2004E + a1 - 131130))(hKernel32, &v3);
v3 = 'triV';
v4 = 'Alau';
v5 = 'coll';
v6 = 0;                                       // GetProcAddress(hkernel32, "VirtualAlloc")
*(_DWORD *)(a1 + 36) = (*(int (__stdcall **)(int, int *))((char *)&loc_2004E + a1 - 131130))(hKernel32, &v3);
```

Figure 9. Layer 2 resolving functions dynamically for later use

Decryption of Layer 3 After resolving **.dlls** and the functions it enters in the following procedure, responsible of preparing the next stage, allocating space for it and transferring its execution through a *JMP* instruction.

```
void __cdecl execute_shellcode2(FuncTable *a1)
{
  int v1; // [sp+0h] [bp-Ch]@2
  void *allocated_shellcode; // [sp+4h] [bp-8h]@2
  void *shellcode; // [sp+8h] [bp-4h]@1

  shellcode = a1->anonymous_1;
  decrypt_payload1(a1, (int)shellcode, *(_DWORD *)a1->size, *((_DWORD *)a1->size + 1));
  if ( *((_BYTE *)a1->size + 8) )
  {
    allocated_shellcode = (void *)((int (__stdcall *)(_DWORD, _DWORD, MACRO_PAGE, MACRO_PAGE))a1->kernel32_VirtualAlloc)(
                                0,
                                *(_DWORD *)((char *)a1->size + 9),
                                MEM_COMMIT,
                                PAGE_EXECUTE_READWRITE);
    v1 = 0;
    decrypt_payload2(shellcode, *(_DWORD *)a1->size, allocated_shellcode, &v1);
    shellcode = allocated_shellcode;
    *(_DWORD *)a1->size = v1;
  }
  JUMPOUT(__CS__, shellcode);
}
```

Figure 10. Decryption and execution of Layer 3 (final layer)

Layer 3 This is the last layer before having the unpacked **Taurus Stealer**. This last phase is very similar to the previous one but surprisingly less stealthy (the use of hashes to find .dlls and API calls has been removed) now **strings** stored in the **stack**, and string comparisons, are used instead. However, some previously unseen new features have been added to this stage, such as **anti-emulation** checks. This is how it looks the beginning of this last layer. The value at the address 0x00200038 is now empty but will be overwritten later with the OEP (Original Entry Point). When calling **unpack** the first instruction will execute ***POP EAX*** to get the address of the **OEP**, check whether it is already set and jump accordingly. If not, it will start the final unpacking process and then a ***JMP EAX*** will transfer execution to the final Taurus Stealer.

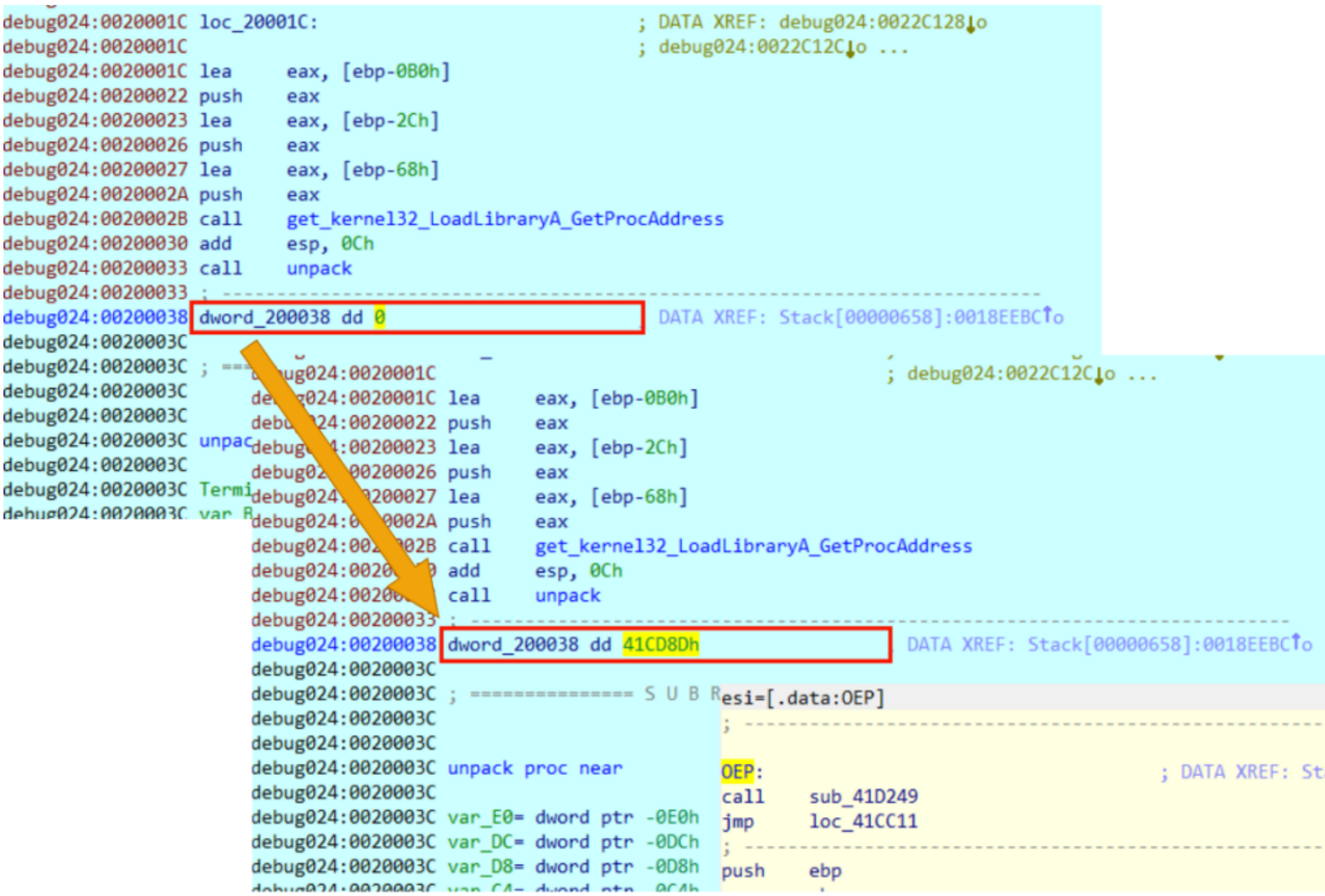

Figure 11. OEP is set. Last Layer before and after the unpacking process.

Finding DLLs and Functions As in the 2nd layer, it will parse the PEB to find *DllBase* of *kernel32.dll* walking through *InLoadOrderModuleList*, and then parse *kernel32.dll* Exports Directory to find the address of *LoadLibraryA* and *GetProcAddress*. This process is very similar to the one seen in the previous layer, but names are stored in the **stack** instead of using a custom hash function.

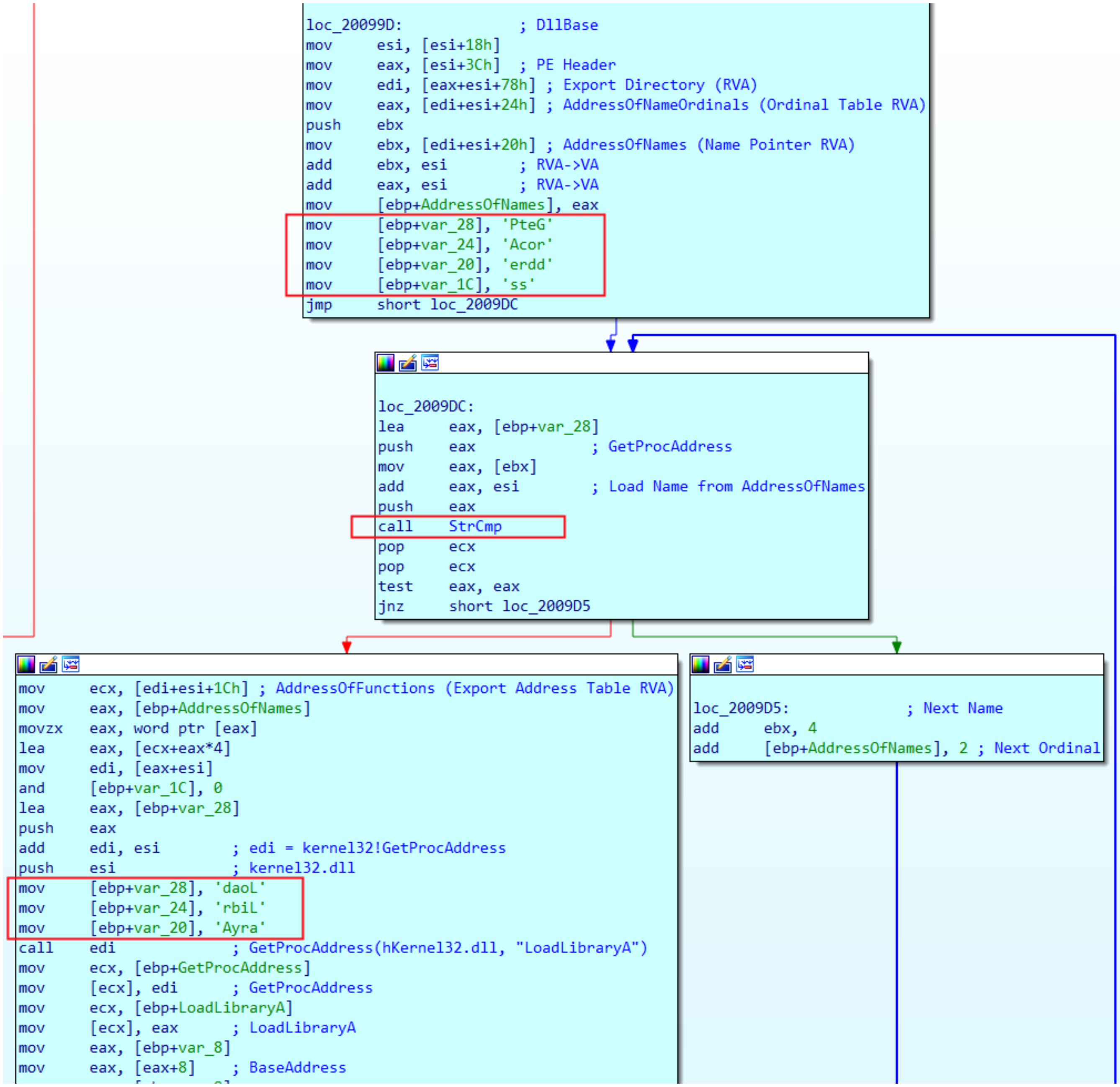

Figure 12. Last layer finding APIs by name stored in the stack instead of using the hashing approach

Once it has access to *LoadLibraryA* and *GetProcAddressA* it will start resolving needed API calls. It will do so by storing strings in the stack and storing the function addresses in memory. The functions being resolved are:

- VirtualAlloc
- VirtualProtect
- VirtualFree
- GetVersionExA
- TerminateProcess
- ExitProcess
- SetErrorMode

```
mov     dword ptr [ebp-90h], 'nrek'
mov     dword ptr [ebp-8Ch], '23le'
mov     dword ptr [ebp-88h], 'lld.'
and     dword ptr [ebp-84h], 0
lea     eax, [ebp-90h]
push    eax             ; kernel32.dll
call    [ebp+LoadLibraryA] ; LoadLibraryA
mov     [ebp+hKernel32], eax
mov     dword ptr [ebp-90h], 'triV'
mov     dword ptr [ebp-8Ch], 'Alau'
mov     dword ptr [ebp-88h], 'coll'
and     dword ptr [ebp-84h], 0
lea     eax, [ebp-90h]
push    eax             ; VirtualAlloc
push    [ebp+hKernel32] ; kernel32.dll
call    [ebp+GetProcAddress]
mov     [ebp+VirtualAlloc], eax
mov     dword ptr [ebp-90h], 'triV'
mov     dword ptr [ebp-8Ch], 'Plau'
mov     dword ptr [ebp-88h], 'etor'
mov     dword ptr [ebp-84h], 'tc'
lea     eax, [ebp-90h]
push    eax             ; VirtualProtect
push    dword ptr [ebp-3Ch] ; kernel32.dll
call    [ebp+GetProcAddress]
mov     [ebp+VirtualProtect], eax
mov     dword ptr [ebp-90h], 'triV'
mov     dword ptr [ebp-8Ch], 'Flau'
mov     dword ptr [ebp-88h], 'eer'
lea     eax, [ebp-90h]
push    eax             ; VirtualFree
push    dword ptr [ebp-3Ch] ; kernel32.dll
call    [ebp+GetProcAddress]
mov     [ebp+VirtualFree], eax
mov     dword ptr [ebp-90h], 'VteG'
mov     dword ptr [ebp-8Ch], 'isre'
mov     dword ptr [ebp-88h], 'xEno'
mov     dword ptr [ebp-84h], 'A'
lea     eax, [ebp-90h]
push    eax             ; GetVersionExA
push    dword ptr [ebp-3Ch] ; kernel32.dll
call    [ebp+GetProcAddress]
mov     [ebp+GetVersionExA], eax
```

Figure 13. Last Layer dynamically resolving APIs before the final unpack

Anti-Emulation After resolving these API calls, it enters in a function that will prevent the malware from detonating if it is being executed in an emulated environment. We‘ve named this function **anti_emulation**. It uses a common environment-based opaque predicate calling *SetErrorMode* API call.

```
int __cdecl anti_emulation(void (__stdcall *SetErrorMode)(signed int), int (__stdcall *ExitProcess)(_DWORD))
{
  int result; // eax@1

  SetErrorMode(1024);
  result = ((int (__stdcall *)(_DWORD))SetErrorMode)(0);
  if ( result != 1024 )
    result = ExitProcess(0);
  return result;
}
```

Figure 14. Anti-Emulation technique used before transferring execution to the final Taurus Stealer

This technique has been previously documented. The code calls SetErrorMode() with a known value (1024) and then calls it again with a different one. *SetErrorMode* returns the previous state of the error-mode bit flags. An emulator not implementing this functionality properly (saving the previous state), would not behave as expected and would finish execution at this point. Transfer execution to Taurus Stealer After this, the packer will allocate memory to copy the **clean** Taurus Stealer process in, parse its PE (more precisely its Import Table) and load all the necessary imported functions. As previously stated, during this process the offset 0x00200038 from earlier will be overwritten with the OEP (Original Entry Point). Finally, execution gets transferred to the unpacked Taurus Stealer via ***JMP EAX***.

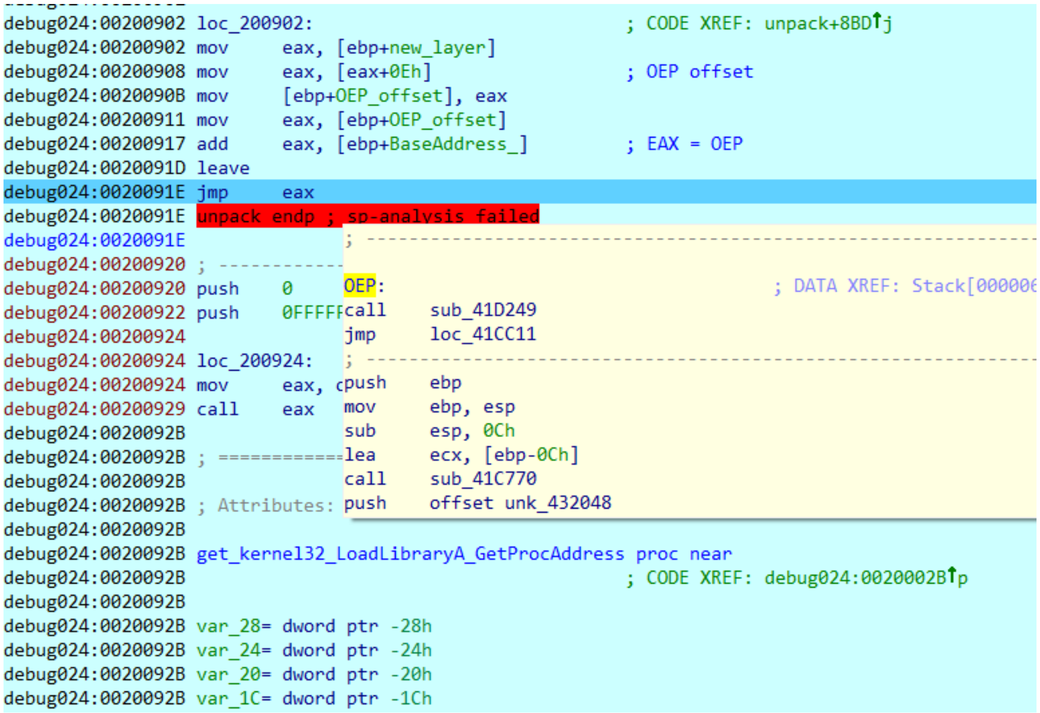

Figure 15. Layer 3 transferring execution to the final unpacked Taurus Stealer

We can **dump** the unpacked Taurus Stealer from memory (for example after copying the clean Taurus process, before the call to *VirtualFree*). We will focus the analysis on the unpacked sample with hash *d6987aa833d85ccf8da6527374c040c02e8dfbdd8e4e4f3a66635e81b1c265c8*.

Taurus Stealer (Unpacked)

The following figure shows **Taurus Stealer's** main **workflow**. Its life cycle is not very different from other malware stealers. However, it is worth mentioning that the **Anti-CIS** feature (avoid infecting machines coming from the Commonwealth of Independent States) is not optional and is the first feature being executed in the malware.

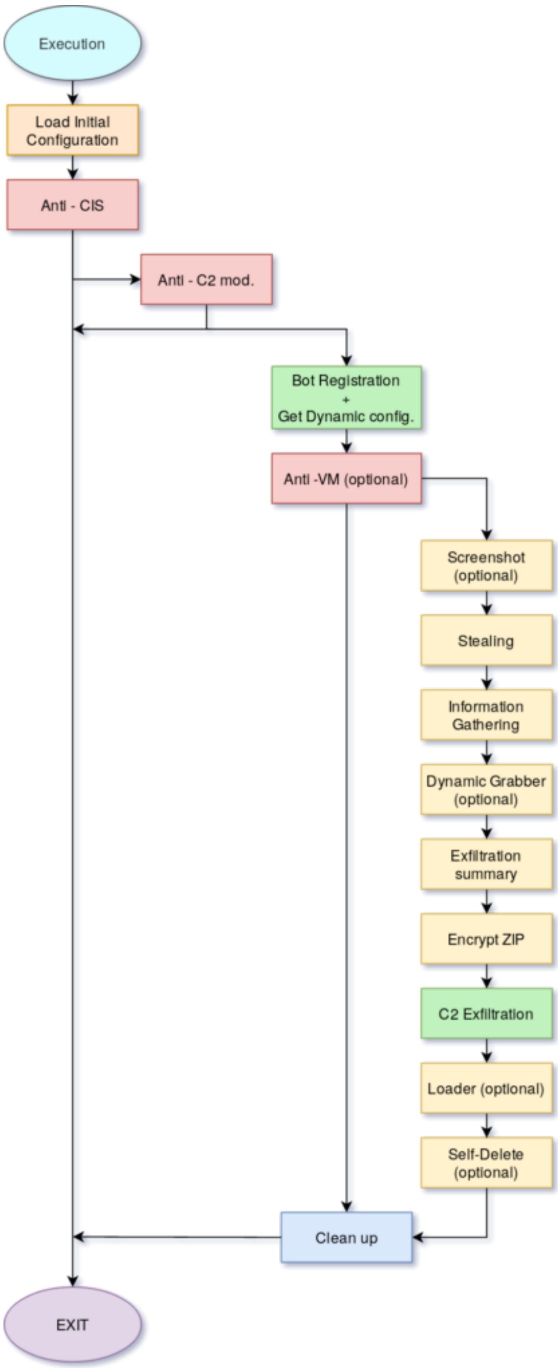

Figure 16. Taurus Stealer main workflow

After loading its initial configuration (which includes resolving APIs, Command and Control server, Build Id, etc.), it will go through two checks that prevent the malware from detonating if it is running in a machine coming from the Commonwealth of Independent States (CIS) and if it has a modified C2 (probably to avoid detonating on cracked builds). These two initial checks are **mandatory**. After passing the initial checks, it will establish communication with its **C2** and retrieve **dynamic configuration** (or a static default one if the C2 is not available) and execute the functionalities accordingly before **exfiltration**. After exfiltration, two functionalities are left: **Loader** and **Self-Delete** (both optional). Following this, a clean-up routine will be responsible for deleting strings from memory before finishing execution. **Code Obfuscation** Taurus Stealer makes heavy use of **code obfuscation** techniques throughout its execution, which translates to a lot of code for every little task the malware might perform. Taurus **string obfuscation** is done in an attempt to hide traces and functionality from static tools and to slow down analysis. Although these techniques are not complex, there is almost no single relevant string in cleartext. We will mostly find:

- XOR encrypted strings
- SUB encrypted strings

XOR encrypted strings We can find encrypted strings being loaded in the stack and decrypted just before its use. Taurus usually sets an initial hardcoded XOR key to start decrypting the string and then decrypts it in a loop. There are different variations of this routine. Sometimes there is only one hardcoded key, whilst other times there is one initial key that decrypts the first byte of the string, which is used as the rest of the

XOR key, etc. The following figure shows the decryption of the string “\Monero” (used in the stealing process). We can see that the initial key is set with ‘***PUSH*** + ***POP***’ and then the same key is used to decrypt the whole string byte per byte. Other approaches use ***strcpy*** to load the initial encrypted string directly, for instance.

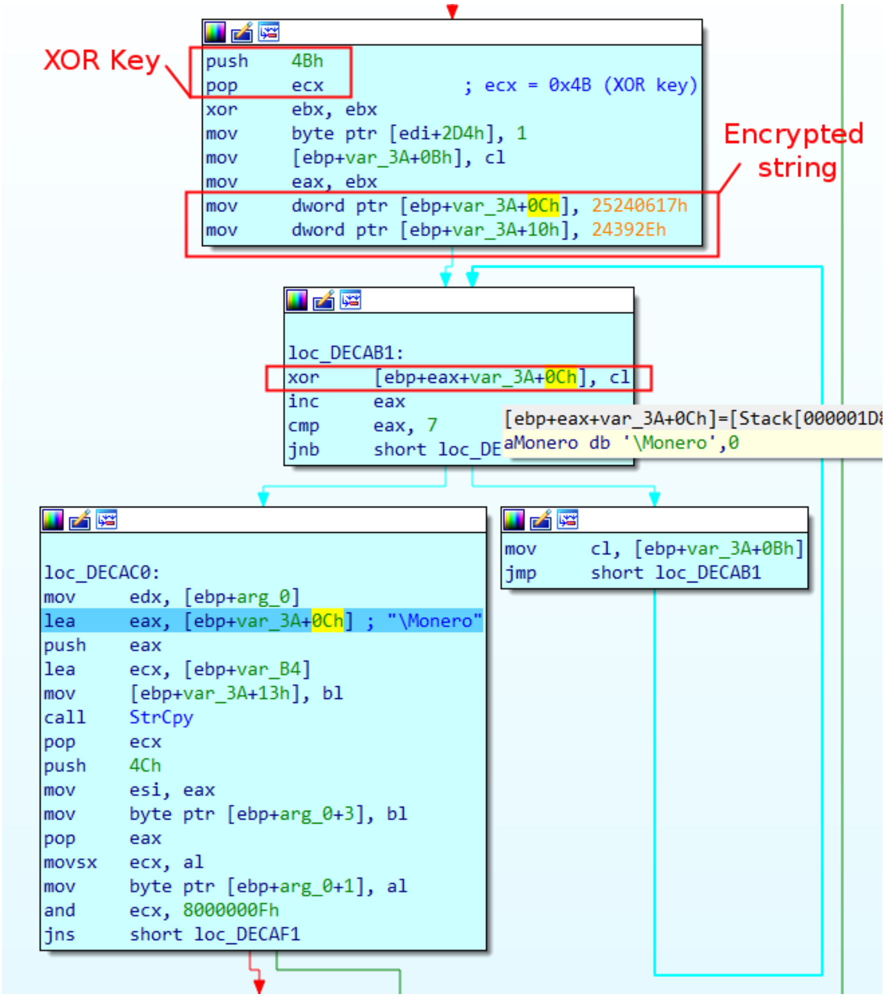

Figure 17. Example of “\Monero” XOR encrypted string

SUB encrypted strings This is the same approach as with *XOR* encrypted strings, except for the fact that the decryption is done with subtraction operations. There are different variations of this technique, but all follow the same idea. In the following example, the SUB key is found at the beginning of the encrypted string and decryption starts after the first byte.

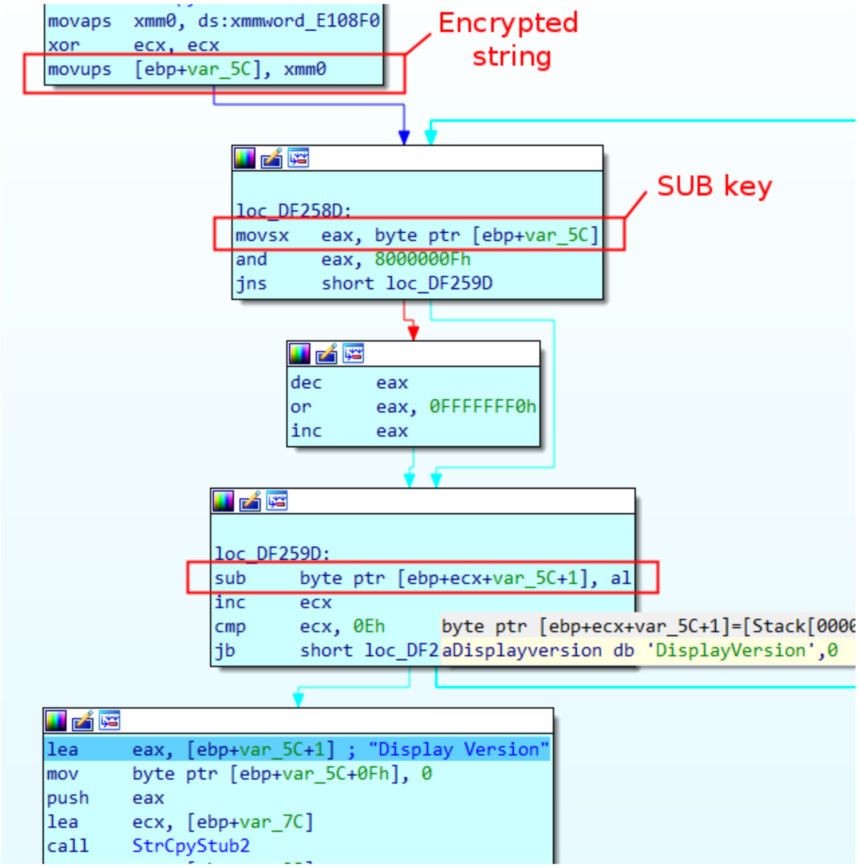

Figure 18. Example of “DisplayVersion” *SUB* encrypted string

Earlier Taurus versions made use of **stack strings** to hide strings (which can make code blocks look very long). However, this method has been completely removed by the *XOR* and *SUB* encryption schemes – probably because these methods do not show the clear strings unless decryption is performed or analysis is done dynamically. Comparatively, in stack strings, one can see the clear string byte per byte. Here is an example of such a replacement from an earlier Taurus sample, when resolving the string “wallet.dat” for *DashCore* wallet retrieval purposes. This is now done via **XOR encryption**:

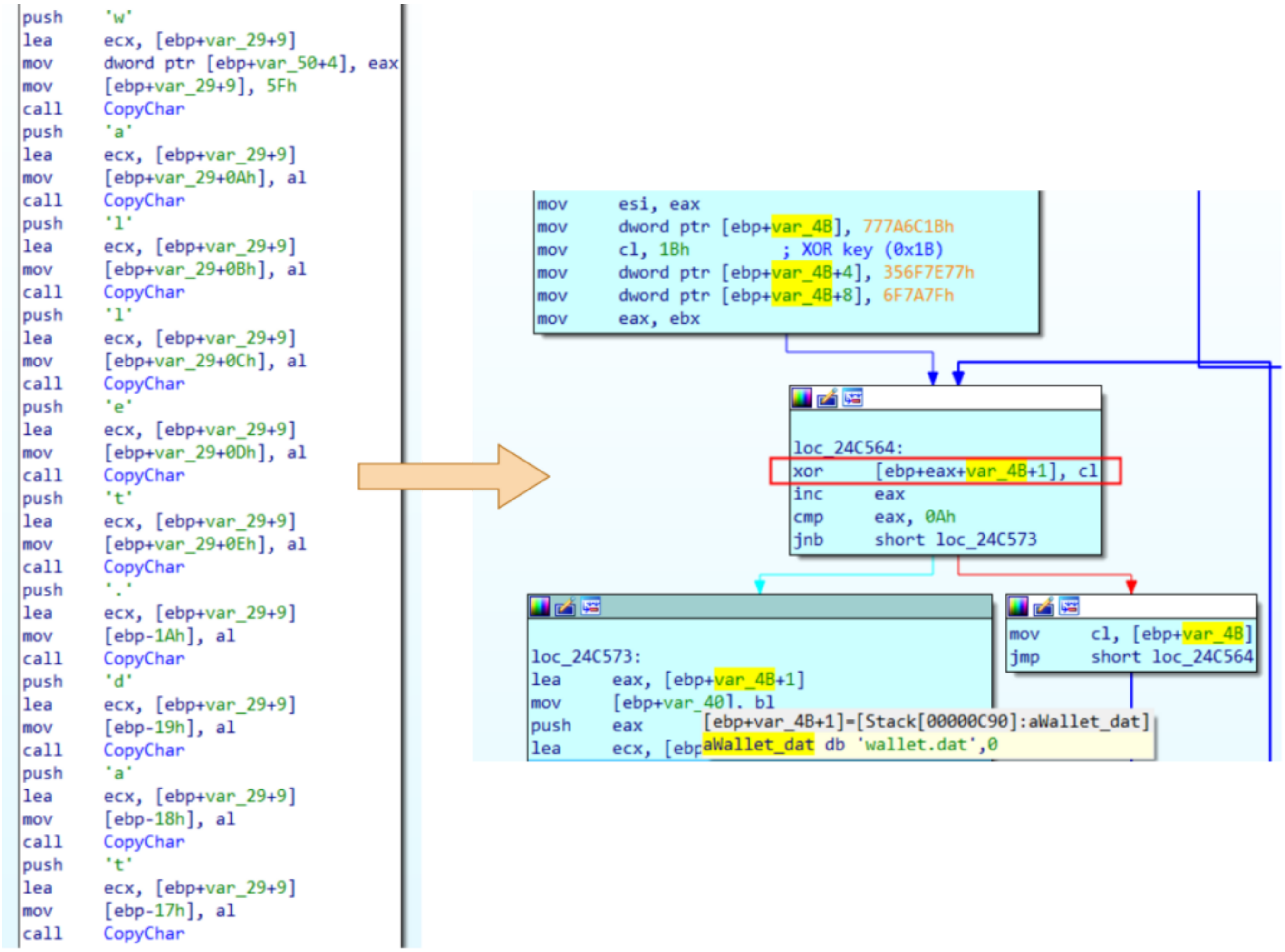

Figure 19. Stack strings are replaced by *XOR* and *SUB* encrypted strings

The combination of these obfuscation techniques leads to a lot of unnecessary loops that slow down analysis and hide functionality from static tools. As a result, the graph view of the core malware looks like this:

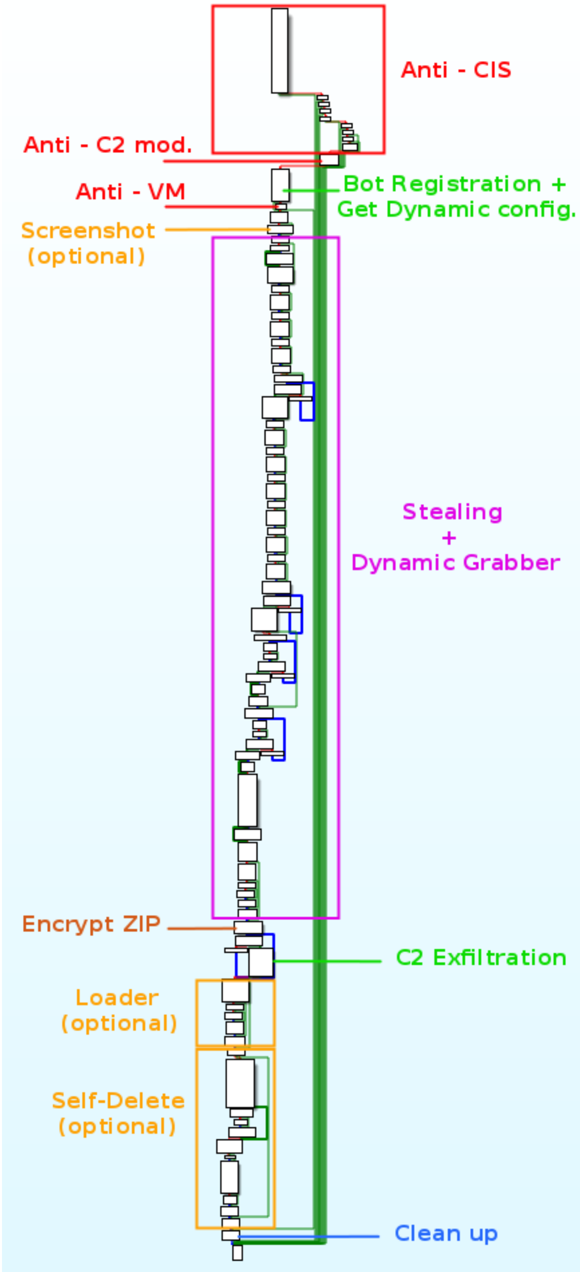

Figure 20. Taurus Stealer core functionality call graph

Resolving APIs The malware will resolve its API calls dynamically using **hashes**. It will first resolve *LoadLibraryA* and *GetProcAddress* from *kernel32.dll* to ease the resolution of further API calls. It does so by accessing the PEB of the process – more precisely to access the *DllBase* property of the **third** element from the *InLoadOrderModuleList* (which happens to be “kernel32.dll”) – and then use this address to walk through the **Export Directory** information.

```
Get_GetProcAddress_and_LoadLibraryA proc near
push    esi
mov     eax, large fs:30h ; TIB[0x30] = PEB
mov     eax, [eax+0Ch]  ; PEB[0x30] = _PEB_LDR_DATA *Ldr
mov     eax, [eax+0Ch]  ; InLoadOrderModuleList (1st entry)
mov     eax, [eax]      ; InLoadOrderModuleList (2nd entry)
mov     eax, [eax]      ; InLoadOrderModuleList (3rd entry)
mov     eax, [eax+18h]  ; DllBase kernel32.dll
mov     esi, eax
mov     edx, 0D3E65C39h ; LoadLibraryA hash
mov     ecx, esi
call    ResolveApi_
mov     edx, 0DC02BA32h ; GetProcAddress hash
mov     LoadLibraryA, eax
mov     ecx, esi
call    ResolveApi_
mov     Get_GetProcAddress, eax
pop     esi
retn
Get_GetProcAddress_and_LoadLibraryA endp
```

Figure 21. Retrieving kernel32.dll *DllBase* by accessing the 3rd entry in the *InLoadOrderModuleList* list

It will iterate *kernel32.dll AddressOfNames* structure and compute a **hash** for every exported function until the corresponding hash for "LoadLibraryA" is found. The same process is repeated for the "GetProcAddress" API call. Once both procedures are resolved, they are saved for future resolution of API calls.

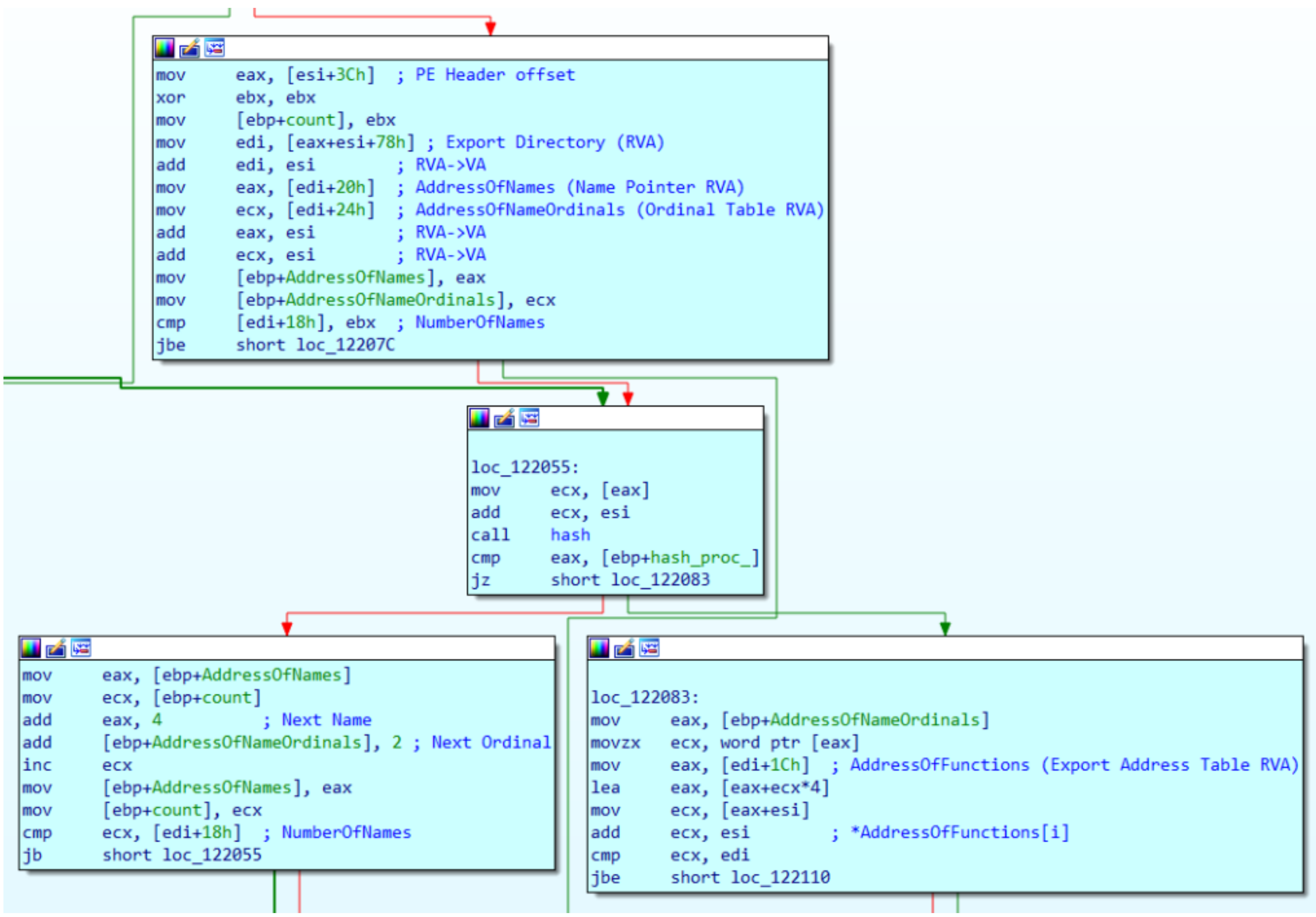

Figure 22. Taurus Stealer iterates AddressOfNames to find an API using a hashing approach

For further API resolutions, a "DLL Table String" is used to index the library needed to load an exported function and then the hash of the needed API call.

```
.rdata:002700B0 ; LPCSTR DLLs_
.rdata:002700B0 DLLs_           dd offset Kernel32_dll_0 ; DATA XREF: ResolveApi+65↑r
.rdata:002700B0                                         ; "Kernel32.dll"
.rdata:002700B4                 dd offset Shell32_dll   ; "Shell32.dll"
.rdata:002700B8                 dd offset Crypt32_dll   ; "Crypt32.dll"
.rdata:002700BC                 dd offset Wininet_dll   ; "Wininet.dll"
.rdata:002700C0                 dd offset Advapi32_dll  ; "Advapi32.dll"
.rdata:002700C4                 dd offset Gdiplus_dll   ; "gdiplus.dll"
.rdata:002700C8                 dd offset Gdi32_dll     ; "Gdi32.dll"
.rdata:002700CC                 dd offset User32_dll    ; "User32.dll"
.rdata:002700D0                 dd offset Ole32_dll     ; "Ole32.dll"
.rdata:002700D4                 dd offset Bcrypt_dll    ; "Bcrypt.dll"
.rdata:002700D8                 dd offset Urlmon_dll    ; "Urlmon.dll"
.rdata:002700DC                 dd offset Vaultcli_dll  ; "Vaultcli.dll"
.rdata:002700E0                 dd offset Netapi32_dll  ; "Netapi32.dll"
```

Figure 23. DLL Table String used in API resolutions

Resolving initial Configuration Just as with Predator The Thief, **Taurus Stealer** will load its initial configuration in a table of function pointers before the execution of the *WinMain()* function. These functions are executed in order and are responsible for loading the **C2**, **Build Id** and the **Bot Id/UUID**. C2 and Build Id are resolved using the *SUB* encryption scheme with a one-byte key. The loop uses a hard-coded length, (the size in bytes of the C2 and Build Id), which means that this has been pre-processed beforehand (probably by the builder) and that these procedures would work for only these properties.

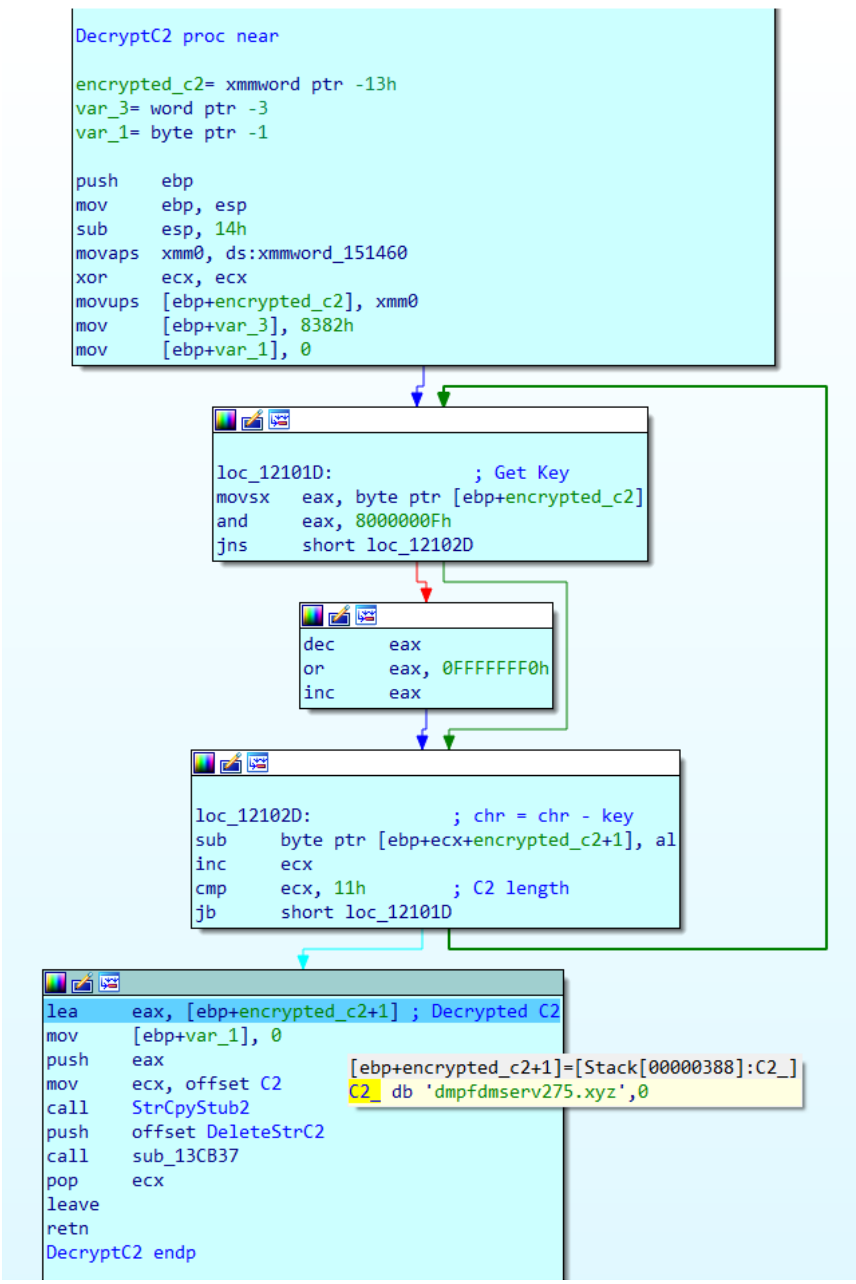

Figure 24. Taurus Stealer decrypting its Command and Control server

BOT ID / UUID Generation Taurus generates a **unique identifier** for every infected machine. Earlier versions of this malware also used this identifier as the .zip filename containing the stolen data. This behavior has been modified and now the .zip filename is randomly generated (16 random ASCII characters).

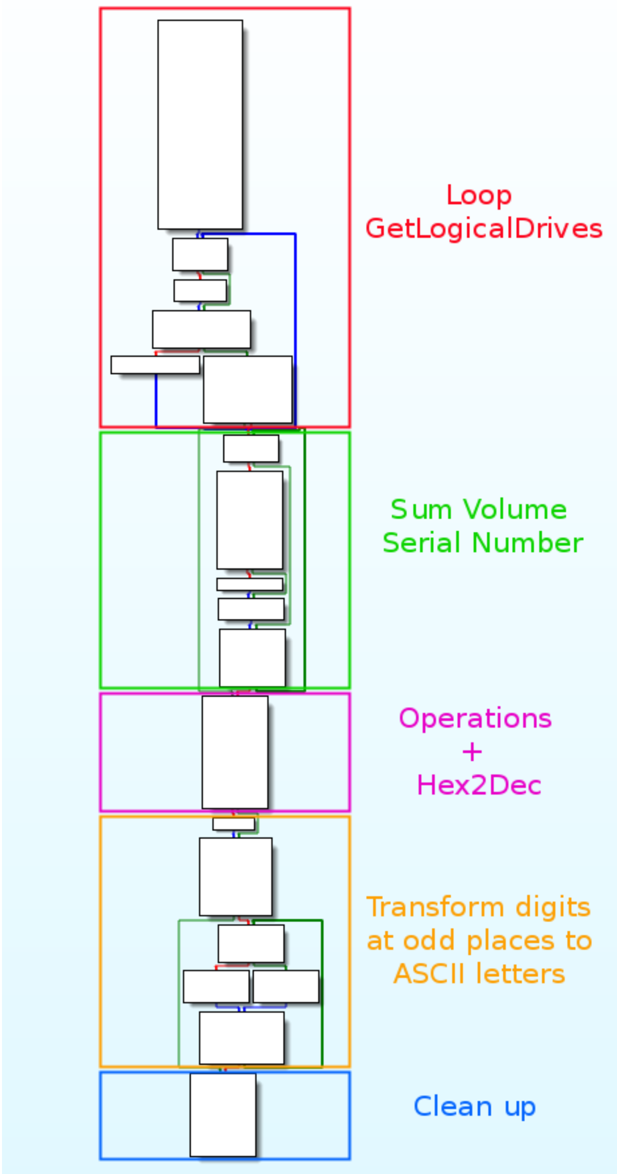

Figure 25. Call graph from the Bot Id / UUID generation routine

It starts by getting a bitmask of all the currently available disk drives using *GetLogicalDrivers* and retrieving their *VolumeSerialNumber* with GetVolumeInformationA. All these values are added into the register *ESI* (holds the sum of all *VolumeSerialNumbers* from all available Drive Letters). *ESI* is then added to itself and right-shifted 3 bytes. The result is a hexadecimal value that is converted to **decimal**. After all this process, it takes out the first two digits from the result and concatenates its full original part at the beginning. The last step consists of transforming digits in **odd** positions to ASCII letters (by adding 0x40). As an example, let's imagine an infected machine with "C:\\", "D:\\" and "Z:\\" drive letters available.

1. Call GetLogicalDrivers to get a bitmask of all the currently available disk drives.

2. Get their VolumeSerialNumber using GetVolumeInformationA:
ESI holds the sum of all VolumeSerialNumber from all available Drive Letters
GetVolumeInformationA("C:\\") -> 7CCD8A24h
GetVolumeInformationA("D:\\") -> 25EBDC39h
GetVolumeInformationA("Z:\\") -> 0FE01h
ESI = sum(0x7CCD8A24+0x25EBDC3+0x0FE01) = 0xA2BA645E

3. Once finished the sum, it will:
`mov edx, esi`
edx = (edx >> 3) + edx
Which translates to:
(0xa2ba645e >> 0x3) + 0xa2ba645e = 0xb711b0e9

4. HEX convert the result to decimal
result = hex(0xb711b0e9) = 3071389929

5. Take out the first two digits and concatenate its full original part at the beginning:
307138992971389929

6. Finally, it transforms digits in **odd** positions to ASCII letters:
s0w1s8y9r9w1s8y9r9

Anti – CIS

Taurus Stealer tries to avoid infection in countries belonging to the **Commonwealth of Independent States** (CIS) by checking the language identifier of the infected machine via GetUserDefaultLangID. Earlier Taurus Stealer versions used to have this functionality in a separate function, whereas the latest samples include this in the main procedure of the malware. It is worth mentioning that this feature is **mandatory** and will be executed at the beginning of the malware execution.

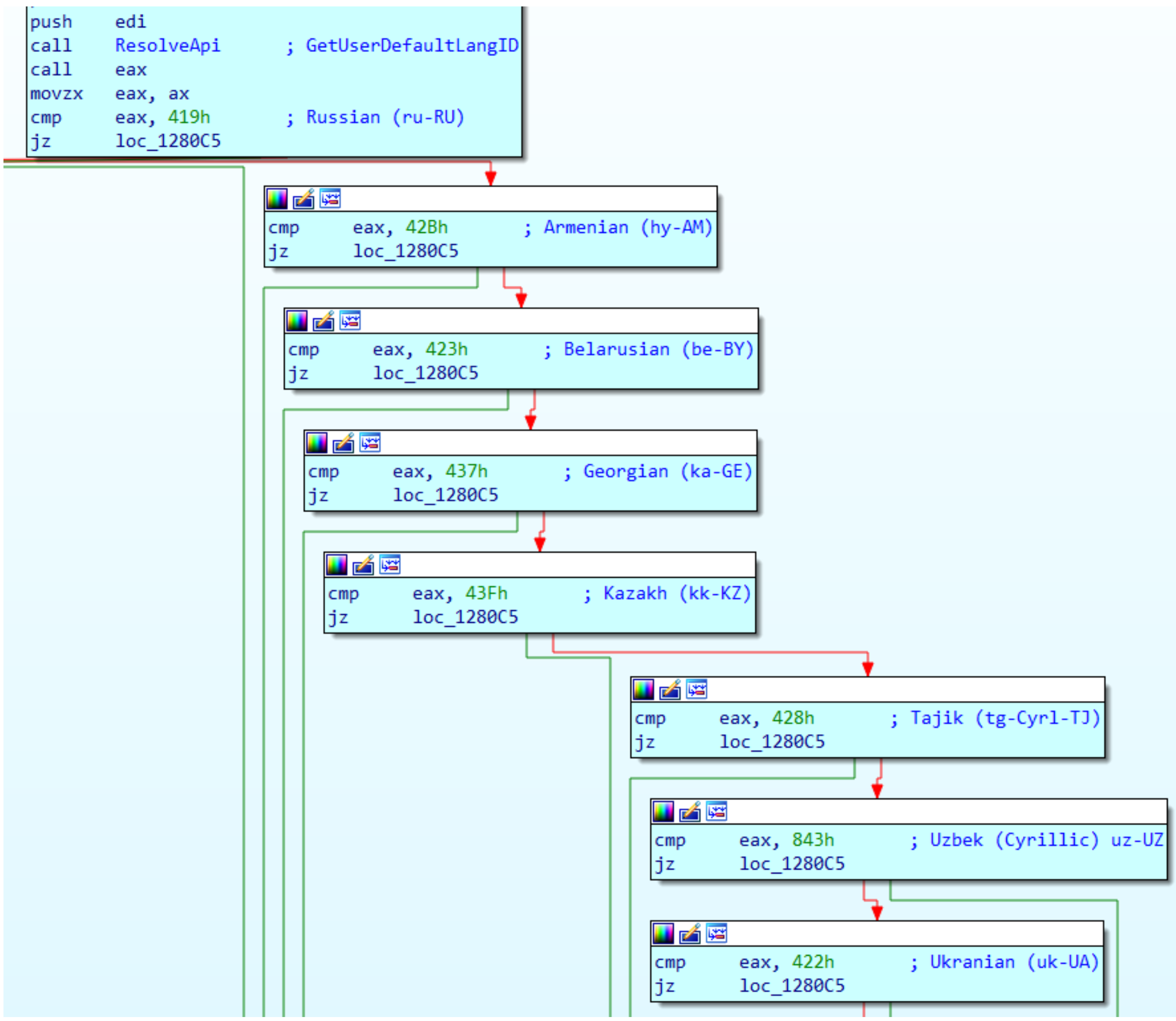

Figure 26. Taurus Stealer Anti-CIS feature

GetUserDefaultLandID returns the language identifier of the Region Format setting for the current user. If it matches one on the list, it will finish execution immediately without causing any harm.

Language Id	SubLanguage Symbol	Country
0x419	SUBLANG_RUSSIAN_RUSSIA	Russia

Language Id	SubLanguage Symbol	Country
0x42B	SUBLANG_ARMENIAN_ARMENIA	Armenia
0x423	SUBLANG_BELARUSIAN_BELARUS	Belarus
0x437	SUBLANG_GEORGIAN_GEORGIA	Georgia
0x43F	SUBLANG_KAZAK_KAZAKHSTAN	Kazakhstan
0x428	SUBLANG_TAJIK_TAJIKISTAN	Tajikistan
0x843	SUBLANG_UZBEK_CYRILLIC	Uzbekistan
0x422	SUBLANG_UKRAINIAN_UKRAINE	Ukraine

Table 2. Taurus Stealer Language Id whitelist (Anti-CIS)

Anti – C2 Mod. After the Anti-CIS feature has taken place, and **before any harmful activity occurs**, the retrieved **C2** is checked against a **hashing** function to avoid running with an invalid or modified Command and Control server. This hashing function is the same used to resolve API calls and is as follows:

```
unsigned int __fastcall hash(_BYTE *a1)
{
  unsigned int hash; // edx@1

  hash = -1;
  while ( *a1 )
    hash = dword_DFCB0[(hash ^ *a1++)] ^ (hash >> 8);
  return hash;
}
```

Figure 27. Taurus Stealer hashing function

Earlier taurus versions make use of the same hashing algorithm, except they execute **two loops** instead of one. If the **hash** of the C2 is not matching the expected one, it will avoid performing any malicious activity. This is most probably done to protect the binary from **cracked** versions and to avoid leaving traces or uncovering activity if the sample has been modified for analysis purposes.

C2 Communication

Perhaps the biggest change in this new Taurus Stealer version is how the **communications** with the **Command and Control** Server are managed. Earlier versions used two main resources to make requests:

Resource	Description
/gate/cfg/?post=1&data=<*bot_id*>	Register Bot Id and get dynamic config. Everything is sent in cleartext
/gate/log?post=2&data=<*summary_information*>	Exfiltrate data in ZIP (cleartext) *summary_information* is encrypted

Table 3. Networking resources from earlier Taurus versions

his new Taurus Stealer version uses:

Resource	Description
/cfg/	Register Bot Id and get dynamic config. BotId is sent encrypted
/dlls/	Ask for necessary .dlls (Browsers Grabbing)
/log/	Exfiltrate data in ZIP (encrypted)
/loader/complete/	ACK execution of Loader module

Table 4. Networking resources from new Taurus samples

This time no data is sent in cleartext. Taurus Stealer uses **wininet** APIs *InternetOpenA*, *InternetSetOptionA*, *InternetConnectA*, *HttpOpenRequestA*, *HttpSendRequestA*, *InternetReadFile* and *InternetCloseHandle* for its networking functionalities.

User-Agent generation

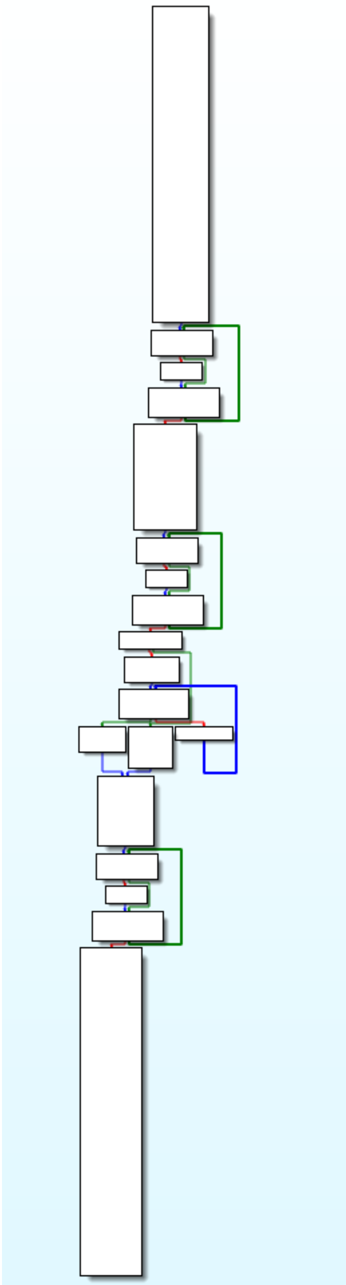
Figure 28. User-Agent generation routine call graph

The way Taurus generates the **User-Agent** that it will use for networking purposes is different from earlier versions and has introduced more steps in its creation, ending up in more **variable** results. This routine follows the next steps:

1. It will first get **OS Major Version** and **OS Minor Version** information from the PEB. In this example, we will let OS Major Version be 6 and OS Minor Version be 1.

1.1 Read TIB[0x30] -> PEB[0x0A] -> OS Major Version -> 6

1.2 Read PEB[0xA4] -> OS Minor Version -> 1

2. Call to *IsWow64Process* to know if the process is running under WOW64 (this will be needed later).

3. Decrypt string ".121 Safari/537.36"

4. Call *GetTickCount* and store result in EAX (for this example: EAX = 0x0540790F)

5. Convert HEX result to decimal result: 88111375

6. Ignore the first 4 digits of the result: 1375

7. Decrypt string " AppleWebKit / 537.36 (KHTML, like Gecko) Chrome / 83.0."

8. Check the result from the previous call to *IsWow64Process* and store it for later.

8.1 If the process is running under WOW64: Decrypt the string " WOW64)"

8.2 If the process is not running under WOW64: Load char ")" In this example we will assume the process is running under WOW64.

9. Transform from HEX to decimal OS Minor Version ("1")

10. Transform from HEX to decimal OS Major Version ("6")

11. Decrypt string "Mozilla/5.0 (Windows NT "

12. Append OS Major Version -> "Mozilla/5.0 (Windows NT 6"

13. Append '.' (hardcoded) -> "Mozilla/5.0 (Windows NT 6."

14. Append OS Minor Version -> "Mozilla/5.0 (Windows NT 6.1"

15. Append ';' (hardcoded) -> "Mozilla/5.0 (Windows NT 6.1;"

16. Append the WOW64 modifier explained before -> "Mozilla/5.0 (Windows NT 6.1; WOW64)"

17. Append string " AppleWebKit / 537.36 (KHTML, like Gecko) Chrome / 83.0." -> "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit / 537.36 (KHTML, like Gecko) Chrome / 83.0."

18. Append result of from the earlier *GetTickCount* (1375 after its processing) -> "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit / 537.36 (KHTML, like Gecko) Chrome / 83.0.1375"

19. Append the string ".121 Safari/537.36" to get the **final result**:

"Mozilla/5.0 (Windows NT **6**.**1**; **WOW64)** AppleWebKit / 537.36 (KHTML, like Gecko) Chrome / 83.0.**1375**.121 Safari/537.36"

Which would have looked like this if the process was not running under WOW64:

"Mozilla/5.0 (Windows NT **6**.**1**;**)** AppleWebKit / 537.36 (KHTML, like Gecko) Chrome / 83.0.**1375**.121 Safari/537.36"

The bold characters from the generated User-Agent are the ones that could vary depending on the OS versions, if the machine is running under WOW64 and the result of *GetTickCount* call.

How the port is set In the analyzed sample, the port for communications is set as a **hardcoded** value in a variable that is used in the code. This setting is usually hidden. Sometimes a simple "*push 80*" in the middle of the code, or a setting to a variable using "*mov [addr], 0x50*" is used. Other samples use https and set the port with a *XOR* operation like "0x3a3 ^ 0x218" which evaluates to "443", the standard **https** port. In the analyzed sample, before any communication with the **C2** is made, a hardcoded "*push 0x50 + pop EDI*" is executed to store the **port** used for communications (port 80) in *EDI*. ***EDI*** register will be used later in the code to access the communications port where necessary. The following figure shows how Taurus Stealer checks which is the port used for communications and how it sets *dwFlags* for the call to *HttpOpenRequestA* accordingly.

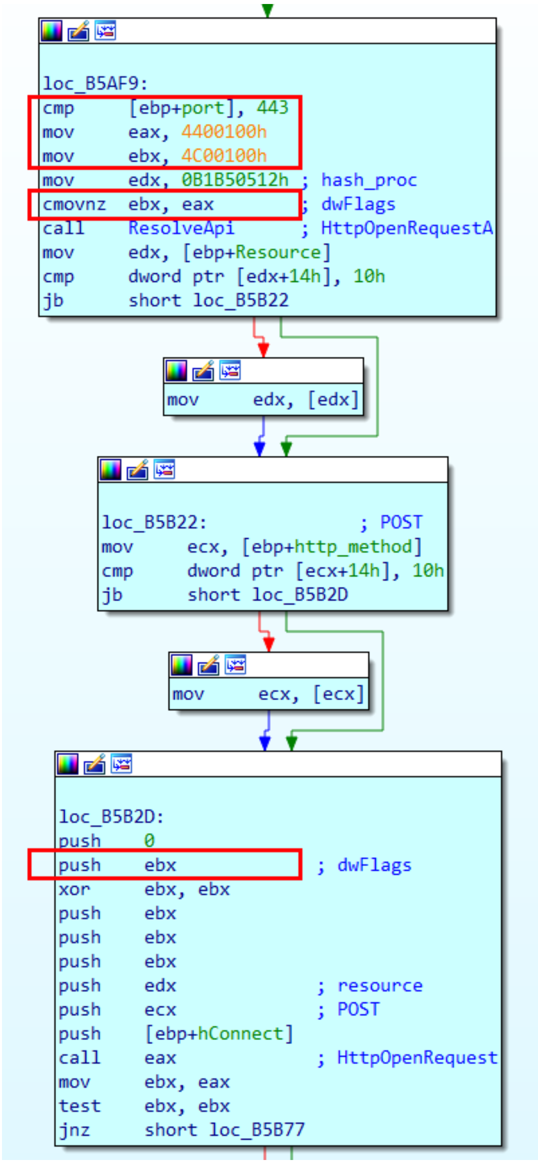

Figure 29. Taurus Stealer sets dwFlags according to the port

So, if the samples uses port 80 or any other port different from 443, the following flags will be used:

0x4400100 = INTERNET_FLAG_DONT_CACHE | INTERNET_FLAG_KEEP_CONNECTION | INTERNET_FLAG_PRAGMA_NOCACHE

If it uses port **443**, the flags will be:

0x4C00100 = NTERNET_FLAG_DONT_CACHE | INTERNET_FLAG_KEEP_CONNECTION | **INTERNET_FLAG_SECURE** | INTERNET_FLAG_PRAGMA_NOCACHE

RC4 Taurus Stealer uses RC4 stream cipher as its first layer of **encryption** for communications with the C2. The **symmetric key** used for this algorithm is **randomly** generated, which means the key will have to be stored somewhere in the body of the message being sent so that the receiver can decrypt the content. **Key Generation** The procedure we've named *getRandomString* is the routine called by Taurus Stealer to generate the **RC4 symmetric key**. It receives 2 parameters, the first is an output buffer that will receive the key and the second is the length of the key to be generated. To create the random chunk of data, it generates an array of bytes loading three XMM registers in memory and then calling **rand()** to get a random index that it will use to get a byte from this array. This process is repeated for as many bytes as specified by the second parameter. Given that all the bytes in these XMM registers are printable, this suggests that *getRandomString* produces an alphanumeric key of **n** bytes length.

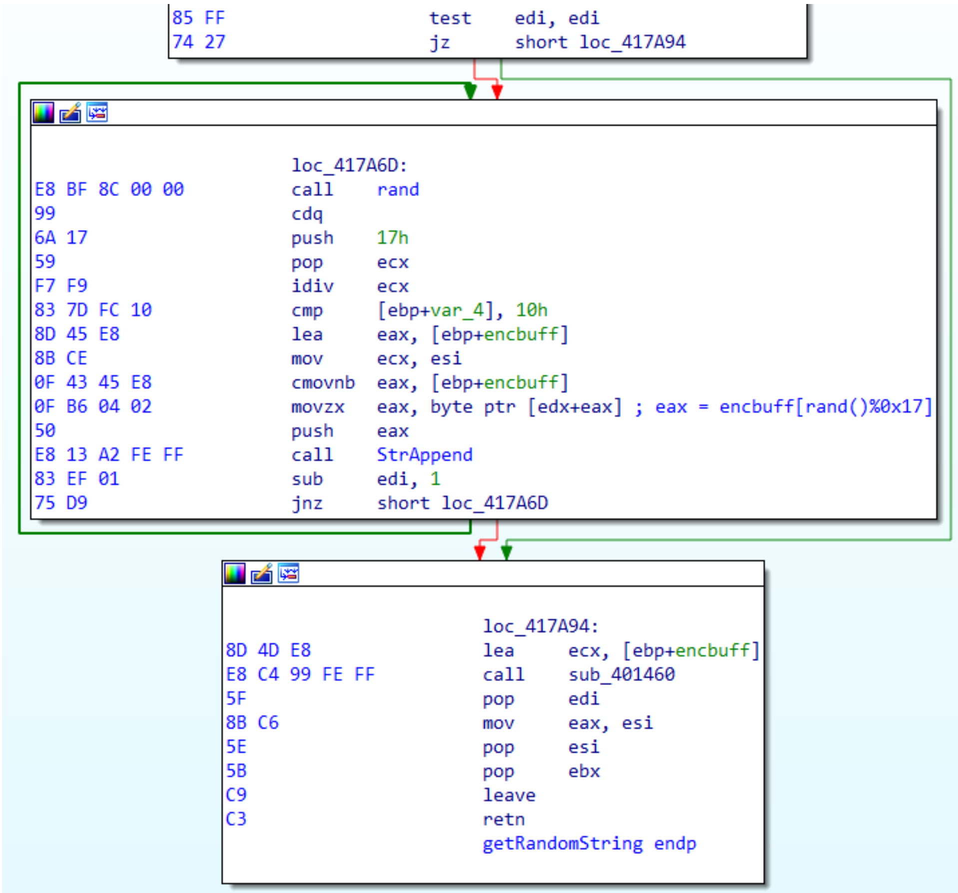

Figure 30. Taurus Stealer getRandomString routine

Given the lack of **srand**, no seed is initialized and the **rand** function will end up giving the same “random” indexes. In the analyzed sample, there is only one point in which this functionality is called with a different initial value (when creating a random directory in *%PROGRAMDATA%* to store *.dlls*, as we will see later). We’ve named this function *getRandomString2* as it has the same purpose. However, it receives an input buffer that has been processed beforehand in another function (we’ve named this function *getRandomBytes*). This input buffer is generated by initializing a big buffer and XORing it over a loop with the result of a *GetTickCount* call. This ends up giving a “random” input buffer which *getRandomString2* will use to get indexes to an encrypted string that resolves in runtime as “ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789”, and finally generate a random string for a given length. We have seen other Taurus Stealer samples moving onto this last functionality (using input buffers XORed with the result of a *GetTickCount* call to generate random chunks of data) every time randomness is needed (generation communication keys, filenames, etc.). The malware sample *d0aa932e9555a8f5d9a03a507d32ab3ef0b6873c4d9b0b34b2ac1bd68f1abc23* is an example of these Taurus Stealer variants.

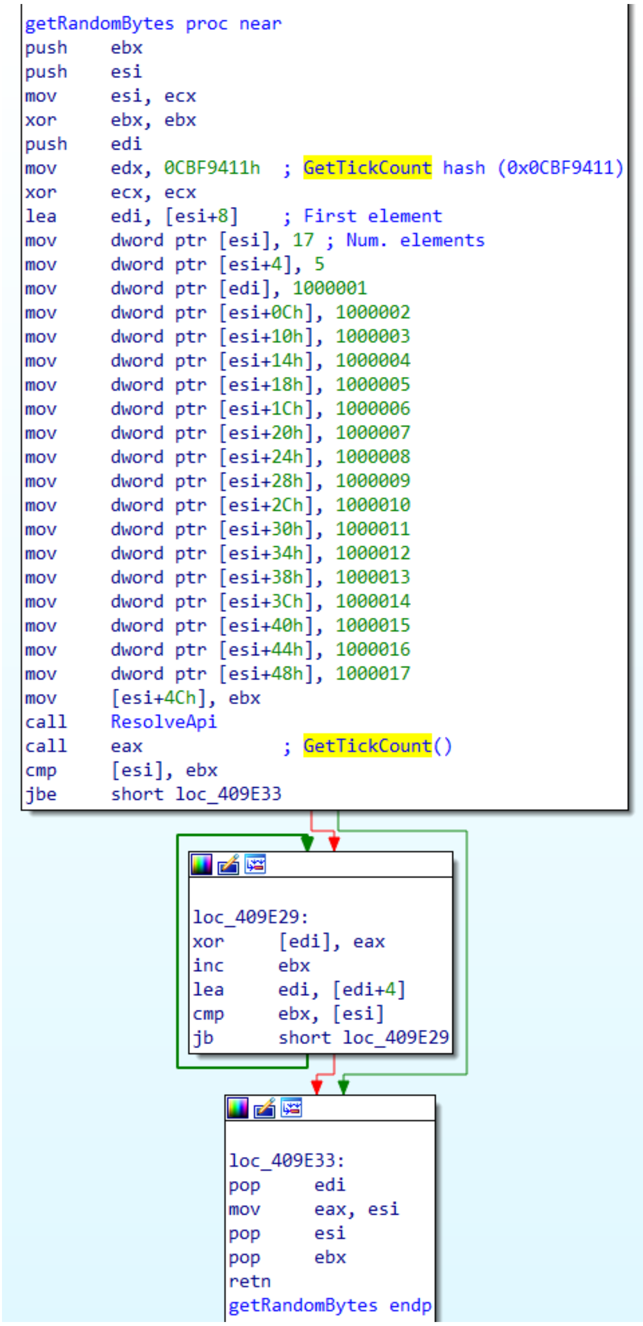

Figure 31. Taurus Stealer getRandomBytes routine

BASE64 This is the last encoding layer before C2 communications happen. It uses a classic **BASE64** to encode the message (that has been previously encrypted with RC4) and then, after encoding, the RC4 symmetric key is **appended** to the beginning of the message. The receiver will then need to get the key from the beginning of the message, BASE64 decode the rest of it and use the retrieved key to decrypt the final RC4 encrypted message. To avoid having a clear BASE64 alphabet in the code, it uses *XMM* registers to load an encrypted alphabet that is decrypted using the previously seen *SUB* encryption scheme before encoding.

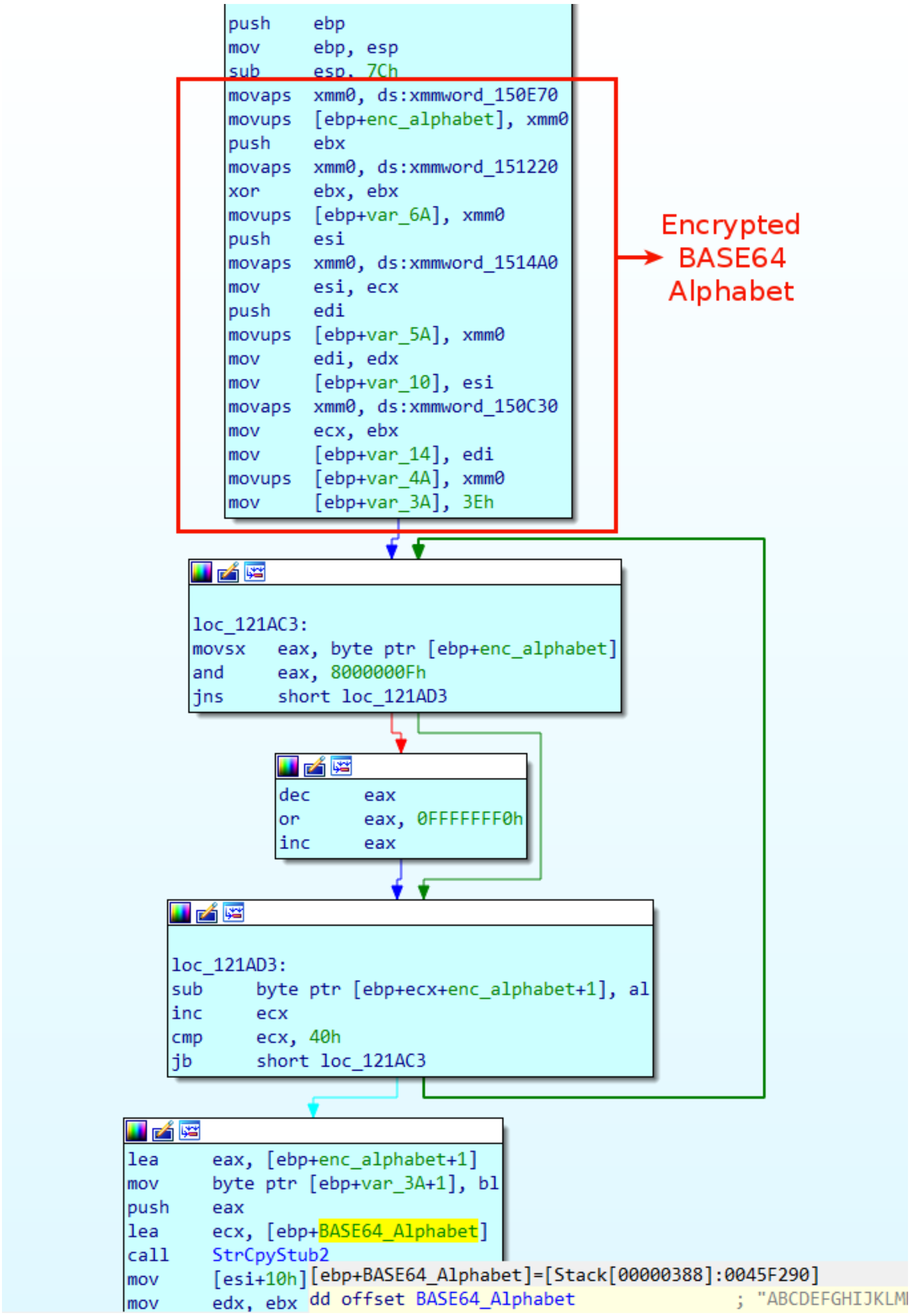

Figure 32. Taurus Stealer hiding Base64 alphabet

This is what the encryption procedure would look like:

- 1. Generate **RC4 key** using *getRandomString* with a hardcoded size of **16** bytes.
- 2. **RC4** encrypt the message using the generated 16 byte key.
- 3. **BASE64 encode** the encrypted message.
- 4. **Append** RC4 symmetric key at the beginning of the encoded message.

```
; Attributes: bp-based frame

; int __cdecl encryption(char *message)
encryption proc near

base64_message= byte ptr -30h
rc4_key= dword ptr -18h
var_8= dword ptr -8
var_4= dword ptr -4
message= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 34h
push    esi
mov     esi, ecx
lea     ecx, [ebp+rc4_key]
push    10h             ; length
pop     edx
call    getRandomString
lea     eax, [ebp+message]
push    eax             ; message
lea     eax, [ebp+rc4_key]
push    eax             ; RC4 key
call    RC4
lea     edx, [ebp+message]
lea     ecx, [ebp+base64_message]
call    base64_encode
cmp     [ebp+var_4], 10h
lea     ecx, [ebp+rc4_key]
push    [ebp+var_8]
cmovnb  ecx, [ebp+rc4_key]
push    ecx
push    ecx
mov     ecx, eax
call    StrCat
push    eax
mov     ecx, esi
call    qmemcpyStub
lea     ecx, [ebp+base64_message]
call    DeleteStr
lea     ecx, [ebp+rc4_key]
call    DeleteStr
lea     ecx, [ebp+message]
call    DeleteStr
mov     eax, esi
pop     esi
leave
retn
encryption endp
```

Figure 33. Taurus Stealer encryption routine

Bot Registration + Getting dynamic configuration Once all the initial checks have been successfully passed, it is time for Taurus to register this new Bot and retrieve the dynamic configuration. To do so, a request to the resource **/cfg/** of the C2 is made with the **encrypted Bot Id** as a message. For example, given a BotId “s0w1s8y9r9w1s8y9r9 and a key “IDaJhCHdIlfHcldJ”:

RC4(“IDaJhCHdIlfHcldJ”, “s0w1s8y9r9w1s8y9r9”) = 018784780c51c4916a4ee1c50421555e4991

It then BASE64 encodes it and appends the RC4 key at the beginning of the message:

IDaJhCHdIlfHcldJAYeEeAxRxJFqTuHFBCFVXkmR

An example of the response from the C2 could be:

xBtSRalRvNNFBNqAx0wL840EWVYxho+a6+R+rfO/Dax6jqSFhSMg+rwQrkxh4U3t6EPpqL8xAL8omji9dhO6biyzjESDBIPBfQSiM4Vs7qQMSg==

The responses go through a decryption routine that will reverse the steps described above to get the plaintext message. As you can see in the following figure, the **key length** is hardcoded in the binary and expected to be **16** bytes long.

```
; Attributes: bp-based frame

DecryptResponse proc near

encoded_message= byte ptr -30h
rc4_key= byte ptr -18h

push    ebp
mov     ebp, esp
sub     esp, 30h
lea     eax, [ebp+rc4_key]
push    esi
push    edi
push    10h             ; length
mov     esi, edx
mov     edi, ecx
push    0               ; from i = 0
push    eax             ; RC4 key
mov     ecx, esi        ; encrypted message
call    StrCpy_i_j      ; Get RC4 key (message[:16])
push    dword ptr [esi+10h] ; length
lea     eax, [ebp+encoded_message]
mov     ecx, esi        ; encrypted message
push    10h             ; from i = 16
push    eax             ; encoded_message
call    StrCpy_i_j      ; Get encoded message (message[16:])
push    eax
mov     ecx, esi
call    StrCpy
lea     ecx, [ebp+encoded_message]
call    DeleteStr
mov     edx, esi
mov     ecx, edi
call    base64_decode
push    edi
lea     eax, [ebp+rc4_key]
push    eax
call    RC4
lea     ecx, [ebp+rc4_key]
call    DeleteStr
mov     eax, edi
pop     edi
pop     esi
leave
retn
DecryptResponse endp
```

Figure 34. Taurus Stealer decrypting C2 responses

To decrypt it, we do as follow: 1. Get **RC4 key** (first 16 bytes of the message) `xBtSRalRvNNFBNqA` 2. BASE64 **decode** the rest of the message (after the RC4 key)

`c41b5245a951bcd34504da80c74c0bf38d04595631868f9aebe47eadf3bf0dac7a8ea485852320fabc10ae4c61e14dede843e9a8bf3100bf289a38bd7613ba6e2cb38c44830483c17d04a233856ceea40c4a`

3. **Decrypt** the message using RC4 key (get dynamic config.) `[1;1;1;1;1;0;1;1;1;1;1;1;1;1;1;1;1;5000;0;0]#[]#[156.146.57.112;US]#[]` We can easily see that consecutive configurations are separated by the character ";", while the character '**#**' is used to separate different configurations. We can summarize them like this: `[STEALER_CONFIG]#[GRABBER_CONFIG]#[NETWORK_CONFIG]#[LOADER_CONFIG]` In case the C2 is down and no dynamic configuration is available, it will use a hardcoded configuration stored in the binary which would enable all stealers, Anti-VM, and Self-Delete features. (Dynamic Grabber and Loader modules are not enabled by default in the analyzed sample).

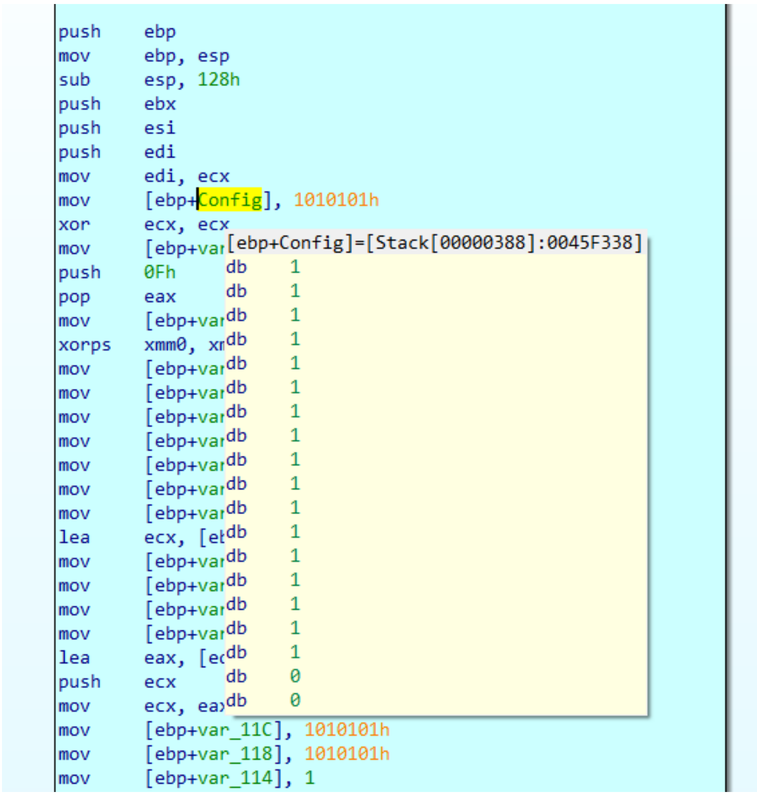

Figure 35. Taurus uses a static hardcoded configuration If C2 is not available

Anti – VM (optional) This functionality is **optional** and depends on the retrieved configuration. If the malware detects that it is running in a Virtualized environment, it will abort execution before causing any damage. It makes use of **old** and common x86 **Anti-VM instructions** (like the **RedPill** technique) to detect the Virtualized environment in this order:

- SIDT
- SGDT
- STR
- CPUID
- SMSW

```
__int8 __cdecl anti_vm()
{
  __int8 result; // al@8
  __int128 v2; // [sp+8h] [bp-20h]@5
  char v3[6]; // [sp+18h] [bp-10h]@1
  int v4; // [sp+20h] [bp-8h]@6
  int v5; // [sp+24h] [bp-4h]@3

  __sidt(v3);
  if ( (*(_DWORD *)&v3[2] & 0xFF000000) == -16777216 )
    goto LABEL_12;
  __sgdt(v3);
  if ( (*(_DWORD *)&v3[2] & 0xFF000000) == -16777216 )
    goto LABEL_12;
  v5 = 0;
  __asm { str     word ptr [ebp+var_4] }
  if ( !(_BYTE)v5 && BYTE1(v5) == 64 )
    goto LABEL_12;
  v2 = xmmword_4307E0;
  cpuid_(&v2);
  if ( (SDWORD2(v2) >> 31) & 1 )
    goto LABEL_12;
  __asm { smsw    eax }
  v4 = _EAX;
  if ( (_EAX & 0xFF000000) != -872415232 )
    goto LABEL_13;
  if ( (_EAX & 0xFF0000) == 13369344 )
LABEL_12:
    result = 1;
  else
LABEL_13:
    result = 0;
  return result;
}
```

Figure 36. Taurus Stealer Anti-VM routine

Stealer / Grabber

We can distinguish **5** main **grabbing methods** used in the malware. All paths and strings required, as usual with Taurus Stealer, are created at runtime and come encrypted in the methods described before. **Grabber 1** This is one of the most used grabbing methods, along with the malware execution (if it is not used as a call to the grabbing routine it is implemented inside another function in the same way), and consists of traversing files (it **ignores** directories) by using *kernel32.dll FindFirstFileA*, *FindNextFileA* and *FindClose* API calls. This grabbing method **does not use recursion**. The grabber expects to receive a directory as a parameter for those calls (it can contain wildcards) to start the search with. Every found file is grabbed and added to a ZIP file in **memory** for future exfiltration. An example of its use can be seen in the Wallets Stealing functionality, when searching, for instance, for *Electrum* wallets: **Grabber 2** This grabber is used in the Outlook Stealing functionality and uses *advapi32.dll RegOpenKeyA*, *RegEnumKeyA*, *RegQueryValueExA* and *RegCloseKey* API calls to access the and steal from **Windows Registry**. It uses a **recursive** approach and will start traversing the Windows Registry searching for a specific key from a given starting point until *RegEnumKeyA* has no more keys to enumerate. For instance, in the Outlook Stealing functionality this grabber is used with the starting Registry key "*HKCU\software\microsoft\office*" searching for the key "*9375CFF0413111d3B88A00104B2A667*". **Grabber 3** This grabber is used to steal **browsers** data and uses the same API calls as **Grabber 1** for traversing files. However, it loops through **all files and directories** from *%USERS%* directory and favors **recursion**. Files found are processed and added to the ZIP file in memory. One curious detail is that if a "wallet.dat" is found during the parsing of files, it will only be dumped if the current **depth** of the recursion is **less or equal to** 5. This is probably done in an attempt to avoid dumping invalid wallets. We can summarize the files Taurus Stealer is interested in the following table:

Grabbed File	Affected Software
History	Browsers
formhistory.sqlite	Mozilla Firefox & Others
cookies.sqlite	Mozilla Firefox & Others
wallet.dat	Bitcoin
logins.json	Chrome
signongs.sqlite	Mozilla Firefox & Others
places.sqlite	Mozilla Firefox & Others
Login Data	Chrome / Chromium based

Grabbed File	Affected Software
Cookies	Chrome / Chromium based
Web Data	Browser

Table 5. Taurus Stealer list of files for Browser Stealing functionalities

Grabber 4

This grabber steals information from the **Windows Vault**, which is the default storage vault for the credential manager information. This is done through the use of **Vaultcli.dll**, which encapsulates the necessary functions to access the Vault. Internet Explorer data, since it's version 10, is stored in the Vault. The malware loops through its items using:

- VaultEnumerateVaults
- VaultOpenVault
- VaultEnumerateItems
- VaultGetItem
- VaultFree

Grabber 5 This last grabber is the customized grabber module (**dynamic grabber**). This module is responsible for grabbing files configured by the **threat actor** operating the botnet. When Taurus makes its first request to the C&C, it retrieves the malware configuration, which can include a customized grabbing configuration to search and steal files. This functionality is not enabled in the default static configuration from the analyzed sample (the configuration used when the C2 is not available). As in earlier grabbing methods, this is done via file traversing using *kernel32.dll FindFirstFileA*, *FindNextFileA* and *FindClose* API calls. The threat actor may set **recursive** searches (optional) and **multiple wildcards** for the search.

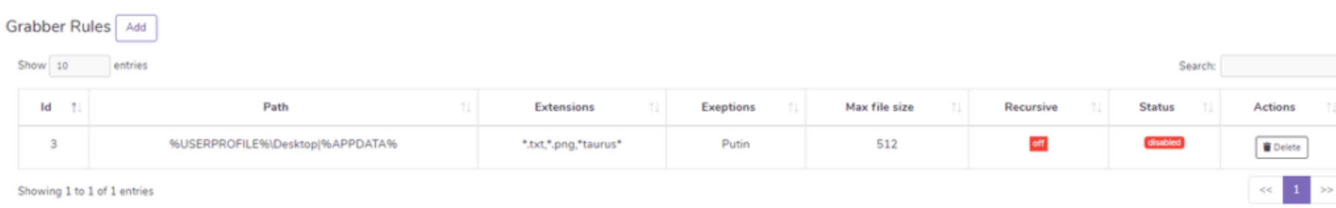

Figure 37. Threat Actor can add customized grabber rules for the dynamic grabber

Targeted Software This is the software the **analyzed sample** is targeting. It has functionalities to steal from: Wallets:

- Electrum
- MultiBit
- Armory
- Ethereum
- Bytecoin
- Jaxx
- Atomic
- Exodus
- Dahscore
- Bitcoin
- Wasabi
- Daedalus
- Monero

Games:

Steam

Communications:

- Telegram
- Discord
- Jabber

Mail:

- FoxMail
- Outlook

FTP:

- FileZilla
- WinSCP

2FA Software:

Authy

VPN:

NordVPN

Browsers:

- Mozilla Firefox (also Gecko browsers)
- Chrome (also Chromium browsers)
- Internet Explorer
- Edge
- Browsers using the same files the grabber targets.

However, it has been seen in other samples and their advertisements that Taurus Stealer also supports other software not included in the list like **BattleNet**, **Skype** and **WinFTP**. As mentioned earlier, they also have an open communication channel with their customers, who can suggest new software to add support to. **Stealer Dependencies** Although the posts that sell the malware in underground forums claim that Taurus Stealer does not have any dependencies, when stealing browser information (by looping through files recursively using the "**Grabber 3**" method described before), if it finds "logins.json" or "signons.sqlite" it will then **ask** for needed ***.dlls*** to its C2. It first creates a directory in *%PROGRAMDATA%\<bot id>*, where it is going to dump the downloaded *.dlls*. It will check if "*%PROGRAMDATA%\<bot id>\nss3.dll*" exists and will ask for its C2 (doing a request to **/dlls/** resource) if not. The *.dlls* will be finally dumped in the following order:

- 1. freebl3.dll
- 2. mozglue.dll
- 3. msvcp140.dll
- 4. nss3.dll
- 5. softokn3.dll
- 6. vcruntime140.dll

If we find the C2 down (when analyzing the sample, for example), we will not be able to download the required files. However, the malware will still try, no matter what, to load those libraries after the request to **/dlls/** has been made (starting by loading "nss3.dll"), which would lead to a crash. The malware would stop working from this point. In contrast, if the C2 is alive, the .dlls will be downloaded and **written to disk** in the order mentioned before. The following figure shows the call graph from the routine responsible for requesting and dumping the required libraries to disk.

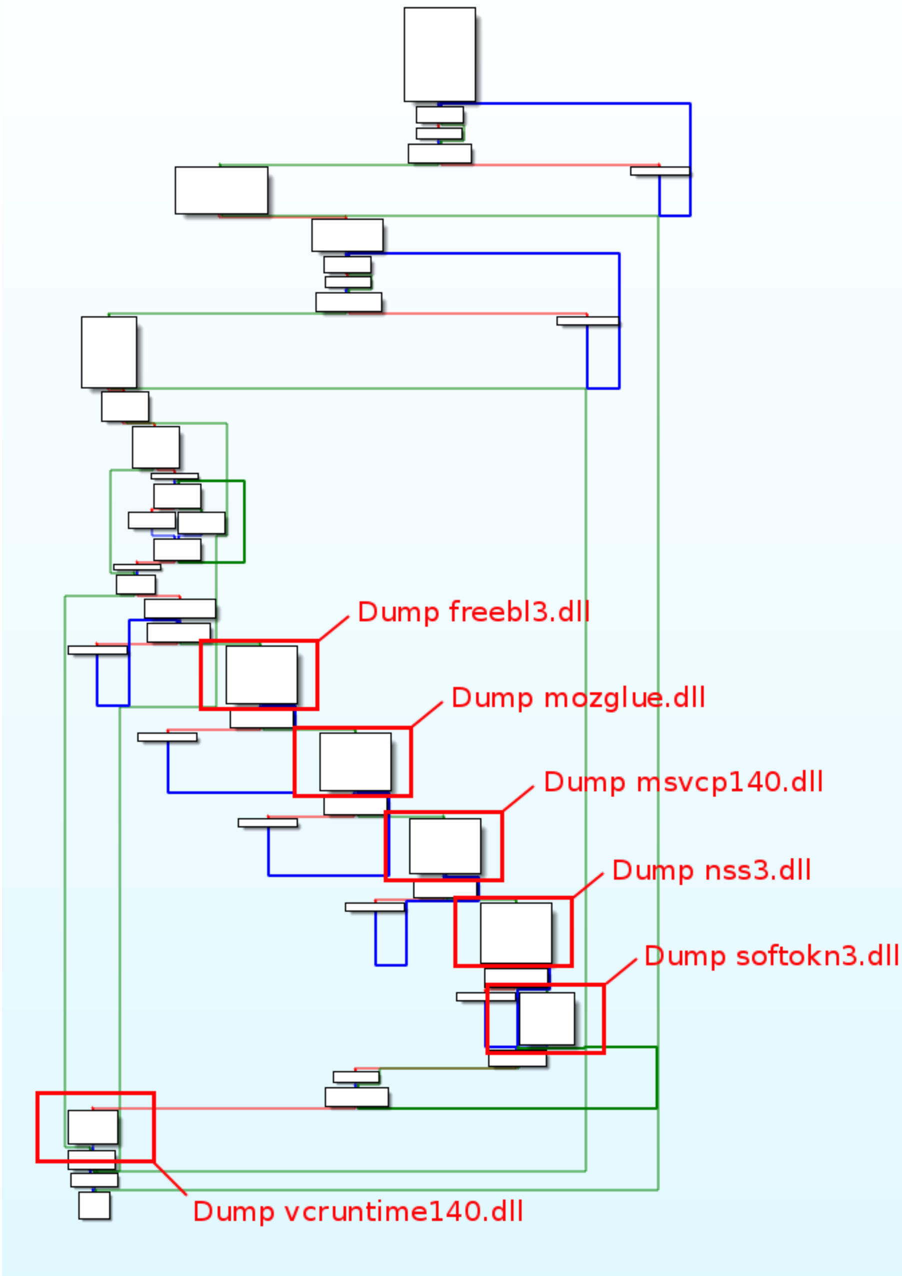

Figure 38. Taurus Stealer dumping retrieved *.dlls* from its Command and Control Server to disk

Information Gathering After the Browser stealing process is finished, Taurus proceeds to gather information from the infected machine along with the Taurus Banner and adds this data to the ZIP file in memory with the filename “Information.txt”. All this functionality is done through a series of unnecessary steps caused by all the obfuscation techniques to hide strings, which leads to a horrible function call graph:

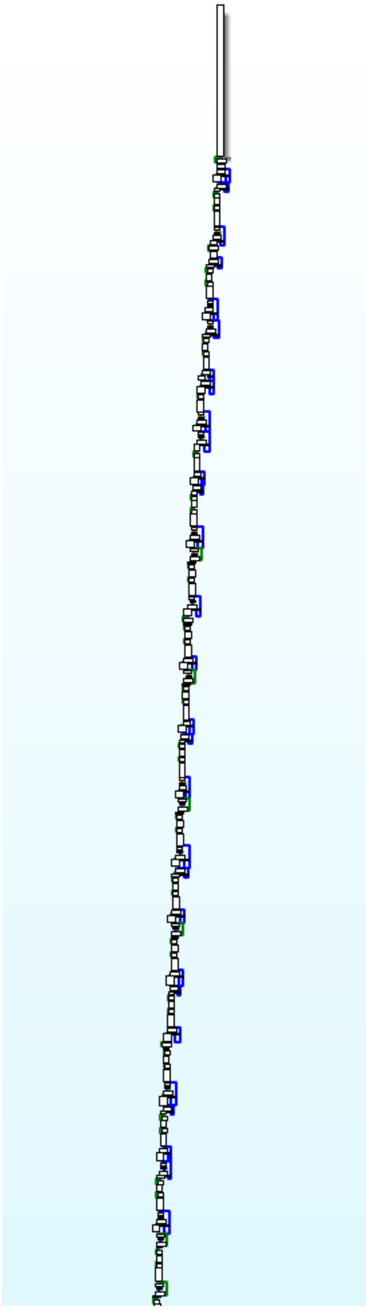

Figure 39. Taurus Stealer main Information Gathering routine call graph

It gets information and concatenates it sequentially in memory until we get the final result:

‘ ________ _______ ___ ___ ‘

‘ ‘

‘ /_ ___/_____ _____ ____________ __________ / _____// /_____ ______ _/ /___ ‘

‘ _______’

‘ / / / ___ `/ / / / _____/ / / / _____/ \___ \/ ___/ _ \/ ___ `/ / _ \’

‘/ _____/’

‘ / / / /_/ / /_/ / / / / /_/ (___) _____/ / /_/ ___/ /_/ / / ___/’

‘ / ‘

‘/_/ ___,_/___,_/_/ ___,_/______/ /______/___/_____/___,_/_/_____/_’

‘/

‘____’

‘____________’

‘|Buy at Telegram: t.me/taurus_seller |Buy at Jabber: taurus_selle’

‘r@exploit.im|’

‘““““““““““““““““““““““““““““““““““““`’

‘““““““`’

‘UID: s0w1s8y9r9w1s8y9r9’

‘Prefix: MyAwesomePrefix’

‘Date: 15.4.2021 14:57’

‘IP: ‘

‘Country: ‘

‘OS: Windows 6.1 7601 x64’

‘Logical drives: C: D: Z: ‘

‘Current username: User’

‘Computername: USER-PC’

‘Domain: WORKGROUP’

‘Computer users: All Users, Default, Default User, Public, User, ‘

‘Keyboard: Spanish (Spain, International Sort)/English (United States)’

‘Active Window: IDA – C:\Users\User\Desktop\TAURUS_v2.idb (TAURUS_’

‘v2.exe)’

‘CPU name: Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz’

‘Number of CPU kernels: 2’

‘GPU name: VirtualBox Graphics Adapter’

‘RAM: 3 GB’

‘Screen resolution: 1918×1017’

‘Working path: C:\Users\User\Desktop\TAURUS_v2.exe’,0

One curious difference from earlier Taurus Stealer versions is that the **Active Window** from the infected machine is now also included in the information gathering process.

Enumerate Installed Software As part of the information gathering process, it will try to get a list of the installed software from the infected machine by looping in the registry from “*HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall*” and retrieving *DisplayName* and *DisplayVersion* with *RegQueryValueExA* until *RegEnumKeyA* does not find more keys. If software in the registry list has the key “DisplayName”, it gets added to the list of installed software. Then, if it also has “Display Version” key, the value is appended to the name. In case this last key is not available, “[Unknown]” is appended instead. Following the pattern: “**DisplayName**\t**DisplayVersion**” As an example:

“Cheat Engine 6.5.1\t[Unknown]” “Google Chrome\t[89.0.4389.90]” (…)

The list of software is included in the ZIP file in memory with the filename “Installed Software.txt”

C2 Exfiltration

During the stealing process, the data that is grabbed from the infected machine is saved in a ZIP file in memory. As we have just seen, information gathering files are also included in this fileless ZIP. When all this data is ready, Taurus Stealer will proceed to:

- 1. Generate a Bot Id results summary message.
- 2. Encrypt the ZIP file before exfiltration.
- 3. Exfiltrate the ZIP file to Command and Control server.
- 4. Delete traces from networking activity

Generate Bot Id results summary The results summary message is created in 2 stages. The first stage loads generic information from the infected machine (Bot Id, Build Id, Windows version and architecture, current user, etc.) and a summary count of the number of passwords, cookies, etc. stolen. As an example:

s0w1s8y9r9w1s8y9r9|MyAwesomePrefix|Windows 6.1 7601 x64|USER-PC|WORKGROUP|||0|576|0|7|empty

Finally, it concatenates a string that represents a mask stating which Software has been available to steal information from (e.g. Telegram, Discord, FileZilla, WinSCP. etc.).

s0w1s8y9r9w1s8y9r9|MyAwesomePrefix|Windows 6.1 7601 x64|USER-PC|WORKGROUP|||0|576|0|7|empty|1|1|0|0|0|1|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0

This summary information is then added in the memory ZIP file with the filename "LogInfo.txt". This behavior is different from earlier Taurus Stealer versions, where the information was sent as part of the URL (when doing exfiltration POST request to the resource /gate/log/) in the parameter "data". Although this summary information was encrypted, the exfiltrated ZIP file was sent in cleartext. **Encrypt ZIP before exfiltration** Taurus Stealer will then encrypt the ZIP file in memory using the techniques described before: using the **RC4** stream cipher with a randomly generated key and encoding the result in **BASE64**. Because the RC4 key is needed to decrypt the message, the key is included at the beginning of the encoded message. In the analyzed sample, as we saw before, the key length is **hardcoded** and is **16 bytes**. As an example, this could be an encrypted message being sent in a POST request to the /log/ resource of a Taurus Stealer C2, where the RC4 key is included at the beginning of the message (first 16 characters).

"jaCghbIiGeGEADIjMayhQpGzXwORMFuHXzsUCiVH12jIA" (…)

Exfiltrate ZIP file to Command and Control server As in the earlier versions, it uses a **try-retry** logic where it will try to exfiltrate up to 10 times (in case the network is failing, C2 is down, etc.). It does so by opening a handle using *HttpOpenRequestA* for the "/log/" resource and using this handle in a call to *HttpSendRequestA***,** where exfiltration is done (the data to be exfiltrated is in the **post_data** argument). The following figure shows this try-retry logic in a loop that executes *HttpSendRequestA*.

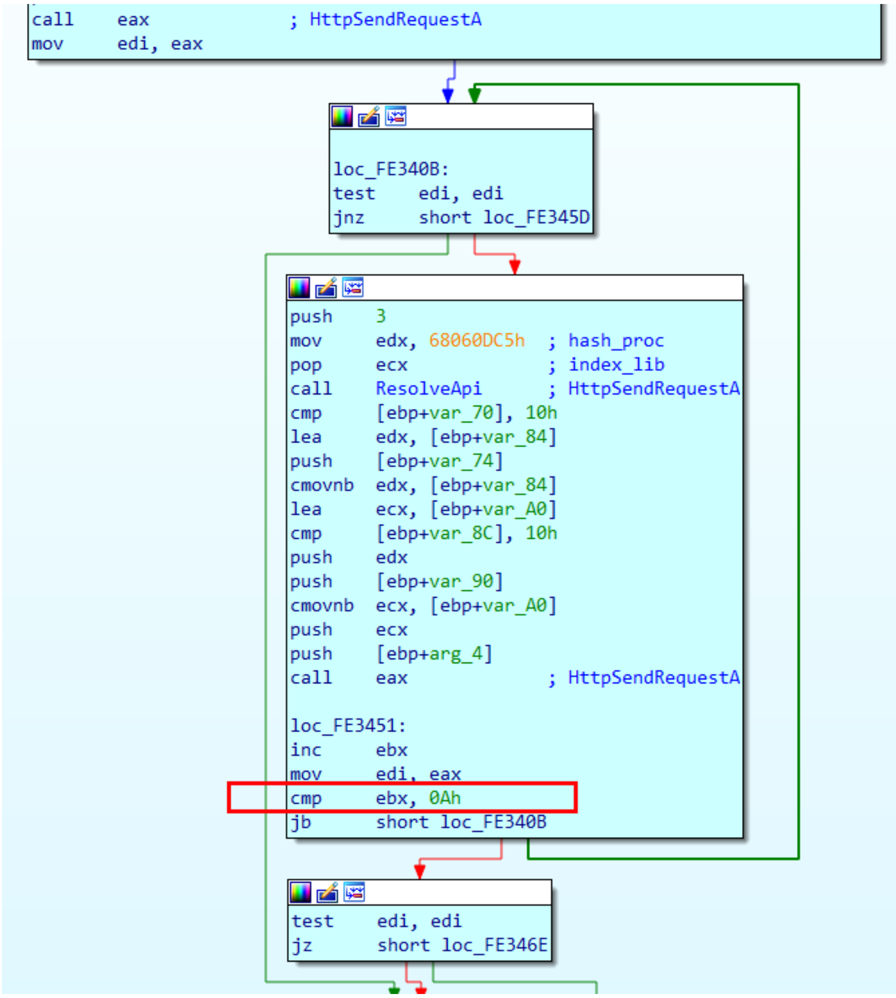

Figure 40. Taurus Stealer will try to exfiltrate up to 10 times

The encrypted ZIP file is sent with *Content-Type: application/octet-stream*. The filename is a randomly generated string of 16 bytes. However, earlier Taurus Stealer versions used the Bot Id as the .zip filename. **Delete traces from networking activity** After exfiltration, it uses *DeleteUrlCacheEntry* with the C2 as a parameter for the API call, which deletes the cache entry for a given URL. This is the last step of the exfiltration process and is done to avoid leaving traces from the networking activity in the infected machine.

Loader (optional)

Upon exfiltration, the **Loader** module is executed. This module is **optional** and gets its configuration from the first C2 request. If the module is enabled, it will load an URL from the Loader configuration and execute *URLOpenBlockingStream* to download a file. This file will then be dumped in *%TEMP%* folder using a random filename of **8** characters. Once the file has been successfully dumped in the infected machine it will execute it using *ShellExecuteA* with the option *nShowCmd* as "SW_HIDE", which hides the window and activates another one. If **persistence** is set in the **Loader configuration**, it will also **schedule** a task in the infected machine to run the downloaded file every minute using:

C:\windows\system32\cmd.exe /c schtasks /create /F /sc minute /mo 1 /tn "\WindowsAppPool\AppP ool" /tr "C:\Users\User\AppData\Local\Temp\FfjDEIdA.exe"

The next figure shows the Schedule Task Manager from an infected machine where the task has been scheduled to run every minute indefinitely.

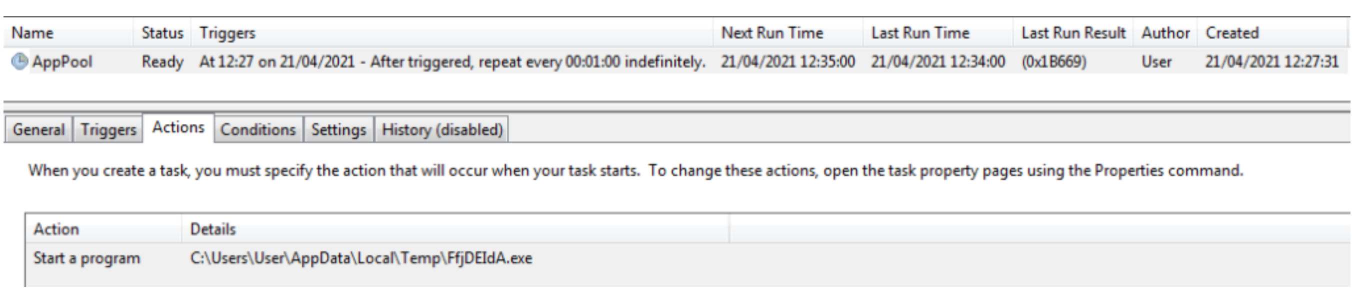

Figure 41. Loader persistence is carried out by creating a scheduled task to run every minute indefinitely

Once the file is executed, a new POST request is made to the C2 to the resource **/loader/complete/**. The following figure summarizes the main responsibilities of the **Loader** routine.

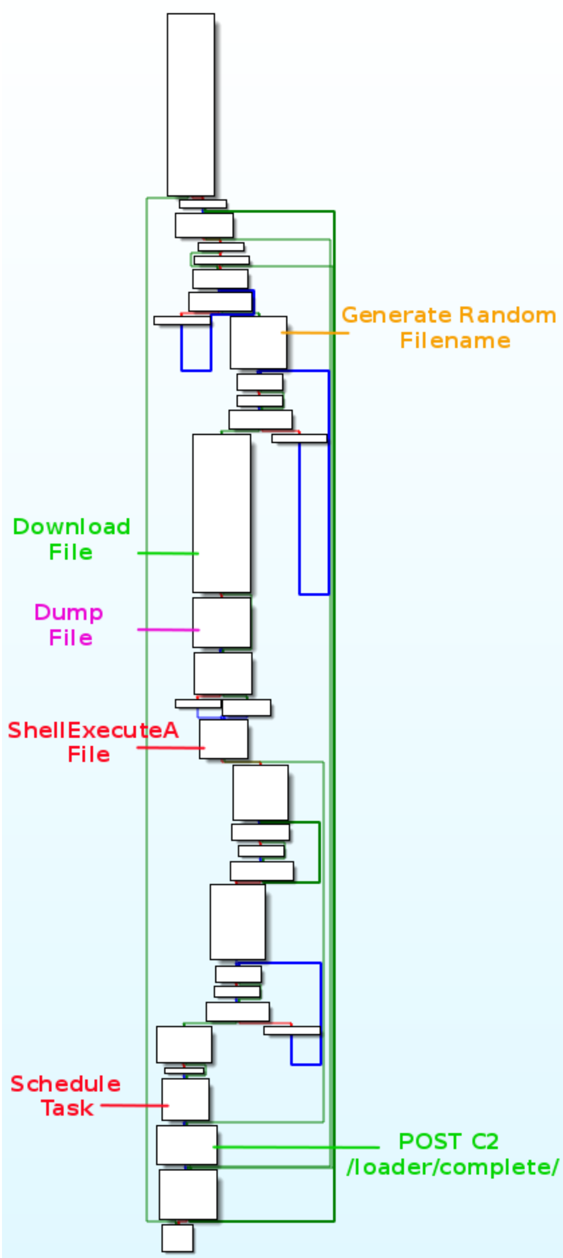

Figure 42. Taurus Stealer Loader routine call graph

Self-Delete (optional)

This functionality is the last one being executed in the malware and is also optional, although it is enabled by default if no response from the C2 was received in the first request. It will use *CreateProcessA* to execute **cmd.exe** with the following arguments:

cmd.exe /c timeout /t 3 & del /f /q <malware_filepath>

Malware_filepath is the actual path of the binary being executed (itself). A small **timeout** is set to give time to the malware to finish its final tasks. After the creation of this process, only a clean-up routine is executed to delete strings from memory before finishing execution.

YARA rule

This memory Yara rule detects both old and new Taurus Stealer versions. It targets some unique functionalities from this malware family:

- **Hex2Dec**: Routine used to convert from a Hexadecimal value to a Decimal value.
- **Bot Id/UUID** generation routine.

- **getRandomString**: Routine used to generate a random string using rand() over a static input buffer
- **getRandomString2**: Routine used to generate a random string using rand() over an input buffer previously "randomized" with GetTickCount
- **getRandomBytes**: Routine to generate "random" input buffers for *getRandomString2*
- **Hashing** algorithm used to resolve APIs and Anti – C2 mod. feature.

```
rule taurus_stealer_memory {
meta:
description = "Detects Taurus Stealer"
author = "Blueliv"
date = "27/04/2021"
strings:
/* Hex2Dec */
$op00 = { 33 D2 4E 6A 0A 59 F7 F1 80 C2 30 88 16 85 C0 75 EF 51 8D 45 FD 8B CF 50 56 E8 ?? ?? ?? ?? 8B C7 5F 5E C9 C3 }
/* Bot Id/UUID Generation */
$op01 = { 8D ?? ?? ?? ?? 8D [2-3] 7? ?? [4-5] 0F [3-4] 8A 04 ?? 04 40 EB }
/* getRandomString */
$op02 = { E8 ?? ?? ?? ?? 99 6A 17 59 F7 F9 (83 ?? ?? ?? 8D ?? ?? | 8D ?? ?? 83 ?? ?? ??) [0-3] 0F 43 ?? ?? }
/* getRandomString2 */
$op03 = { 33 D2 F7 36 8B 74 8E 08 8B 4D FC 6A 3F 03 74 91 08 33 D2 8B 41 4C F7 31 }
/* getRandomBytes */
$op04 = { C7 46 ?? ?? 42 0F 00 C7 46 ?? ?? 42 0F 00 C7 46 ?? ?? 42 0F 00 89 ?? ?? E8 ?? ?? ?? ?? FF D0 39 1E 76 0A 31 07 43 8D 7F 04
3B 1E 72 F6 }
/* Hashing algorithm */
$op05 = { 0F BE [1-2] 33 C2 (C1 EA 08 0F B6 C0 | 0F B6 C0 C1 EA 08) 33 14 85 ?? ?? ?? ?? 4? }
condition:
4 of them
}
```

MITRE ATT&CK

Tactic	Technique ID	Technique
Execution	T1059	Command and Scripting Interpreter
Execution / Persistence	T1053	Scheduled Task/Job
Defense Evasion	T1140	Deobfuscate/Decode Files or Information
Defense Evasion	T1070	Indicator Removal on Host
Defense Evasion	T1027	Obfuscated Files or Information
Defense Evasion / Discovery	T1497	Virtualization/Sandbox Evasion
Credential Access	T1539	Steal Web Session Cookie
Credential Access	T1555	Credentials from Password Stores
Credential Access	T1552	Unsecured Credentials
Discovery	T1087	Account Discovery
Discovery	T1010	Application Window Discovery
Discovery	T1083	File and Directory Discovery
Discovery	T1120	Peripheral Device Discovery
Discovery	T1012	Query Registry
Discovery	T1518	Software Discovery
Discovery	T1082	System Information Discovery
Discovery	T1016	System Network Configuration Discovery
Discovery	T1033	System Owner/User Discovery
Discovery	T1124	System Time Discovery

Collection	T1560	Archive Collected Data
Collection	T1005	Data from Local System
Collection	T1113	Screen Capture
Command and Control	T1071	Application Layer Protocol
Command and Control	T1132	Data Encoding
Command and Control	T1041	Exfiltration over C2 Channel

Conclusion

Information Stealers like **Taurus Stealer** are **dangerous** and can cause a lot of damage to individuals and organizations (privacy violation, leakage of confidential information, etc.). Consequences vary depending on the significance of the stolen data. This goes from usernames and passwords (which could be targetted by threat actors to achieve **privilege escalation** and **lateral movement**, for example) to information that grants them immediate **financial profit**, such as cryptocurrency wallets. In addition, stolen email accounts can be used to send spam and/or distribute **malware**. As has been seen throughout the analysis, Taurus Stealer looks like an **evolving** malware that is still being updated (improving its code by adding features, more obfuscation and bugfixes) as well as it's Panel, which keeps having updates with more improvements (such as adding filters for the results coming from the malware or adding statistics for the loader). The fact the malware is being **actively used** in the wild suggests that it will continue evolving and adding more features and protections in the future, especially as customers have an open dialog channel to request new software to target or to suggest improvements to improve functionality. For more details about how we reverse engineer and analyze malware, visit our targeted malware module page.

IOCs

Hashes Taurus Stealer (earlier version):

- Packed: 4a30ef818603b0a0f2b8153d9ba6e9494447373e86599bcc7c461135732e64b2
- Unpacked: ddc7b1bb27e0ef8fb286ba2b1d21bd16420127efe72a4b7ee33ae372f21e1000

Taurus Stealer (analyzed sample):

- Packed: 2fae828f5ad2d703f5adfacde1d21a1693510754e5871768aea159bbc6ad9775
- Unpacked: d6987aa833d85ccf8da6527374c040c02e8dfbdd8e4e4f3a66635e81b1c265c8

C2 64[.]225[.]22[.]106 (earlier Taurus Stealer) dmpfdmserv275[.]xyz (analyzed Taurus Stealer)

References

Cyber Intelligence Infoblox, "WordyThief: A Malicious Spammer", October, 2020. [Online]. Available: https://docs.apwg.org/ecrimeresearch/2020/56_Wordythief-AMaliciousSpammer_20201028.pdf [Accessed April 25, 2021]

fumik0, "Predator The Thief: In-depth analysis (v2.3.5)", October, 2018. [Online]. Available: https://fumik0.com/2018/10/15/predator-the-thief-in-depth-analysis-v2-3-5/ [Accessed April 25, 2021]

fumik0, "Let's play (again) with Predator the thief", December, 2019. [Online]. Available: https://fumik0.com/2019/12/25/lets-play-again-with-predator-the-thief/ [Accessed April 25, 2021]

Threat Intelligence Team, "Taurus Project stealer now spreading via malvertising campaign", September, 2020. [Online]. Available: https://blog.malwarebytes.com/malwarebytes-news/2020/09/taurus-project-stealer-now-spreading-via-malvertising-campaign/ [Accessed April 25, 2021]

Avinash Kumar, Uday Pratap Singh, "Taurus: The New Stealer in Town", June, 2020. [Online] Available: https://www.zscaler.com/blogs/security-research/taurus-new-stealer-town [Accessed April 25, 2021]

Joxean Koret, "Antiemulation Techniques (Malware Tricks II)", February, 2010. [Online] Available: http://joxeankoret.com/blog/2010/02/23/antiemulation-techniques-malware-tricks-ii/ [Accessed April 25, 2021]