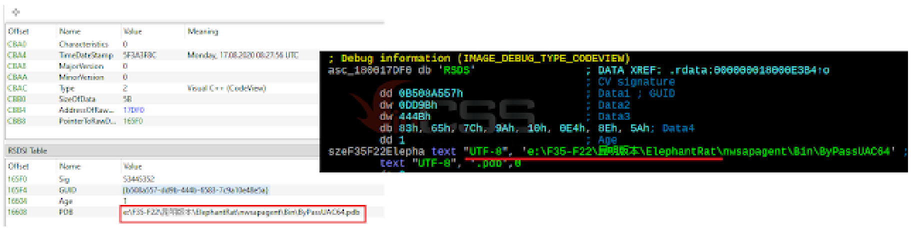
[RE020] ElephantRAT (Kunming version): our latest discovered RAT of Panda and the similarities with recently Smanager RAT

blog.vincss.net/2021/02/re020-elephantrat-kunming-version-our-latest-discovered-RAT-of-Panda.html

Recently, ESET published a report on a supply chain attack targeting software company BigNox, taking advantage of the update mechanism of the NoxPlayer software - an Android emulator on PC and Mac. This software is used by many gamers in Vietnam as well as in all over the world. ESET has named this campaign Operation NightScout. With the assessment that Vietnam can also have many people infected due to a large number of users, we have begun to investigate and analyze further.

Based on the hashes of the samples provided by ESET, we have not only re-analyzed them, but also digged deeper. We found many points that the ESET did not mention in their report. At the same time, we have found a number of similarities and relationships between these samples and those used in the last campaign against the Vietnam Government CertificationAuthority as well as a large Vietnamese corporation that we already mentioned. Not only that, we have discovered a new RAT, which is named **ElephantRat**.



"昆明版本" means "Kunming version"

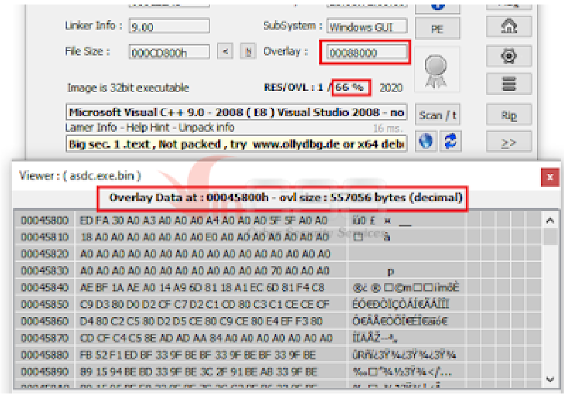
In those samples, we focus on the E45A5D9B03CFBE7EB2E90181756FDF0DD690C00C sample and analyze through to embedded PE(s) and execute fileless on memory to the very end. Looking for similarities in the binary pattern, we discovered another pattern that is being used by hackers recently, similar to the one used in our attack on large corporations in Vietnam.

Because the hacker does not use much C++ in OOP Style, the main tool we use is still IDA and the following main plugins: *FindCrypt3, SusanRTTI, LazyIDA*.

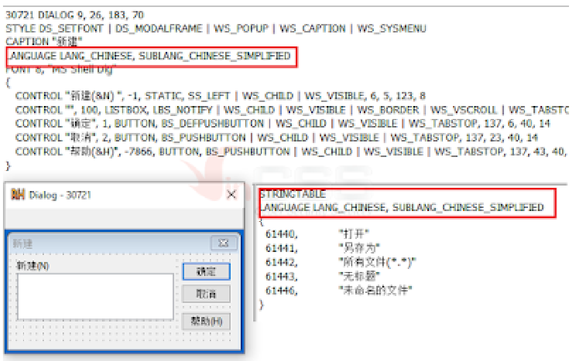
Sample E45A5D9B03CFBE7EB2E90181756FDF0DD690C00C (SHA-1), in ESET report is **UpdatePackageSilence.exe**, has:

- MD5 = 06AF27C0F47837FB54490A8FE8332E04
- SHA-256 = E76567A61F905A2825262D5F653416EF88728371A0A2FE75DDC53AAD100E6F46
- Compiler time: Wednesday, 26.08.2020 08:39:20 UTC

It is the first stage in the infection chain. The way to code, execute, and behavior like **VVSup.exe** mentioned in the previous blog post. The sample is compiled using *Visual Studio 2008 (Linker version 9.00)*. In particular, this file has a very large overlay data at the end of PE, offset 0x45800.



This Exe file is also an MFC Dialog application, except that it uses MFC version 9.0 which included in Visual Studio 2008 (*VVSup uses MFC ver 4.2, included in Visual Studio 6*), ANSI mode. And the Visual Studio that hacker used is the Chinese version, so all default resource items that MFC Wizard automatically generates are in Chinese.

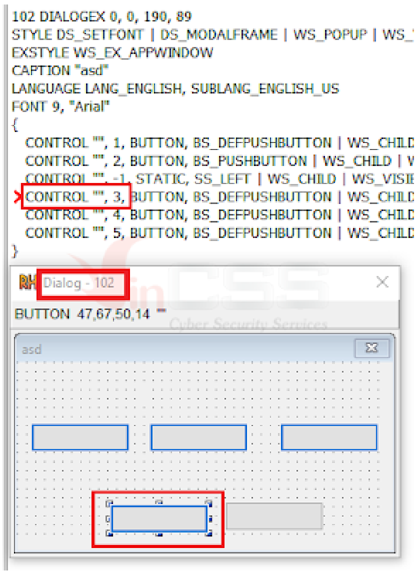


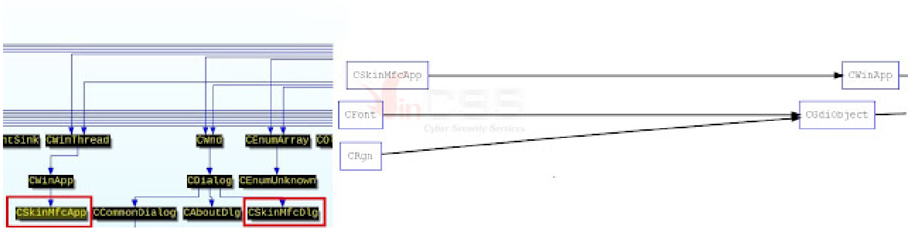

```
FILETYPE 0x1
{
BLOCK "StringFileInfo"
{
        BLOCK "040904b0"
        {
                VALUE "Comments", "asd"
                VALUE "CompanyName", "asd"
                VALUE "FileDescription", "asd"
                VALUE "FileVersion", "17, 12, 27, 1"
                VALUE "InternalName", "asdc"
                VALUE "LegalCopyright", "asdc"
                VALUE "LegalTrademarks", "asdc"
                VALUE "OriginalFilename", "asdc.EXE"
                VALUE "PrivateBuild", "asd"
                VALUE "ProductName", "asd"
                VALUE "ProductVersion", "17, 12, 27, 1"
                VALUE "SpecialBuild", "asdc"
        }
}
```

Dialog 30721 is the MFC's default "New Item" Dialog, the StringTable ID from 60000 is also the default resource string ID of MFC. Hacker randomly entered the About Wizard named Exe and version number. The dialog that the hacker added was reseted to English. Main Dialog has ID = 102, About Dialog has ID = 100.

Control IDs 1 and 2 are the default MFC Wizard generates, which are IDOK and IDCANCEL. Buttons 3 (ID_ABORT), 4 (ID_RETRY), 5 (ID_IGNORE) are added by hacker. We need to notice Button ID_ABORT 3. The main icon of the app (ID 1) is used by the hacker using the icons that installers often use.

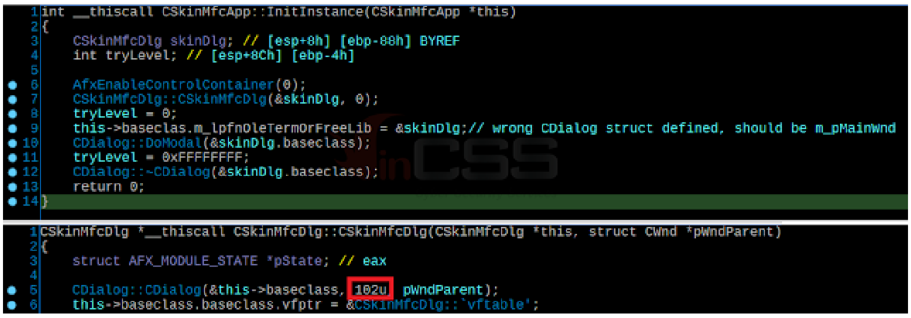
SusanRTTI gives us the class flowchart of the app. The figure below is part of the flowchart.





Using LazyIDA's Search features, with CSkinMfcApp and CSkinMfcDlg, we just found this one link from China, which mention about skin dialog creation technique for MFC app.

With the addition of the CRgn class, we can believe that hackers took this entire project and made a few changes. The execution mechanism of a dialog-type MFC app, we released in the previous blog post, you can review but in this blog post, we just focus on the main point.

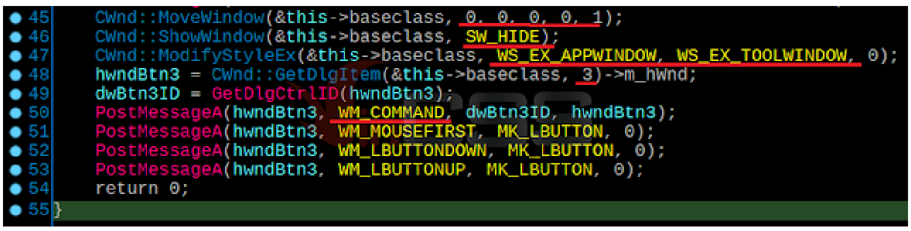


```
int __thiscall CSkinMfcApp::InitInstance(CSkinMfcApp *this)
{
  CSkinMfcDlg skinDlg; // [esp+8h] [ebp-88h] BYREF
  int tryLevel; // [esp+8Ch] [ebp-4h]

  AfxEnableControlContainer(0);
  CSkinMfcDlg::CSkinMfcDlg(&skinDlg, 0);
  tryLevel = 0;
  this->baseclas.m_lpfnOleTermOrFreeLib = &skinDlg;// wrong CDialog struct defined, should be m_pMainWnd
  CDialog::DoModal(&skinDlg.baseclass);
  tryLevel = 0xFFFFFFFF;
  CDialog::~CDialog(&skinDlg.baseclass);
  return 0;
}
```

```
CSkinMfcDlg *__thiscall CSkinMfcDlg::CSkinMfcDlg(CSkinMfcDlg *this, struct CWnd *pWndParent)
{
  struct AFX_MODULE_STATE *pState; // eax

  CDialog::CDialog(&this->baseclass, 102u, pWndParent);
  this->baseclass.baseclass.vfptr = &CSkinMfcDlg::`vftable';
```

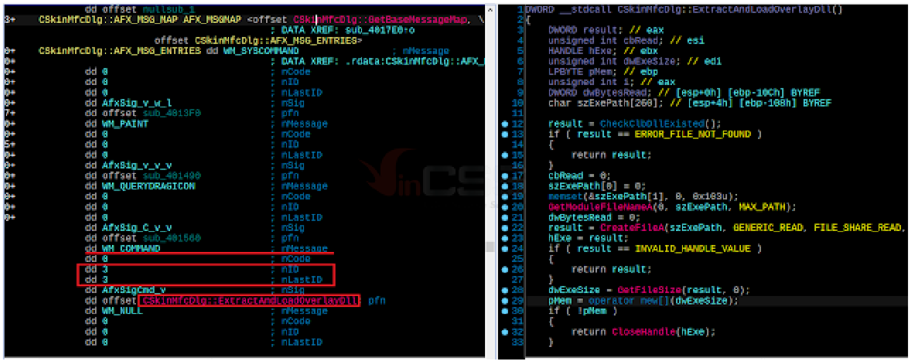
In the OnInitDialog method of CSkinMfcDlg, the hacker has changed the call to the main infection task and added code:

- Resize Dialog to 0
- Hide Dialog
- Change the style of Dialog to not show the Windows Taskbar
- Post WM_COMMAND to Button ID 3
- Hackers are also careful to simulate adding user left mouse to click on Button ID 3



```
  CWnd::MoveWindow(&this->baseclass, 0, 0, 0, 0, 1);
  CWnd::ShowWindow(&this->baseclass, SW_HIDE);
  CWnd::ModifyStyleEx(&this->baseclass, WS_EX_APPWINDOW, WS_EX_TOOLWINDOW, 0);
  hwndBtn3 = CWnd::GetDlgItem(&this->baseclass, 3)->m_hWnd;
  dwBtn3ID = GetDlgCtrlID(hwndBtn3);
  PostMessageA(hwndBtn3, WM_COMMAND, dwBtn3ID, hwndBtn3);
  PostMessageA(hwndBtn3, WM_MOUSEFIRST, MK_LBUTTON, 0);
  PostMessageA(hwndBtn3, WM_LBUTTONDOWN, MK_LBUTTON, 0);
  PostMessageA(hwndBtn3, WM_LBUTTONUP, MK_LBUTTON, 0);
  return 0;
}
```

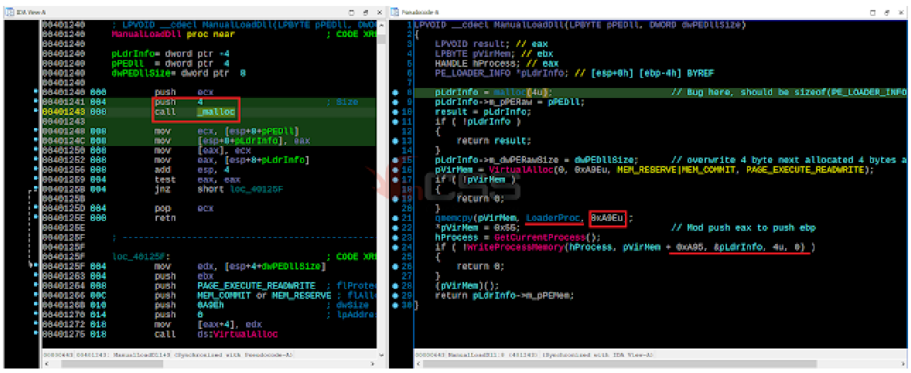
At the AFX_MSGMAP of CSkinMfcDlg, we found the function that performs the primary infection task.



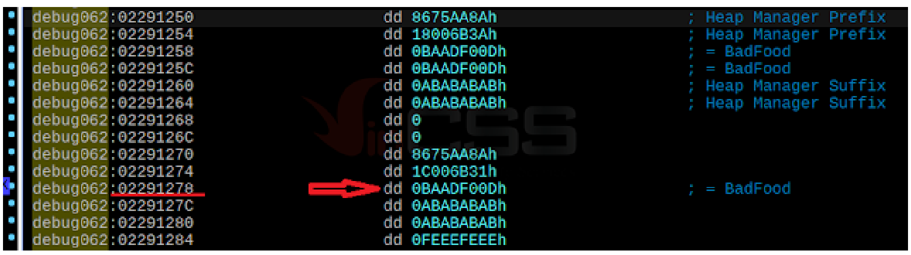
When ExtractAndLoadOverlayDll is called, the hacker will first check if the app has read permission to the Windows\System32 directory and check if the clb.dll file exists. Clb.dll is Windows file - Column ListBox. Then the hacker opens the Exe, reads the Overlay data at offset 0x45800 and xor with 0xA0 to decrypt the PE file is a DLL. It will then manually load this DLL to memory, starting a long series of manually load fileless PE.

At this ManualLoadDll function, we discovered a hacker programming error. Specifically, Malloc does not have free and wrong code: malloc(sizeof(PE_LOADER_INFO)) (16 bytes) to malloc(sizeof(pLdrInfo)) (4 bytes). The PE_LOADER_INFO struct that we renamed, including 4 data members, size is 16 bytes.

```
Offset Size struct PE_LOADER_INFO
            {
0000   0004     LPBYTE m_pPERaw;
0004   0004     DWORD  m_dwPERawSize;
0008   0004     LPBYTE m_pPEMem;
000C   0004     DWORD  m_dwPEMemSize;
       0010 };
```



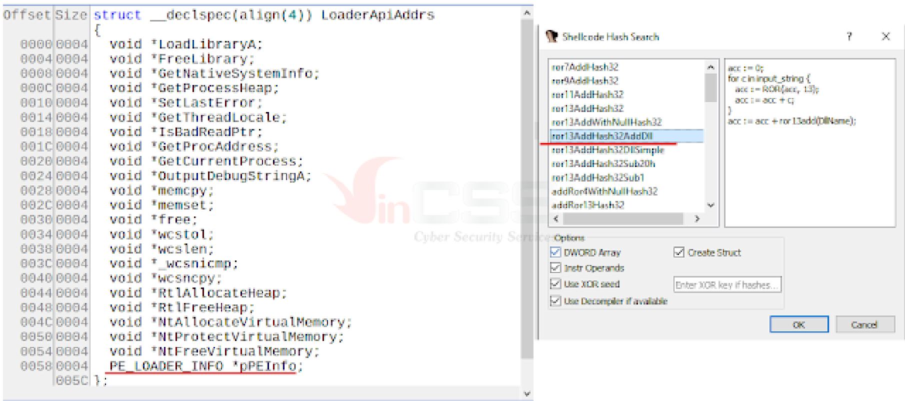
After alloc 4 byte:



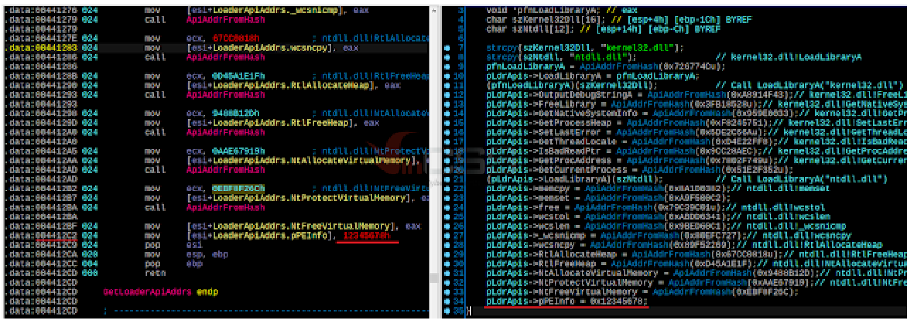
After overwrite:

About values 0xBAADFOOD and 0xABABABAB ... of VC RTL and Windows Heap Manager, you can read more here. The functions that manually (reflective) load overlay Dll functions are compiled into a shellcode array of bytes, embedded in the .data section, and have a total size of 0xA9E. Start at the address of the LoaderProc function: .data:00440830. 0xA95 is the RVA of constant 0x12345678, which will be overwrite by the memory contents of the variable pLdrInfo after being saved by malloc, sizeof(pointer) = 4 (x86). The first byte of the LoaderProc function will be modified to 0x55 = push ebp

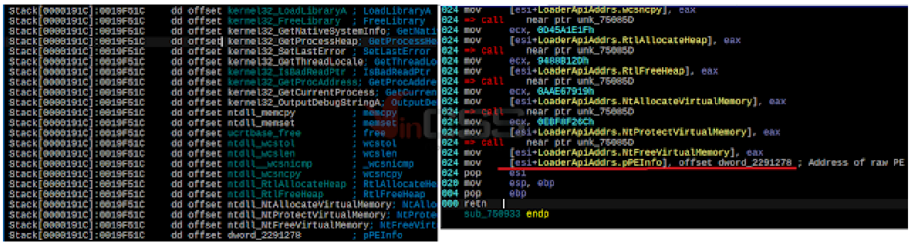
GetLoaderApiAddrs function retrieves the API addresses from kernel32.dll and ntdll.dll into a struct containing pointers to those API functions. The algorithm used to calculate the hash value from the exported API name is ROR13, which is commonly used in Metasploit. Readers can use the plugin shellcode_hashes_search_plugin.py in FireEye's Flare_ida toolkit to automatically determine the name of the API function, select the hash function ror13AddHash32AddDll. This struct has been redefined as follows:



GetLoaderApiAddrs function:



The value of this struct variable in the LoaderProc function after the GetLoaderApiAddrs function is called and returned.



The remarkable point is the manual/reflective load feature is used directly with Ntdll.dll native functions, not through kernel32 functions. This is possible to avoid detecting by the AV/EDR hook kernel32.dll. And it also goes with other samples and later fileless PE(s).

The code of ReflectiveLoadDll is similar to the other manually load/reflective open source. We will not talk about it again. Searching on Github, Google, and VirusTotal for GetLoaderApiAddrs function, we found no such function. So we think this is a manually/reflectively load library that this group wrote themselves and didn't use any open source.

At this point, the Overlay Dll has been loaded and the execution flows directly into the OEP of the Dll. The parent exe does not exit immediately like VVSup.exe, the fileless child dlls will call ExitProcess or TerminateProcess later.

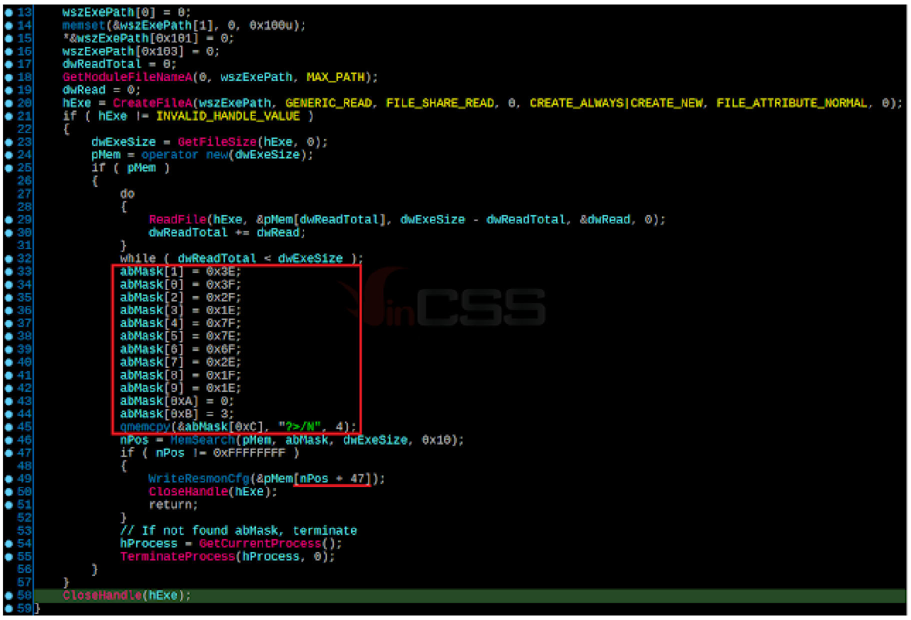
We temporarily move to another sample that the ESET report mentioned has SHA1 = 5732126743640525680C1F9460E52D361ACF6BB0. This sample was compiled using Visual Studio 2012, built on 11/16.2020 08:35:32 UTC, also an MFC app, however no longer Dialog app but a Doc - View app, using new MFC Ribbon classes. As a result, the amount of code and classes are bigger, and it is possible that the first stage uses the latest MFC of this group. Hackers no longer rely on extrac32.exe to extract embedded Cab files, but write a CCabinet class using Cabinet API functions available from Windows to unpack.

PDB path = "C:\Users\enWin7x64\Desktop\XActor\CreateServer_src\XActorCreateServer\DATA_RES\CommandoLoader\mfeesp\Release\mfeesp.pdb". The executable code that extracts two cab files from the resource is written directly into the InitInstance function of the CmfeespApp class. And LBTServ.dll malware file is extracted from the cab file is a Dll, written in Delphi and built using Embarcadero's latest RAD Studio 10.4 Sydney. This could be a shift to another language, compiler/IDE for future malware development of this group. For the purposes and scope of this article, we do not present these samples.

Back on the above Dll overlay, after extracting, we have a DLL with the following information:

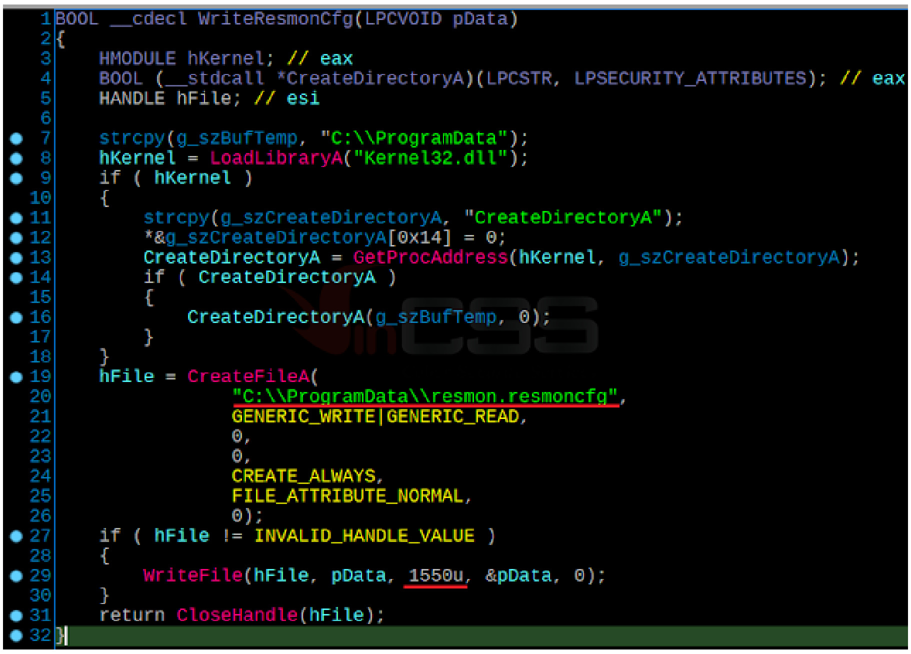
- Size = 557,056 bytes
- MD5 = 054E07CB00E9B21786E2815E9B43CDA9
- SHA256 = 8BF3DF654459B1B8F553AD9A0770058FD2C31262F38F2E8BA12943F813200A4D
- Compile time: Monday, 17.08.2020 09:56:11 UTC
- Visual Studio 6
- There is no PDB path and export, so the original DLL name could not be determined.

The size of the .data section is large, after running FindCrypt3, we found that there were large data. All the main tasks of this Dll reside entirely within the DllMain function. When DllMain is called with fdwReason other than DLL_PROCESS_ATTACK, hacker checks whether the Dll can OpenProcess with System Process (PID = 4) with the highest permissions 0x1F0FFF or not. If OpenProcess succeeds, it will return TRUE and do nothing next. Then hacker read from the parent Exe, use the MemSearch function as in VVSup.exe to find and extract the C&C information and save it into a file *C:\ProgramData/resmon.resmoncfg*. The small difference is that VVSup uses MemSearch to get the C&C info from the parent to write in the child's Dll. And here is the Dll child search back from the parent Exe.



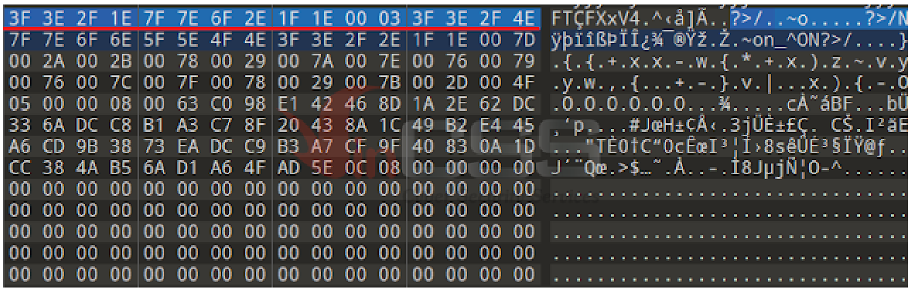
```
    wszExePath[0] = 0;
    memset(&wszExePath[1], 0, 0x100u);
    *&wszExePath[0x101] = 0;
    wszExePath[0x103] = 0;
    dwReadTotal = 0;
    GetModuleFileNameA(0, wszExePath, MAX_PATH);
    dwRead = 0;
    hExe = CreateFileA(wszExePath, GENERIC_READ, FILE_SHARE_READ, 0, CREATE_ALWAYS|CREATE_NEW, FILE_ATTRIBUTE_NORMAL, 0);
    if ( hExe != INVALID_HANDLE_VALUE )
    {
        dwExeSize = GetFileSize(hExe, 0);
        pMem = operator new(dwExeSize);
        if ( pMem )
        {
            do
            {
                ReadFile(hExe, &pMem[dwReadTotal], dwExeSize - dwReadTotal, &dwRead, 0);
                dwReadTotal += dwRead;
            }
            while ( dwReadTotal < dwExeSize );
            abMask[1] = 0x3E;
            abMask[0] = 0x3F;
            abMask[2] = 0x2F;
            abMask[3] = 0x1E;
            abMask[4] = 0x7F;
            abMask[5] = 0x7E;
            abMask[6] = 0x6F;
            abMask[7] = 0x2E;
            abMask[8] = 0x1F;
            abMask[9] = 0x1E;
            abMask[0xA] = 0;
            abMask[0xB] = 3;
            qmemcpy(&abMask[0xC], "?>/N", 4);
            nPos = MemSearch(pMem, abMask, dwExeSize, 0x10);
            if ( nPos != 0xFFFFFFFF )
            {
                WriteResmonCfg(&pMem[nPos + 47]);
                CloseHandle(hExe);
                return;
            }
            // If not found abMask, terminate
            hProcess = GetCurrentProcess();
            TerminateProcess(hProcess, 0);
        }
    }
    CloseHandle(hExe);
}
```

Write C&C info to resmon.resmoncfg file



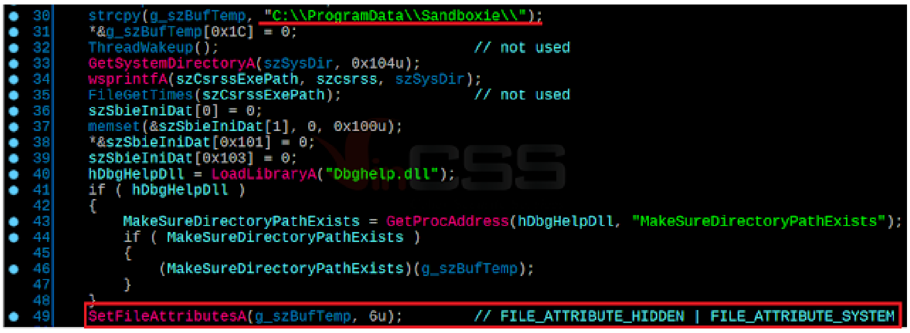
```
BOOL __cdecl WriteResmonCfg(LPCVOID pData)
{
    HMODULE hKernel; // eax
    BOOL (__stdcall *CreateDirectoryA)(LPCSTR, LPSECURITY_ATTRIBUTES); // eax
    HANDLE hFile; // esi

    strcpy(g_szBufTemp, "C:\\ProgramData");
    hKernel = LoadLibraryA("Kernel32.dll");
    if ( hKernel )
    {
        strcpy(g_szCreateDirectoryA, "CreateDirectoryA");
        *&g_szCreateDirectoryA[0x14] = 0;
        CreateDirectoryA = GetProcAddress(hKernel, g_szCreateDirectoryA);
        if ( CreateDirectoryA )
        {
            CreateDirectoryA(g_szBufTemp, 0);
        }
    }
    hFile = CreateFileA(
              "C:\\ProgramData\\resmon.resmoncfg",
              GENERIC_WRITE|GENERIC_READ,
              0,
              0,
              CREATE_ALWAYS,
              FILE_ATTRIBUTE_NORMAL,
              0);
    if ( hFile != INVALID_HANDLE_VALUE )
    {
        WriteFile(hFile, pData, 1550u, &pData, 0);
    }
    return CloseHandle(hFile);
}
```

Byte array is the mask for searching is "3F 3E 2F 1E 7F 7E 6F 2E 1F 1E 00 03 3F 3E 2F 4E". File size of resmon.resmoncfg file is 1550 bytes, copy the content from mask offset + 47.

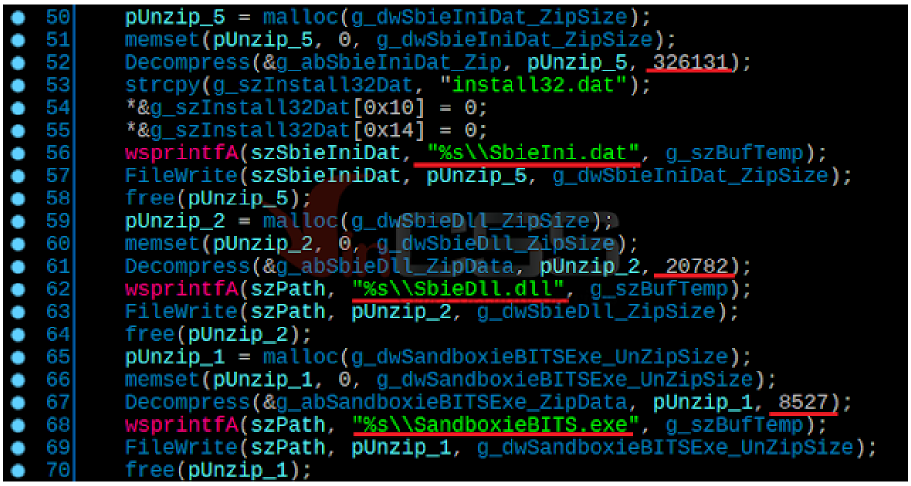


```
3F 3E 2F 1E 7F 7E 6F 2E 1F 1E 00 03 3F 3E 2F 4E
7F 7E 6F 6E 5F 5E 4F 4E 3F 3E 2F 2E 1F 1E 00 7D
00 2A 00 2B 00 78 00 29 00 7A 00 7E 00 76 00 79
00 76 00 7C 00 7F 00 78 00 29 00 7B 00 2D 00 4F
05 00 00 08 00 63 C0 98 E1 42 46 8D 1A 2E 62 DC
33 6A DC C8 B1 A3 C7 8F 20 43 8A 1C 49 B2 E4 45
A6 CD 9B 38 73 EA DC C9 B3 A7 CF 9F 40 83 0A 1D
CC 38 4A B5 6A D1 A6 4F AD 5E 0C 08 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

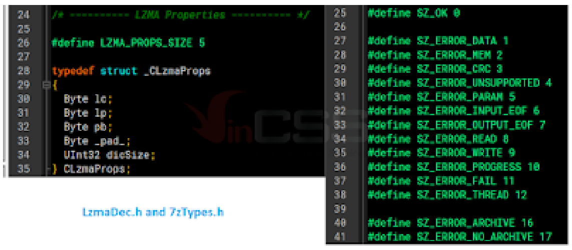
Hackers also use the MakeSureDirectoryPathExists export function from dbghelp.dll to create directory, same as VVSup, and also use a lot of global variables, strings, and arrays. There is a lot of redundant code such as getting *CreationTime, LastAccessTime, LastWriteTime* of the csrss.exe file system that is not used, and initializing unused strings. Create Sandboxie directory, attribute hidden and system

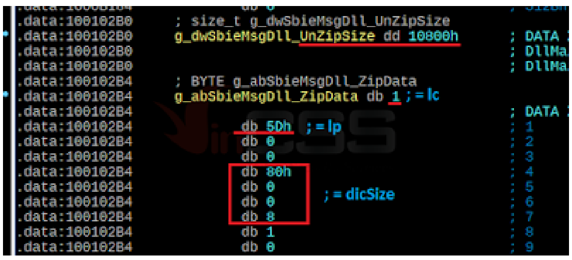


Dll continues to unpack embedded data in DLL into files: SbieIni.dat, SbieDll.dll, SandboxieBITS.exe and saves them into *C:\ProgramData\Sanboxie*.



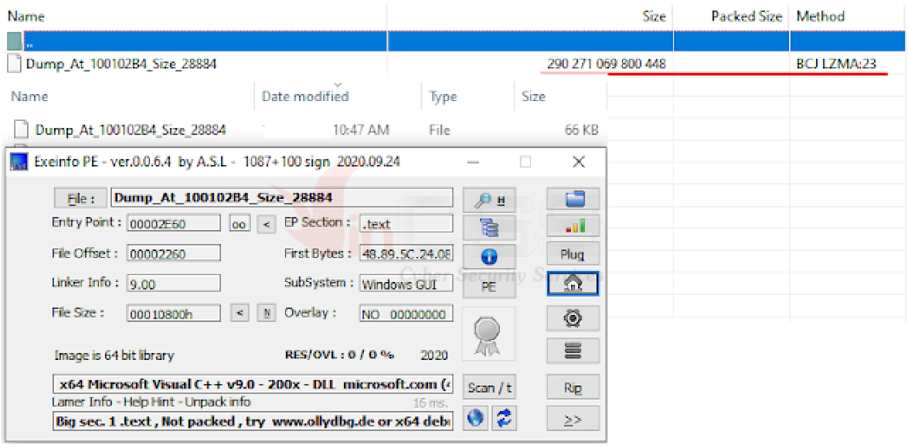
The compression and decompression algorithm that hackers use here is the LZMA algorithm. LZMA's SDK can be downloaded and referenced here. The LZMA algorithm identifier used is LZMA_PROPS_SIZE = 5 and the first 8 bytes of the struct CLzmaProps at the beginning of the data compressed.





The uncompressed function, the size of the compressed data is passed in minus 4, the size value of the uncompressed data region DWORD immediately preceded the data compressed.

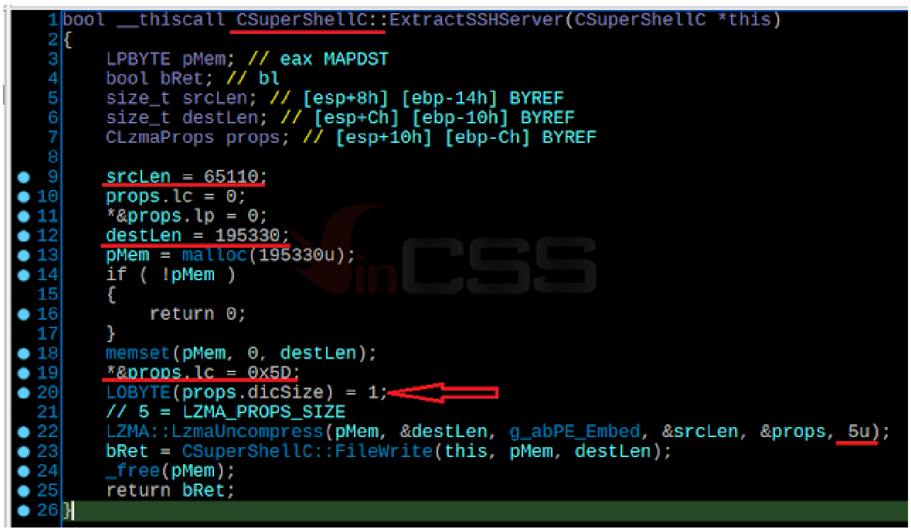
But especially the hacker has changed in the code of this LZMA algorithm, so if we statically extract these compressed data areas according to the above information then when decompressing with 7z or tool, lib will normally error, but It is still possible to extract the first area of the correct data compared to the results when debugging and dumping.



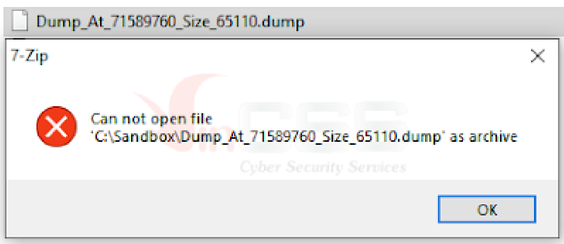
Using this custom LZMA compression algorithm, we also found in a new sample SManager RAT plugin, uploaded to the first VirusTotal on 23/01/2021:

- MD5 = 0603145EFAD6A63F52B6D5161CC5E5AE
- SHA256 = 321045519CC3A50CE7948C33C6BBC837B063CD878F8C2CE67DC8DE0825515E10
- File name: SuperShellC_x86.dll

In this DLL file, the CSuperShellC class has the task of extracting an embedded Exe, the original name is ssh_server.exe.



This LZMA algorithm continues to be improved by hackers, so with static dump we could not open, we had to debug and dump it.



Return to Overlay Dll, after extracting 3 files x86 files into *C:\ProgramData\Sandboxie* folder, Dll continues to check if itself has write permissions to the System32 directory and target Windows operating system is x64 or not. If all is passed, Dll will extract two additional files SbieMsg.dll and SbieMsg.dat into that directory.

```
if ( HavePermission() )
{
    if ( IsX64() )
    {
        szSbieMsgDatPath[0] = 0;
        memset(&szSbieMsgDatPath[1], 0, 0x100u);
        *&szSbieMsgDatPath[0x101] = 0;
        szSbieMsgDatPath[0x103] = 0;
        pUnzip_3 = malloc(g_dwSbieMsgDll_UnZipSize);
        memset(pUnzip_3, 0, g_dwSbieMsgDll_UnZipSize);
        Decompress(&g_abSbieMsgDll_ZipData, pUnzip_3, 0x70D8);
        strcpy(g_szInstall64DllPath, "install64.dll");// not used
        *&g_szInstall64DllPath[0x10] = 0;
        *&g_szInstall64DllPath[0x14] = 0;
        wsprintfA(szSbieMsgDllPath, "%s\\SbieMsg.dll", g_szBufTemp);
        FileWrite(szSbieMsgDllPath, pUnzip_3, g_dwSbieMsgDll_UnZipSize);
        free(pUnzip_3);
        pUnzip_4 = malloc(g_dwSbieMsgDa_UnZipSize);
        memset(pUnzip_4, 0, g_dwSbieMsgDa_UnZipSize);
        Decompress(&g_abZipData_4, pUnzip_4, 114746);
        strcpy(g_szInstall64DatPath, "install64.dat");// not used
        *&g_szInstall64DatPath[0x10] = 0;
        *&g_szInstall64DatPath[0x14] = 0;
        wsprintfA(szSbieMsgDatPath, "%s\\SbieMsg.dat", g_szBufTemp);
        FileWrite(szSbieMsgDatPath, pUnzip_4, g_dwSbieMsgDa_UnZipSize);
        free(pUnzip_4);
        ExecuteAndSelfDelete("ByPassUAC", "rundll32.exe C:\\ProgramData\\Sandboxie\\SbieMsg.dll,installsvc");
        return 1;
    }
    ExecuteAndSelfDelete("ByPassUAC", szPath);
}
else
{
    ExecuteAndSelfDelete("InsertS", szPath);
}
return 1;
```

At the HavePermission function, hacker will create a random file in System32, the first name is wmkawe_ and the content is only one line of text: "*Stupid Japanese*".

```
strcpy(szMask, "Stupid Japanese");
bResult = 0;
dwBytesWritten = 0;
GetSystemDirectoryA(szSysDir, MAX_PATH);
dwTick = GetTickCount();
wsprintfA(szWmkaveDatPath, "%s\\wmkawe_%d.data", szSysDir, dwTick);
hFile = CreateFileA(szWmkaveDatPath, GENERIC_ALL, 0, 0, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, 0);
GetLastError();                       // bug, unused
if ( hFile == INVALID_HANDLE_VALUE )
{
    return 1;
}
if ( !WriteFile(hFile, szMask, strlen(szMask), &dwBytesWritten, 0) )
{
    bResult = 1;
}
CloseHandle(hFile);
```

In addition, the hacker also checks to see if there are two files with the same random name wmkawe_xxx.data in the two folders: "%LOCALAPPDATA%\VirtualStore\Windows\System32\" and "% LOCALAPPDATA%\VirtualStore\Windows\SysWOW64\", if any, it will be deleted. The function will check in the targeted machine OS is Windows, hacker doesn't use the usual IsWow64Process API function, but uses the GetNativeSystemInfo API function.

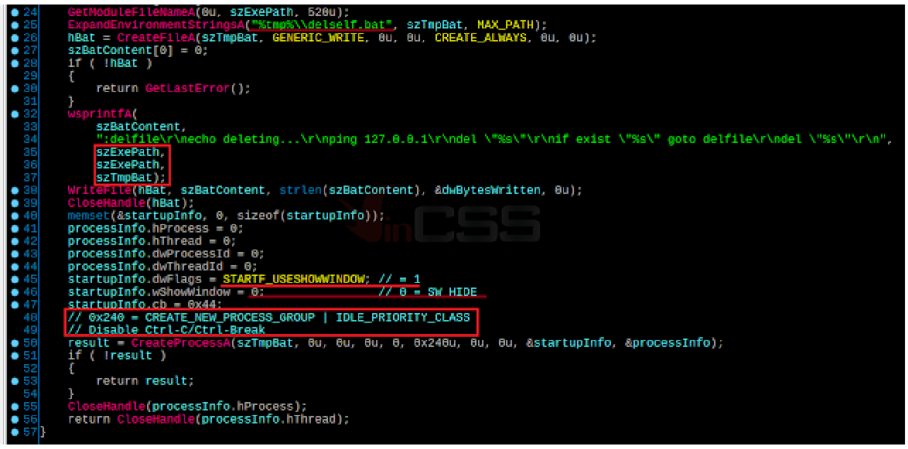
```
BOOL __stdcall IsX64()
{
    HMODULE hKernel32; // eax
    void (__stdcall *GetNativeSystemInfo)(LPSYSTEM_INFO); // eax
    BOOL result; // eax
    struct _SYSTEM_INFO sysInfo; // [esp+4h] [ebp-24h] BYREF

    hKernel32 = GetModuleHandleA("kernel32.dll");
    GetNativeSystemInfo = GetProcAddress(hKernel32, "GetNativeSystemInfo");
    result = 0;
    if ( !GetNativeSystemInfo )
    {
        return result;
    }
    GetNativeSystemInfo(&sysInfo);
    if ( sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_AMD64
      || sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_IA64 )
    {
        result = 1;
    }
    return result;
}
```

After extracting two more files SbieMsg.dat and SbieMsg.dll, Dll will load SbieMsg.dll by using rundll32.exe utility of Windows, call the exported function is "installsvc", pass the parameter as "ByPassUAC".

If it's not Windows x64, SandboxieBITS.exe will be called with the parameter "ByPassUAC" aswell. And if there is no write permission to System32, the Dll just calls SandboxieBITS.exe with the parameter "InsertS". Finally, Dll will create bat file to delete parent Exe itself and the bat file itself and then exit parent Exe.

```
BOOL __cdecl ExecuteAndSelfDelete(const char *pszParam, const char *pszExePath)
{
    HANDLE hProcess; // eax
    CHAR szCmdLine[260]; // [esp+0h] [ebp-104h] BYREF

    wsprintfA(szCmdLine, "%s %s", pszExePath, pszParam);
    CreateProcessA(0, szCmdLine, 0, 0, 0, CREATE_NO_WINDOW, 0, "C:\\", &startupInfo, &processInfo);
    Sleep(1000u);
    SelfDelete();
    hProcess = GetCurrentProcess();
    return TerminateProcess(hProcess, 0);
}
```

The SelfDelete execute cmd.exe function in the hidden window, idle priority and disable Ctrl-C/Ctrl-Break.



```
GetModuleFileNameA(0u, szExePath, 520u);
ExpandEnvironmentStringsA("%tmp%\\delself.bat", szTmpBat, MAX_PATH);
hBat = CreateFileA(szTmpBat, GENERIC_WRITE, 0u, 0u, CREATE_ALWAYS, 0u, 0u);
szBatContent[0] = 0;
if ( !hBat )
{
    return GetLastError();
}
wsprintfA(
    szBatContent,
    ":delfile\r\necho deleting...\r\nping 127.0.0.1\r\ndel \"%s\"\r\nif exist \"%s\" goto delfile\r\ndel \"%s\"\r\n",
    szExePath,
    szExePath,
    szTmpBat);
WriteFile(hBat, szBatContent, strlen(szBatContent), &dwBytesWritten, 0u);
CloseHandle(hBat);
memset(&startupInfo, 0, sizeof(startupInfo));
processInfo.hProcess = 0;
processInfo.hThread = 0;
processInfo.dwProcessId = 0;
processInfo.dwThreadId = 0;
startupInfo.dwFlags = STARTF_USESHOWWINDOW; // = 1
startupInfo.wShowWindow = 0;             // 0 = SW_HIDE
startupInfo.cb = 0x44;
// 0x240 = CREATE_NEW_PROCESS_GROUP | IDLE_PRIORITY_CLASS
// Disable Ctrl-C/Ctrl-Break
result = CreateProcessA(szTmpBat, 0u, 0u, 0u, 0, 0x240u, 0u, 0u, &startupInfo, &processInfo);
if ( !result )
{
    return result;
}
CloseHandle(processInfo.hProcess);
return CloseHandle(processInfo.hThread);
}
```

At this point, stage one of the infection is complete. Stage 2 starts from executing SandboxieBITS.exe or SbieMsg.dll (x64) run as a service Dll.

We would like to stop here and publish the following sections when the time appropriate.

We wish you a happy new year!

Click here for Vietnamese version.

Truong Quoc Ngan (aka HTC)

Malware Analysis Expert - VinCSS (a member of Vingroup)