

[RE023] Quick analysis and removal tool of a series of new malware variant of Panda group that has recently targeted to Vietnam VGCA

blog.vincss.net/2021/07/re023-quick-analysis-and-removal-tool-series-of-new-malware-variant-of-Panda-group-that-has-recently-targeted-to-Vietnam-VGCA.html

Through continuous cyber security monitoring and hunting malware samples that were used in the attack on **Vietnam Government Certification Authority**, and they also have attacked a large corporation in Vietnam since 2019, we have discovered a series of new variants of the malware related to this group. Readers can re-read and compare the information below with the previously analyzed article: [\[RE018-2\] Analyzing new malware of China Panda hacker group used to attack supply chain against Vietnam Government Certification Authority - Part 2](#)

I. Analysing loaders

Sample [2b15479eb7ec43f7a554dce40fe6a4263a889ba58673b7490a991e7d66703bc8](#) was discovered by us on [VirusTotal](#) on **11/06/2021**, and was submitted from Vietnam:



The remarkable point in this file is the **.NLS** (*National Language Support*) extension, but it's exactly a **DLL PE64**. We conduct an in-depth analysis of this sample and determined it seems to be crafted by the same hacker who wrote and built `smanager_ssl.dll`, `msiscsi.dll`, `verifierpr.dll`, `wercplsupport.dll`.

- Hash:
2B15479EB7EC43F7A554DCE40FE6A4263A889BA58673B7490A991E7D66703BC8
- Compiled time: Tuesday, 04.08.2020 06:48:49 UTC
- Original DLL: `DIISvchDtchX64.bin`
- Malicious file: `C_20253.NLS`, in `\Windows\System32`
- Visual Studio version: 2015, linker 14.0, update 3
- Coding language: C

RichID Information:

@comp.id	Counter	Version	Tool	Toolset
0x00FF5E92	1	14.0.24210	CVTRES, RES to COFF	VS 2015 14.0 Upd 3
0x00010000	109		IAT Entry	
0x01005E97	1	14.0.24215	Linker, Exports in DEF file	VS 2015 14.0 Upd 3 SR4
0x00937809	7	9.0.30729	Linker, Import Library	VS 2008 9.0 SP1
0x01025E97	1	14.0.24215	Linker, Link	VS 2015 14.0 Upd 3 SR4
0x00F19CB4	4	12.10.40116	MASM, ASM COFF	VS 2013 12.10
0x01035E3B	7	14.0.24123	MASM, ASM COFF	VS 2015 14.0 Upd 3 RC
0x00F29CB4	13	18.10.40116	UTC CL, C COFF	VS 2013 12.10
0x01045E3B	16	19.0.24123	UTC CL, C COFF	VS 2015 14.0 Upd 3 RC
0x01085E97	11	19.0.24215	UTC CL, C OBJ (LTCG)	VS 2015 14.0 Upd 3 SR4
0x00F39CB4	118	18.10.40116	UTC CL, C++ COFF	VS 2013 12.10
0x01055E3B	20	19.0.24123	UTC CL, C++ COFF	VS 2015 14.0 Upd 3 RC

You can compare with the RichID information in Figure 3 of this [article](#). Attackers used impersonating NLS in the Windows\System32 and Windows\SysWow64 folders, which contain config and C&C info during the attack on a large Vietnam corporation ([Figure4](#)). After that, attackers upgraded in April 2020 to real PE(s) to perform other tasks.

DllSvchDtchX64.bin is written as a service Dll, the code and style is exactly like the code of smanager_ssl.dll and wercplsupport.dll. The ServiceMain, SvcCtrlHandler, and SetSvcStatus functions are all the same.

ServiceMain function (compare with [Figure5](#)):

```

1 void __fastcall ServiceMain(int dwArgc, const LPWSTR *lpwszArgv)
2 {
3     SERVICE_STATUS_HANDLE hSvcStatus; // rax
4     struct _SERVICE_STATUS statusStart; // [rsp+20h] [rbp-258h] BYREF
5     struct _SERVICE_STATUS statusRunning; // [rsp+40h] [rbp-238h] BYREF
6     WCHAR ServiceName[256]; // [rsp+60h] [rbp-218h] BYREF
7
8     if ( dwArgc >= 0 )
9     {
10        memset(ServiceName, 0, sizeof(ServiceName));
11        wcsncpy(ServiceName, *lpwszArgv, 0xFFui64);
12        hSvcStatus = RegisterServiceCtrlHandlerW(ServiceName, SvcCtrlHandler);
13        g_hServiceStatus = hSvcStatus;
14        if ( hSvcStatus )
15        {
16            statusStart.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
17            statusStart.dwServiceSpecificExitCode = 0;
18            g_dwServiceState = SERVICE_START_PENDING;
19            statusStart.dwCurrentState = SERVICE_START_PENDING;
20            // 7 = SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_PAUSE_CONTINUE | SERVICE_ACCEPT_SHUTDOWN
21            *&statusStart.dwControlsAccepted = 7i64;
22            statusStart.dwCheckPoint = 1;
23            statusStart.dwWaitHint = 3000;
24            SetServiceStatus(hSvcStatus, &statusStart);
25            statusRunning.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
26            g_dwServiceState = SERVICE_RUNNING;
27            statusRunning.dwCurrentState = SERVICE_RUNNING;
28            *&statusRunning.dwControlsAccepted = 7i64;
29            *&statusRunning.dwServiceSpecificExitCode = 0i64;
30            statusRunning.dwWaitHint = 0xBB8;
31            SetServiceStatus(g_hServiceStatus, &statusRunning);
32            while ( !g_dwSvcStopped )
33            {
34                Sleep(0x3E8u);
35            }
36        }
37    }
38 }

```

Another small difference is that in addition to the global variable `g_dwServiceState`, the hacker has added another global variable, `g_dwSvcStopped` to Sleep continuously until this service of `DllSvchDtchX64.bin` was stopped by Windows. With this sample, the main task of executing malware code is not included in the `ServiceMain` function, but directly in the `DllMain` function.

The `SetSvcStatus` function (compare with [Figure 7](#) and [8](#)):

```

1 BOOL __usercall SetSvcStatus@<eax>(DWORD dwNewState@<ecx>, DWORD dwCheckPoint@<r8d>)
2 {
3     struct _SERVICE_STATUS svcStatus; // [rsp+20h] [rbp-38h] BYREF
4
5     g_dwServiceState = dwNewState;
6     svcStatus.dwCurrentState = dwNewState;
7     svcStatus.dwServiceSpecificExitCode = 0;
8     svcStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
9     *&svcStatus.dwControlsAccepted = 7i64;
10    svcStatus.dwCheckPoint = dwCheckPoint;
11    svcStatus.dwWaitHint = 0xBB8;
12    return SetServiceStatus(g_hServiceStatus, &svcStatus);
13 }

```

In the `DllMain` function, the malware decrypts the SID and Mutex name, creating a thread to execute another task. This SID and Mutex name are used in the `MainThreadProc` of the created thread.

The encrypt and decrypt encryption algorithm used by the hacker in this sample is **Salsa/Chacha20**. It can be detected by `FindCrypt3` or `Capa` of [FireEye](#).

```

3491338 salsa_ivsetup proc near
3491338 000 mov     dword ptr [rcx], 61707B65h ; ChaChaInitStates_Salsa_exp32k
3491338 000 mov     r10, rcx
3491338 000 mov     dword ptr [rcx+4], 3320646Eh ; ChaChaInitStates_Salsa_exp32k
3491340 000 mov     dword ptr [rcx+8], 79622D32h ; ChaChaInitStates_Salsa_exp32k
3491347 000 mov     dword ptr [rcx+0Ch], 6B206574h ; ChaChaInitStates_Salsa_exp32k
349134E 000 movzx  eax, byte ptr [rdx+2]
3491352 000 movzx  r8d, byte ptr [rdx+3]
3491357 000 shl     r8d, 8
349135B 000 or     r8d, eax
349135E 000 movzx  eax, byte ptr [rdx+1]
3491362 000 shl     r8d, 8
3491366 000 or     r8d, eax
3491369 000 movzx  eax, byte ptr [rdx]
349136C 000 shl     r8d, 8
3491370 000 or     r8d, eax

```

```

void __fastcall salsa_ivsetup(chacha_ctx 'ctx',
2 {
3   cryptlib.CryptLib.expand_32_byte_key, 0x10);
4   ctx->input[4] = k[4] | ((k[1] | *(k + 1) <<
5   ctx->input[5] = k[4] | ((k[5] | *(k + 3)
6   ctx->input[6] = k[6] | ((k[9] | *(k + 5)
7   ctx->input[7] = k[12] | ((k[13] | *(k + 7)
8   ctx->input[8] = k[16] | ((k[17] | *(k + 9)
9   ctx->input[9] = k[20] | ((k[21] | *(k + 6)
10  ctx->input[10] = k[24] | ((k[25] | *(k +
11  ctx->input[11] = k[28] | ((k[29] | *(k +
12  ctx->input[12] = 0x0;
13  ctx->input[13] = *iv | ((iv[1] | *(iv + 1
14  ctx->input[14] = *iv[4] | ((iv[5] | *(iv +
15  ctx->input[15] = *iv[8] | ((iv[9] | *(iv +
16)

```

```

[FindCrypt3] - 0x7FFCD3491334: found sparse constants ChaChaInitStates_Salsa_exp32k for SALSa/Chacha
[FindCrypt3] - 0x7FFCD349EE59: found const array __newctype (used in VC CRT/UCRT Library), size = 384, elsize = 2
[FindCrypt3] - 0x7FFCD349F159: found const array __newclmap (used in VC CRT/UCRT Library), size = 384, elsize = 1

```

Source code C implementing Salsa and Chacha algorithms is abundant in Lib Crypto libraries, for example Cryptlib, libtomcrypt, libcrypto... But the C source we decompiled is more similar to the source here: <http://cr.yo.to/snuffle/ecrypt.c>. We are not 100% sure if the hacker changed the algorithm or because of the optimization mechanism of the VC compiler. Readers can refer to it for self comparison.

For decrypting the mutex name and SID, the hacker converts two hardcoded hex strings into a byte buffer using the Hex2Bytes function at address 0x7FFCD3492220, and then feeds this buffer to the Salsa/Chacha20 function at address 0x7FFCD34914F0.

```

.rdata:00007FFCD34A4230 g_szMutexHex db '2F001C6B09A02DC465DF3ACC970035DB9AF0EBD5DABE3F46DD426693EF230B7' ; DATA XREF: .data:g_pcMutexName:o
.rdata:00007FFCD34A4230 ; DATA XREF: .data:g_pcMutexName:o
.rdata:00007FFCD34A4230 db '21E77BCA7F903',0
.rdata:00007FFCD34A427F align 20h
.rdata:00007FFCD34A4280 g_szSIDHex db '3B494229EC48CD27',0 ; DATA XREF: .data:g_pcSID:o
.rdata:00007FFCD34A4291 align 8
.rdata:00007FFCD34A6B58 align 20h
.rdata:00007FFCD34A6B60 ; unsigned __int8 g_szIV0
.rdata:00007FFCD34A6B60 g_szIV0 db 'yRho04sZrtk',6 ; DATA XREF: Decode_SID_And_Mutex_Name+6D:o
.rdata:00007FFCD34A6B60 ; Decode_SID_And_Mutex_Name+DA:o
.rdata:00007FFCD34A6B6C align 10h
.rdata:00007FFCD34A6B70 ; unsigned __int8 g_szK0
.rdata:00007FFCD34A6B70 g_szK0 db 'abcdefghijklmnopqrstuvwxyz',0
.rdata:00007FFCD34A6B70 ; DATA XREF: Decode_SID_And_Mutex_Name+79:o
.rdata:00007FFCD34A6B70 ; Decode_SID_And_Mutex_Name+E6:o

```

After decrypt, we get:

1. SID = S-1-5-18
2. Mutex Name = Global\24yQoCWKY3kbZexjzTR6hc7pHU11I0EV

SID = "S-1-5-18" is known as Local System, Dll Services run under this account. Readers can refer here: [Well-known security identifiers in Windows operating systems](#). Hackers declared and used a struct like the one below to save the config that regulate the operation of this malware family. This struct has sizeof = 0x248 (584 decimal), and has been encrypted using the Salsa/Chach20 algorithm used above.

```

0000000 CConfig struc ; (sizeof=0x248, align=0x4, copyof=61)
0000000 ; XREF: Decrypt+13/r
0000000 ; Decrypt+29/r ...
0000000 dwSizeData dd ? ; XREF: MainThreadProc+4/r
0000004 dwHash dd ?
0000008 rgbIV db 12 dup(?)
0000014 fExecuteShellcode db ? ; XREF: MainThreadProc:loc_7FFCD34A6B60
0000015 fCreateMutex db ? ; XREF: MainThreadProc:loc_7FFCD34A6B60
0000016 fCheckSID db ? ; XREF: MainThreadProc:loc_7FFCD34A6B6C
0000017 fCheckExePath db ? ; XREF: MainThreadProc:loc_7FFCD34A6B70
0000018 wszExePath db 24 dup(?) ; XREF: MainThreadProc+59/r
0000030 dwShellcodeSize dd ? ; XREF: MainThreadProc+96/r
0000034 rgbShellcode db 532 dup(?) ; XREF: GetShellcodePath+3/r
0000034 ; MainThreadProc+BD/o
0000248 CConfig ends

```

The meaning of these fields in this struct:

- dwSizeData: The actual size of the real data area from the fExecuteShellcode field
- dwHash: ROL 0xB hash of the whole data range from rgbIv
- rgbIv: 12-byte array, used as value for parameter Iv for salsa_decrypt_bytes function
- fExecuteShellcode: flag specifies whether the data area in the rgbShellcode array has shellcode data, and whether the malware will execute this shellcode or not
- fCreateMutex: flag determines whether or not to create a mutex with the above decoded name
- fCheckSID: check if the malware is being executed correctly in the above decrypted SID group
- fCheckExePath: check whether the executing malware has the correct Exe name or the correct Parent Exe name with the szExePath field
- szExePath: Name of Exe or Parent Exe that needed to be checked
- dwShellcodeSize: The actual size of the rgbShellcode area (rgb = Range of Bytes) or length (in bytes) of the shellcode file's path
- rgbShellcode: Shellcode or path of another dll or shellcode that needs to be loaded and executed

Source decompiler of MainThreadProc, address = 0x7FFCD3491FD0.

```

1  int64 __fastcall MainThreadProc(LPVOID lpThreadParameter)
2  {
3      DWORD dwSize; // edi
4      void *pMem; // rax MAPDST
5      CConfig *pShellcode; // [rsp+38h] [rbp+10h] BYREF
6
7      if ( g_config.dwSizeData <= 0 || (g_config.dwSizeData + 0x14i64) > 0x248 )
8      {
9          return 0i64;
10     }
11     if ( IDecrypt(g_config.dwSizeData + 0x14, &g_config) )
12     {
13         return 0i64;
14     }
15     if ( g_config.fCheckSID && ICheckUserSID()
16         || g_config.fCheckExePath && ICheckExePath(g_config.wszExePath)
17         || g_config.fCreateMutex && ICreateMutex() )
18     {
19         return 0i64;
20     }
21     if ( g_config.fExecuteShellcode )
22     {
23         dwSize = g_config.dwShellcodeSize;
24         pMem = VirtualAlloc(0i64, g_config.dwShellcodeSize, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE);
25         if ( pMem )
26         {
27             memmove(pMem, g_config.rgbShellcode, dwSize);
28             (pMem)(0i64);
29             VirtualFree(pMem, 0i64, 0x0000u);
30         }
31     }
32     else if ( GetShellcodePath() )
33     {
34         pShellcode = 0i64;
35         if ( ReadShellcodeFile(&pShellcode) )
36         {
37             (&pShellcode->fExecuteShellcode)(0i64);
38         }
39     }
40     if ( !g_config.fCreateMutex )
41     {
42         return 0i64;
43     }
44     if ( g_hMutex )
45     {
46         CloseHandle(g_hMutex);
47     }
48     return 0i64;
49 }

```

Note: *g_config* is a global variable of the above struct *CConfig*. *0x14 (20)* is the total size of the 3 fields: *DWORD dwSizeData*, *DWORD dwHash* and *BYTE rgbIv[12]*.

After checking the correct size, the *Decrypt* function will decrypt the hardcoded config, encrypted with the decrypt Salsa/Chacha20 functions.

```
1 BOOL __fastcall Decrypt(int sizeOut, CConfig *pConfig)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     *sizeData = pConfig->dwSizeData;
6     if ( *sizeData + 0x14164 != sizeOut )
7     {
8         return 0;
9     }
10    pIv = pConfig->rgbIv;
11    hash = 0;
12    count = sizeData + 0xC;
13    pEnd = pIv;
14    if ( pIv )
15    {
16        if ( sizeData != 0xFFFFFFFF ) // != -0xC
17        {
18            do
19            {
20                aByte = *pEnd++;
21                hash = aByte + __ROL4__(hash, 0xB);
22                --count;
23            }
24            while ( count );
25        }
26        hash = __ROL4__(hash, 0xB);
27    }
28    if ( hash != pConfig->dwHash )
29    {
30        return 0;
31    }
32    strcpy(key, "u0FBSP2dDyTLhIQ9MXsEexmH7Jb1N3k");
33    salsa_decrypt_bytes(key, pEnd, pIv, &pConfig->fExecuteShellcode, sizeData);
34    return 1;
35 }
```

The value of the local hash variable in the *Decrypt* function is calculated from the address of *rgbIv*, the loop size is the value of the field *dwSizeData + 0xC* (12 = *sizeof(rgbIv)*). If the hash value matches the *dwHash* field, the data region will be decoded.

The decryption key is the hardcoded string *"u0FBSP2dDyTLhIQ9MXsEexmH7Jb1N3k"*, the *Iv* value is *rgbIv*, the output decoding starts at the address of the *fExecuteShellcode* field (offset *0x14*). After decoding the hardcoded config, in the *MainThreadProc* image above, the malware starts checking flags, flags that are set to 1 will call the corresponding check function.

The function checks the user's SID that the malware is running:

```

1 BOOL __stdcall CheckUserSID()
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     bRet = 0;
6     hToken = 0;
7     hProcess = GetCurrentProcess();
8     if ( !OpenProcessToken(hProcess, TOKEN_QUERY, &hToken) )
9     {
10        return 0;
11    }
12    LODWORD(ulReturn) = 0;
13    GetTokenInformation(hToken, TokenUser, 0, 0, &ulReturn);
14    if ( GetLastError() != ERROR_INSUFFICIENT_BUFFER )
15    {
16        return 0;
17    }
18    dwSize = ulReturn;
19    hHeap = GetProcessHeap();
20    pTokenUser = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, dwSize);
21    if ( !pTokenUser )
22    {
23        return bRet;
24    }
25    if ( GetTokenInformation(hToken, TokenUser, pTokenUser, ulReturn, &ulReturn) )
26    {
27        if ( IsValidSid(pTokenUser->User.Sid) )
28        {
29            pSid = pTokenUser->User.Sid;
30            pStrSid = 0;
31            if ( ConvertSidToStringSidA(pSid, &pStrSid) )
32            {
33                if ( !stricmp(pStrSid, g_pwszSID) )
34                {
35                    bRet = 1;
36                }
37                LocalFree(pStrSid);
38            }
39        }
40    }
41    hHeap = GetProcessHeap();
42    HeapFree(hHeap, 0, pTokenUser);
43    return bRet;
44 }

```

The variable g_pwszSID is WCHAR * type, decrypted from the beginning ("S-1-5-18"). If the SID is equal (strcmp return 0) then the function will return TRUE.

Function to check current Exe name or Parent Exe name (when malware runs as a service Dll). The function also returns TRUE when the Exe Name matches the szExePath field:

```

1 BOOL __fastcall CheckExePath(char *pszExeCheck)
2 {
3     const char *pFileName; // rax
4     CHAR szExePath[272]; // [rsp+20h] [rbp-128h] BYREF
5
6     memset(szExePath, 0, 0x105ui64);
7     GetModuleFileNameA(0, szExePath, 0x104u);
8     pFileName = PathFindFileNameA(szExePath);
9     return strcmp(pFileName, pszExeCheck) == 0;
10 }

```

Function creates mutex is the same as the regular CreateMutex functions:

```

1 BOOL __stdcall CreateMutex()
2 {
3     struct _SECURITY_ATTRIBUTES mutexAttr; // [rsp+20h] [rbp-48h] BYREF
4     SECURITY_DESCRIPTOR desc; // [rsp+38h] [rbp-30h] BYREF
5
6     mutexAttr.nLength = 0x18;
7     *&mutexAttr.bInheritHandle = 0i64;
8     InitializeSecurityDescriptor(&desc, 1u);
9     SetSecurityDescriptorDacl(&desc, 1, 0i64, 0);
10    mutexAttr.lpSecurityDescriptor = &desc;
11    g_hMutex = CreateMutexA(&mutexAttr, 1, g_pwszMutexName);
12    GetLastError();
13    if ( !g_hMutex )
14    {
15        return 0;
16    }
17    if ( !GetLastError() )
18    {
19        return 1;
20    }
21    CloseHandle(g_hMutex);
22    return 0;
23 }

```

The value of g_pwszMutexName variable has been decoded from the beginning: "Global\24yQoCWKY3kbZexjzTR6hc7pHU1IIOEV". The created Mutex will be saved to the g_hMutex global variable.

As shown in the figure of MainThreadProc, if the fExecuteShellcode field is set to 1, the shellcode will be executed as usual (VirtualAlloc, copy and execute). When it is 0, the shellcode file will be read from rgbShellcode.

```

1 BOOL __stdcall GetShellcodePath()
2 {
3     __int16 wszShellcodePath[264]; // [rsp+20h] [rbp-438h] BYREF
4     __int16 wszDllPath[264]; // [rsp+230h] [rbp-228h] BYREF
5
6     memset(wszShellcodePath, 0, 0x208ui64);
7     ExpandEnvironmentStringsW(g_config.rgbShellcode, wszShellcodePath, 0x104u);
8     if ( PathIsRelativeW(wszShellcodePath) )
9     {
10        memset(wszDllPath, 0, 0x208ui64);
11        GetModuleFileNameW(g_hInstDLL, wszDllPath, 0x104u);
12        PathRemoveFileSpecW(wszDllPath);
13        PathAddBackslashW(wszDllPath);
14        PathAppendW(wszDllPath, wszShellcodePath);
15        wcsncpy(wszShellcodePath, wszDllPath, 0x104ui64);
16    }
17    if ( !PathFileExistsW(wszShellcodePath) )
18    {
19        return 0;
20    }
21    wcsncpy(&g_config, wszShellcodePath, 260ui64);
22    return 1;
23 }

```

g_hInstDLL is the HINSTANCE of the malware, running as a service DLL, assigned value at the Entrypoint DllMain function. Readers notice three functions PathRemoveFileSpec, PathAddBackslash and PathAppend. These three functions will regenerate the path for the

Shellcode file that is from a subdirectory of the same level as the Malware. In this case, malware has an impersonation name of `C:\Windows\System32\C_20253.NLS`, that subfolder will also be located in `C:\Windows\System32`.

After getting the path of the shellcode file, the shellcode will be read, decrypted, and return a pointer to the decrypted shellcode.

```
1 BOOL __fastcall ReadShellcodeFile(void **ppShellcode)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     bRet = 0;
6     hFile = CreateFileW(@g_config, GENERIC_READ, FILE_SHARE_READ, 0i64, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0i64);
7     if ( !hFile )
8     {
9         return bRet;
10    }
11    dwSize = GetFileSize(hFile, 0i64);
12    if ( dwSize >= 0x14 )
13    {
14        pMem = VirtualAlloc(0i64, dwSize, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE);
15        if ( pMem )
16        {
17            dwTotalBytesRead = 0;
18            for ( dwBytesRead = 0; dwTotalBytesRead < dwSize; dwTotalBytesRead += dwBytesRead )
19            {
20                if ( !ReadFile(hFile, pMem + dwTotalBytesRead, dwSize - dwTotalBytesRead, &dwBytesRead, 0i64) )
21                {
22                    break;
23                }
24            }
25            if ( *pMem > 0 && Decrypt(dwSize, pMem) )
26            {
27                *ppShellcode = pMem;
28                bRet = 1;
29            }
30            else
31            {
32                VirtualFree(pMem, 0i64, 0x0000u);
33            }
34        }
35    }
36    CloseHandle(hFile);
37    return bRet;
38 }
```

The memory containing this decrypted shellcode is executed by `MainThreadProc`. With this sample, after the `Decrypt` function, we only have the following config information:

- All flags are 0
- File shellcode is "ErrorSvc.dll", field `dwShellcodeSize` = 0x1A (26)

II. Hunting the same loader

Based on the special hardcoded string "`u0FBSP2dDyTLhIQ9MXsEexmH7JbiN3k`", is used as IV for the Salsa/Chacha20 encrypt/decrypt function, we did a search on VirusTotal and Hybrid Analysis, there are many similar loaders, most of which are uploaded by users recently, from Vietnam, Korea, Japan, Hong Kong... and lastest is a sample from China.

- <https://www.virustotal.com/gui/search/content%253A%2522u0FBSP2dDyTLhIQ9MXsEexmH7JbiN3k%2522/files>
- <https://www.hybrid-analysis.com/string-search/results/08f2e828fe16c22515f0b8b7a5ccf9489ceeb58802ded94da4a3e13acd011e32>

Until yesterday, we have found and analyzed 7 more loader samples like this. The source code is completely the same, only the final build format are different (EXE or DLL). And it's all PE64, include the following samples:

1. 4578b3bf586658c47c8db1d497a8994d7637d28f16a11af9f6af64836085d4ed
 - Build Exe
 - Flags = 0
 - Shellcode path: stufte.dll
- 8061df4d29ea57a420491f0db4bf37964070cc695f4b1b45af40e46194cc8c36
 - Build Exe
 - Flags = 0
 - Shellcode path: tmp01.dat
- 4b1928dbaf68e427db2f3971ea2ff5604d210ef0dee876d57281d7e395da8c37
 - The impersonated file name is C_892.NLS
 - Build as Dll, original Dll name: DllSvchDtchX64.bin
 - Flags = 0
 - Shellcode path: winsec.dll
- d2beff6d7f5be68cdda36182d010e8103d86053fcc63f1166fec42727c26558d
 - Build as Dll, original Dll name: DllSvchDtchX64.bin
 - Flags = 0
 - Shellcode path: access.sys
- d28984576620aebfa929767ad9453fe7549c969716d41ba49cbe6ca7fae72789
 - Build as Exe
 - Flag fExcuteShellcode = 1
 - Shellcode size = 0x107A2 (67490)
- 3714568d8c8b7359259e968664de3a6c13d6d7c16559dfb0a25f9aa8194e8de4
 - Build as Dll, original Dll name: DllHijkDtchX64.bin
 - fCreateMutex, fCheckSID, fCheckExePath set 1
 - Exe Parent Name to check: WmiApSrv.exe
 - Shellcode path: AxLnst.bin

Notice the file name Exe Parent. This exe is the original Windows file, located in the \Windows\System32\Wbem folder. So this malware and AxLnst.bin will also be in this directory (according to the GetShellcodePath function). This could be an attack using WMI Exploitation that the Winnti/Derusbj Group has been using. Read more [here](#).
- b69d9ed06cba8eea081df01bad146abb004a4cf5fb6b296017d82ebb18975386
 - Build as Exe
 - Flags = 0
 - Shellcode path: koreanflass.bin

III. Hunting the updated malware of this group

Continuing to hunt for signs of old malware samples that this group has used in campaigns targeted Vietnam over the years, we found that this group still uses old samples, has updated code and rebuilt with Visual Studio 2019, v16.4 or later. Completely build in x64 mode.

This group continues to use files that impersonate Windows' NLS files as config containers or as shellcode files. The identification point is that this group has a coder who specializes in CryptoPP library and coding in C++ style, using std::string.

The samples we collected were released recently, in May and June, also from the countries mentioned above. We collected and analyzed the following samples:

1. 5afc41060cf62d1613219caa108eb9714074479a413f4a26797c0358fc95a4db

- Built with Visual Studio 2019 v16.9
 - PDB Path: C:\Users\VS\Desktop\Auto_Firefox\x64\Release\8.1.pdb
 - Using CryptoPP, C++ style
 - Xor value: 0x28
 - Build time: 08/06/2021 - 1:24:48 AM (UTC)
 - Export function: ServiceMain, run as a service Dll
 - Read and execute MSIscl.Dll in the same directory and load vsmapi.dll in SysWow64, calling the netEntryApi export function.
-
- 8dd13f34d1734d3c844474ce98a4f39244e511bafbafd59b18bb7fb0b52ce895
 - Built with Visual Studio 2019 v16.9
 - PDB Path:
C:\Users\Machine\Desktop\Work\20200913\Auto_Firefox\x64\Release\8.pdb
 - Using CryptoPP, C++ style
 - Build time: 19/09/2020 - 8:58:34 AM (UTC)
 - Export function: NetworkChecker

- Decrypt, read two configs from two fake NLS files, C_4868.NLS and C_4869.NLS

```

dq 0Ah
dq 0Fh
szC4868NLS db 'C_4868.NLS', 0
db 0
db 83h ; f
db 0
db 0
db 0
db 0
std::string::_Mysize -> dq 0Ah
dq 0Fh <- std::string::_Myres
szC4869NLS db 'C_4869.NLS', 0
db 0
db 83h ; f
db 0
db 0
db 0

```

- [9abf047566c6e9bd77120e8eb6c3503eef7c05dd4fd0abac9046d495291e5c8d](#)
 - Built with Visual Studio 2008, code C style
 - Export two functions Run and main. Two different functions but the code is exactly the same
 - PDB path: C:\Dev\16\3\x64\Release\F71.pdb
 - Build time: 01/06/2016 - 4:38:32 PM (UTC)
 - Impersonate as Windows VfWDM.dll in Resource Version Info
 - C2 hardcoded, xor with 0x27, is "www.newshcm.com"
 - Read two files is NLS fake are C_436.NLS and C_20130.NLS. The xor value to decode the contents of 2 files is 0x26 and 0x27
- [60fe689bafb1ce4def3fab1c91e69e46b223869314e4364fa8efb12e6a0bafba](#)
 - Built with Visual Studio 2019 v16.9, C style
 - PDB path: C:\Users\VS\Desktop\Auto_Firefox\x64\Release\8.1.pdb
 - Export function: ServiceMain
 - Xor value: 0x2B, load dll pubiapi.dll in Windows\SysWow64, calling export function netEntryApi of this Dll.
- [68e871190f405131635ccaa851339c9ca3f61c3b6a9d84dbd7afc99b65edd588](#)
 - Built with Visual Studio 2019 v16.9
 - Using CryptoPP, C++ style
 - Build time: 12/04/2021 - 9:18:26 PM (UTC)
 - Export function: netEntryApi
 - Load 2 fake NLS files are C_4868.NLS and C_4869.NLS like (2)
- [918ad6c918b26de1e112281393f6ced9141712484bb0da5f8250fb36fc0d476b](#)
 - Built with Visual Studio 2012, C style

- PDB Path: C:\Dev\17D\Release\7.pdb
- Build time: 30/04/2017 - 12:29:05 AM (UTC)
- Export two functions are Run and main like (3)
- CC hardcoded, xor with 0x1B, is “www.sexphm.com” and IP hardcoded 172.16.22.22
- Read two fake NLS files are C_20831.NLS and C_20832.NLS in Windows\System32
- c092546e9db9424d454cc21047d847ad93424440e7a4d339fe58fa9a4d8f6913
 - Is vsmapi.dll of (1)
 - Built with Visual Studio 2019 v16.9
 - Using CryptoPP, C++ style
 - PDB path: C:\Users\VS\Desktop\Auto_Firefox\x64\Release\8.pdb
 - Build time: 08/06/2021 - 1:24:51 AM (UTC)
 - Export function: netEntryApi
 - Load two fake NLS files are C_4868.NLS and C_4869.NLS like (2) and (5)

Thus, we can see that the samples that this group used in this campaign are mostly rebuilt, besides some old samples in their inventory that have not been detected. Maybe they have been used, installed and infected in many companies and organizations of many countries, including Vietnam since 2016. Until now, we have discovered a number of fake NLS files that this group used throughout, including:

- C_201263a.NLS
- C_20130.NLS
- C_20253.NLS
- C_20831.NLS
- C_20832.NLS
- C_20834.NLS
- C_20835.NLS
- C_21871.NLS
- C_21872.NLS
- C_436.NLS
- C_4868.NLS
- C_4869.NLS
- C_877.NLS
- C_878.NLS
- C_892.NLS

And probably many more impersonated NLS files out there that we may not discovered.

IV. Analysing Windows C_XXXX.NLS files

The value xxxx is a number, which is a codepage identifier. For example, Vietnam has a codepage of 1258, the file C_1258.nls on Windows is for Vietnam.

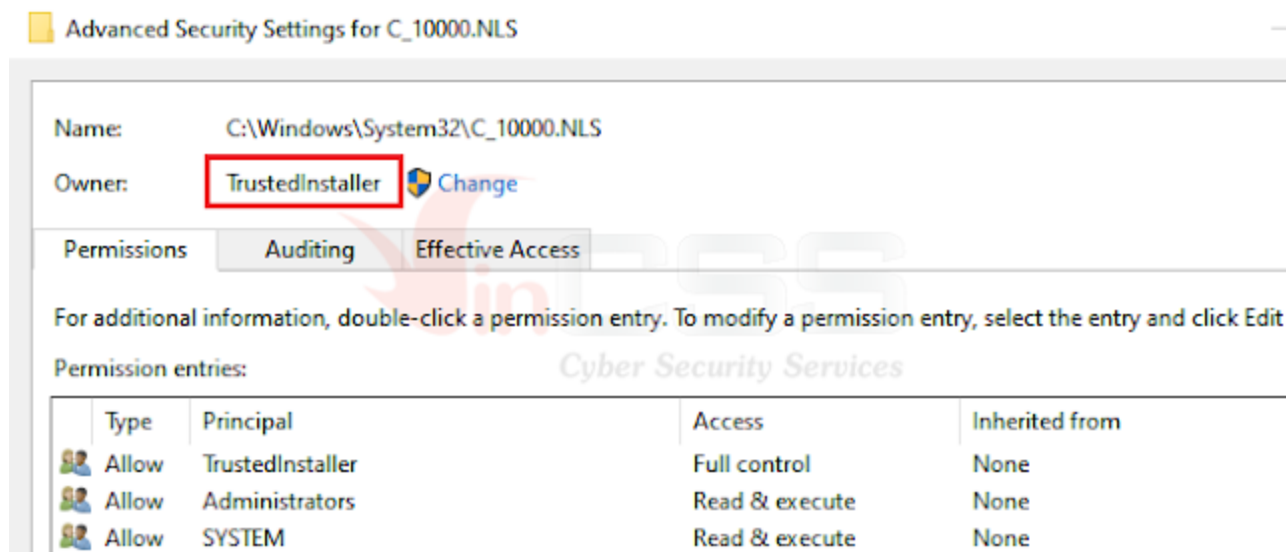
For more codepage definition, readers can read it [here](#). Values of Codepages that international and Windows have a convention can refer [here](#). The current maximum value of Codepage is 65501, Unicode UTF-8. Between 1 and 65535 (0xFFFF), you will see a lot of space, more than 65,000 numbers. This group used numbers not on the above Codepage identifiers to name the impersonated C_XXXX.NLS files.

The original Windows C_XXXX.NLS files are used for mapping and converting from MultiByte to Unicode characters. Two common API functions commonly used in Windows, MultiByteToWideChar and WideCharToMultiByte, are based on these C_XXXX.nls files corresponding to the current Windows Codepage on the user's machine.

On Windows 2000, XP operating systems, these .nls files are not included in the list of Windows Protection Files files, only .exe, .dll, .sys, .ocx files. From Windows Vista onwards, the list of Windows Protection Files file types is expanded and the .nls file is added. Readers can refer to protected files [here](#).

The C_XXX.nls are installed when the user installs Windows, located in the Windows\System32 and Windows\WinSXS\ folders in several subfolders named xxx.codepage-core.xxx and xxx.codepage-additional-xxx.

These C_XXXX.nls files all have Owner *Trust Installer*, users with *System* and *Administrators* rights can only Read, no change rights. When trying to switch Owner and change these files, Windows Resource Protection will notify and recover immediately.



When the user installs Windows, the list of C_XXXX.nls files were created by Windows is located at KEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage. The name value is the codepage number and the Data value is the codepage file name.

Name	Type	Data
ab 869	REG_SZ	c_869.nls
ab 866	REG_SZ	c_866.nls
ab 865	REG_SZ	c_865.nls
ab 864	REG_SZ	c_864.nls
ab 863	REG_SZ	c_863.nls
ab 862	REG_SZ	c_862.nls
ab 861	REG_SZ	c_861.nls
ab 860	REG_SZ	c_860.nls
ab 858	REG_SZ	c_858.nls
ab 857	REG_SZ	c_857.nls
ab 855	REG_SZ	c_855.nls
ab 852	REG_SZ	c_852.nls
ab 850	REG_SZ	c_850.nls
ab 775	REG_SZ	c_775.nls
ab 737	REG_SZ	c_737.nls
ab 720	REG_SZ	c_720.nls
ab 708	REG_SZ	c_708.nls
ab 57011	REG_SZ	c_jscii.dll
ab 57010	REG_SZ	c_jscii.dll
ab 57009	REG_SZ	c_jscii.dll
ab 57008	REG_SZ	c_jscii.dll

The image above is part of the list of Windows 10. In addition to .nls files, Windows also uses C_XXXX.dll files to serve for mapping, converting between that codepage back and forth Unicode. These dlls only export a single function is NlsDllCodePageTranslation. Prototype of this function: `DWORD __stdcall NlsDllCodePageTranslation(DWORD CodePage, DWORD dwFlags, LPSTR lpMultiByteStr, int cchMultiByte, LPWSTR lpWideCharStr, int cchWideChar, LPCPINF lpCPInfo)`

On Windows XP and 2000, the number of codepages in the above registry is less, although there are many .nls files copied by the Windows installer to System32\Dllcache, they are not considered to have been installed and updated in the above registry.

Name	Type	Data
ab(Default)	REG_SZ	(value not set)
ab10000	REG_SZ	c_10000.nls
ab10001	REG_SZ	
ab10002	REG_SZ	
ab10003	REG_SZ	
ab10004	REG_SZ	
ab10005	REG_SZ	
ab10006	REG_SZ	c_10006.nls
ab10007	REG_SZ	c_10007.nls
ab10008	REG_SZ	
ab10010	REG_SZ	c_10010.nls
ab10017	REG_SZ	c_10017.nls
ab10021	REG_SZ	
ab10029	REG_SZ	c_10029.nls
ab10079	REG_SZ	c_10079.nls
ab10081	REG_SZ	c_10081.nls
ab10082	REG_SZ	c_10082.nls
ab1026	REG_SZ	c_1026.nls
ab1047	REG_SZ	
ab1140	REG_SZ	
ab1141	REG_SZ	
ab1142	REG_SZ	
ab1143	REG_SZ	
ab1144	REG_SZ	
ab1145	REG_SZ	
ab1146	REG_SZ	
ab1147	REG_SZ	
ab1148	REG_SZ	
ab1149	REG_SZ	

The number of codepages that are considered installed and supported on each version of Windows is also different. From Windows Vista onwards, all .nls files that are installed are turned on as installed. Codepage installed and supported on Windows XP:

```

Number of CodePage supported: 134
Number of CodePage installed: 55

Supported CodePages = [ 37 437 500 708 720 737 775 850 852 855 857 858 860 861 8
62 863 864 865 866 869 870 874 875 932 936 949 950 1026 1047 1140 1141 1142 1143
1144 1145 1146 1147 1148 1149 1250 1251 1252 1253 1254 1255 1256 1257 1258 1361
10000 10001 10002 10003 10004 10005 10006 10007 10008 10010 10017 10021 10029 1
0079 10081 10082 20000 20001 20002 20003 20004 20005 20105 20106 20107 20108 201
27 20261 20269 20273 20277 20278 20280 20284 20285 20290 20297 20420 20423 20424
20833 20838 20866 20871 20880 20905 20924 20932 20936 20949 21025 21027 21866 2
8591 28592 28593 28594 28595 28596 28597 28598 28599 28603 28605 38598 50220 502
21 50222 50225 50227 50229 51949 52936 57002 57003 57004 57005 57006 57007 57008
57009 57010 57011 65000 65001 ]

Installed CodePages = [ 37 437 500 737 775 850 852 855 857 860 861 863 865 866 8
69 874 875 932 936 949 950 1026 1250 1251 1252 1253 1254 1255 1256 1257 1258 136
1 10000 10006 10007 10010 10017 10029 10079 10081 10082 20127 20261 20866 21866
28591 28592 28594 28595 28597 28599 28603 28605 65000 65001 ]

```

Codepage installed and supported on Windows 7


```
Number of CodePage supported: 135
Number of CodePage installed: 135

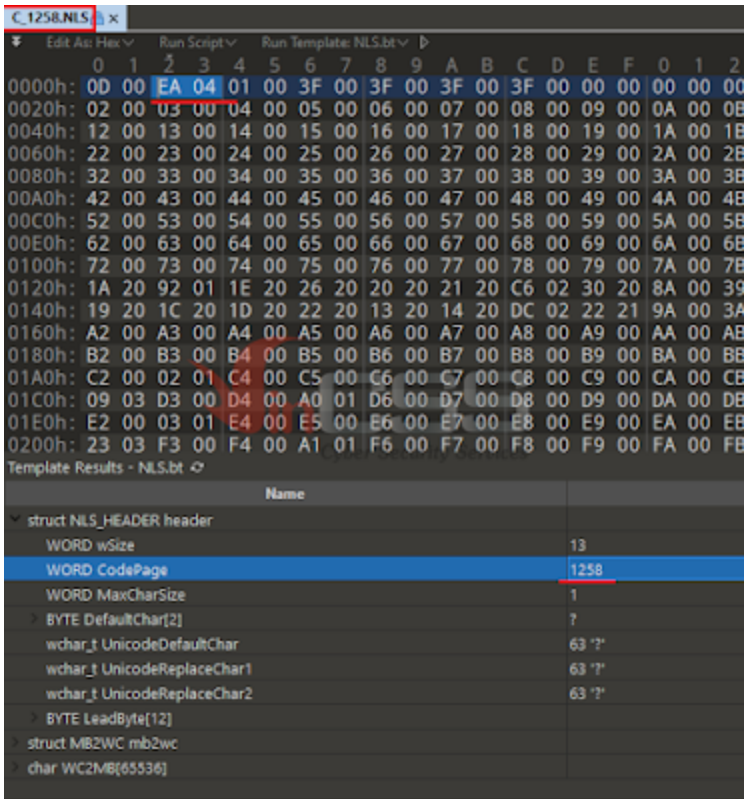
All CodePages = [ 37 437 500 708 720 737 775 850 852 855 857 858 860 861 862 863 864
865 866 869 870 874 875 932 936 949 950 1026 1047 1140 1141 1142 1143 1144 1145 1146
1147 1148 1149 1250 1251 1252 1253 1254 1255 1256 1257 1258 1361 10000 10001 10002 10
003 10004 10005 10006 10007 10008 10010 10017 10021 10029 10079 10081 10082 20000 200
01 20002 20003 20004 20005 20105 20106 20107 20108 20127 20261 20269 20273 20277 2027
8 20280 20284 20285 20290 20297 20420 20423 20424 20833 20838 20866 20871 20880 20905
20924 20932 20936 20949 21025 21027 21866 28591 28592 28593 28594 28595 28596 28597
28598 28599 28603 28605 38598 50220 50221 50222 50225 50227 50229 51949 52936 54936 5
7002 57003 57004 57005 57006 57007 57008 57009 57010 57011 65000 65001 ]
```

Codepage installed and supported on Windows 10

```
Number of CodePage supported: 140
Number of CodePage installed: 140

All CodePages = [ 37 437 500 708 720 737 775 850 852 855 857 858 860 861 862 863 864 86
5 866 869 870 874 875 932 936 949 950 1026 1047 1140 1141 1142 1143 1144 1145 1146 1147
1148 1149 1250 1251 1252 1253 1254 1255 1256 1257 1258 1361 10000 10001 10002 10003 10
004 10005 10006 10007 10008 10010 10017 10021 10029 10079 10081 10082 20000 20001 20002
20003 20004 20005 20105 20106 20107 20108 20127 20261 20269 20273 20277 20278 20280 20
284 20285 20290 20297 20420 20423 20424 20833 20838 20866 20871 20880 20905 20924 20932
20936 20949 21025 21027 21866 28591 28592 28593 28594 28595 28596 28597 28598 28599 28
603 28605 38598 50220 50221 50222 50225 50227 50229 51949 52936 54936 55000 55001 55002
55003 55004 57002 57003 57004 57005 57006 57007 57008 57009 57010 57011 65000 65001 ]
```

Since Microsoft does not disclose the structure of the .nls file, we searched the Internet and relied on the WinNLS.h file in the Windows SDK to create the NLS.bt file. This file is used as a template parser for **010 Editor**, a HexEditor program that supports scripts and parse templates very strongly, widely used by the RE community, forensics When using 010Editor to open an NLS file and select the template file as NLS.bt, 010Editor will show us the internal structure of an NLS file. We uploaded NLS.bt [here](#).



V. Tools to check the number of codepages and scan fake NLS files file

After analyzing the structure of an official, original Windows C_XXXX.NLS file, VinCSS has developed two tools to check and scan fake NLS files of this group. These two tools are written in Delphi (Object Pascal) and built with [Free Embarcadero Delphi Community Edition](#).

We provide both the built exe and the source code for your reference, which can be easily tested, rebuilt by yourself, also in the [above repo](#). According to [this report](#) by Positive Technologies, hackers have now updated to new fake NLS.

1. CheckCP:

Based on the EnumSystemCodePages API function with two parameters CP_INSTALLED and CP_SUPPORTED, CheckCP will display a list of installed and supported codepages on the current Windows. If you detect a suspicious C_XXXX.NLS file, you can enter that number into the CheckCP program to check if the codepage number is fake or belong to Windows. XXXX is a number, for example: file C_20130.NSL, the number to check is 20130.

With the list of fake NLS files above, we immediately see that all the codepage numbers are fake (1258 and 1252 are valid, we added):

```
Enter the codepage number to check: 201263
201263 is not a valid CodePage
Enter the codepage number to check: 20130
20130 is not a valid CodePage
Enter the codepage number to check: 20253
20253 is not a valid CodePage
Enter the codepage number to check: 20831
20831 is not a valid CodePage
Enter the codepage number to check: 20832
20832 is not a valid CodePage
Enter the codepage number to check: 20834
20834 is not a valid CodePage
Enter the codepage number to check: 20835
20835 is not a valid CodePage
Enter the codepage number to check: 21871
21871 is not a valid CodePage
Enter the codepage number to check: 21872
21872 is not a valid CodePage
Enter the codepage number to check: 436
436 is not a valid CodePage
Enter the codepage number to check: 4868
4868 is not a valid CodePage
Enter the codepage number to check: 4869
4869 is not a valid CodePage
Enter the codepage number to check: 877
877 is not a valid CodePage
Enter the codepage number to check: 878
878 is not a valid CodePage
Enter the codepage number to check: 892
892 is not a valid CodePage
Enter the codepage number to check: 1258
1258 is in installed CodePages  Việt Nam
1258 is in supported CodePages
Enter the codepage number to check: 1252
1252 is in installed CodePages  Windows
1252 is in supported CodePages
```

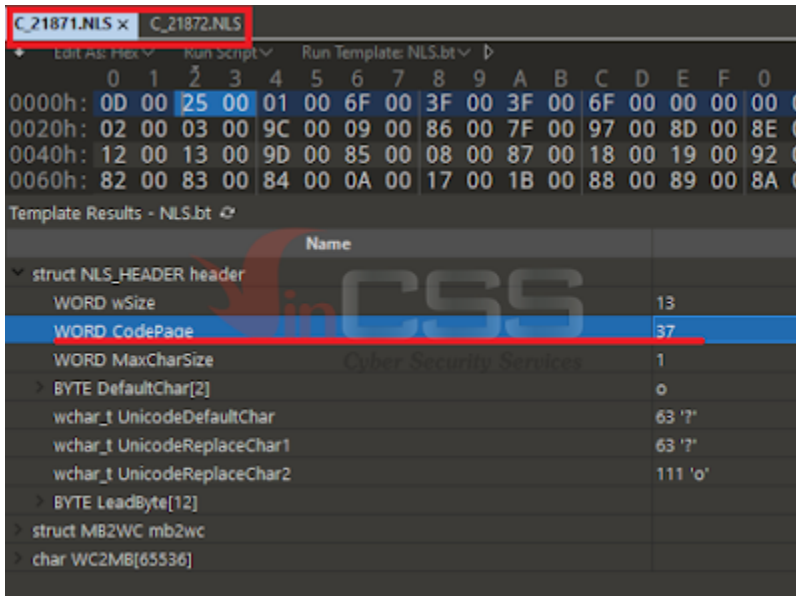
CheckCP.exe source code you download at the above repo, in the subdirectory .\src, file CheckCP.dpr

2. NLSScan.exe:

NLSScan is the main program to scan all C_XXXX.NLS files in Windows\SysWow64 and Windows\System32 folders, deep into all subfolders. This file is built with 32bit mode, running on old Windows such as XP, 2000 because there is a high possibility that many computers in organizations still use these operating systems.

This group always put fake NLS files in the above two folders. NLSScan checks many factors to ensure that a C_XXXX.NLS file must meet all of those conditions to not be considered fake or malware. When running NLSScan with no parameters, NLSScan will request Admin privileges to read only a small portion of the files C_XXXX.NLS finds. If you choose Yes, NLSScan will automatically run again in Admin privilege.

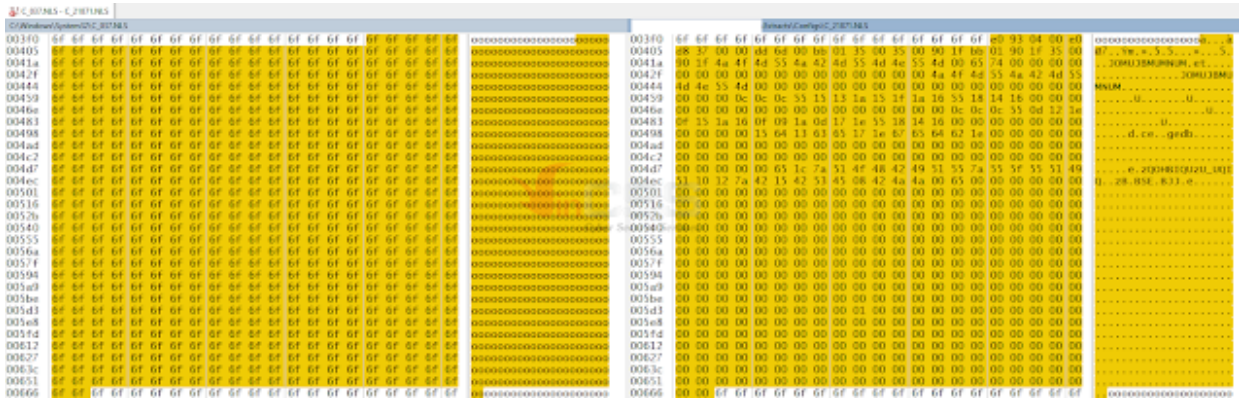
For example, a requirement that the codepage number be consistent in the file name and in the file content. We found two cases where the content of the .NLS file was valid when parsed in NLS.bt, but the codepage number in the file name was invalid, and the codepage number in the content was inconsistent. Hackers copied the original Windows file C_037.NLS into two new files C_21871.NLS and C_21872.NLS, overwriting the config of the content of the file.



```
0000h: 0D 00 25 00 01 00 6F 00 3F 00 3F 00 6F 00 00 00 00 00
0020h: 02 00 03 00 9C 00 09 00 86 00 7F 00 97 00 8D 00 8E 00
0040h: 12 00 13 00 9D 00 85 00 08 00 87 00 18 00 19 00 92 00
0060h: 82 00 83 00 84 00 0A 00 17 00 1B 00 88 00 89 00 8A 00
```

Name	
struct NLS_HEADER header	
WORD wSize	13
WORD CodePage	37
WORD MaxCharSize	1
BYTE DefaultChar[2]	0
wchar_t UnicodeDefaultChar	63 '?'
wchar_t UnicodeReplaceChar1	63 '?'
wchar_t UnicodeReplaceChar2	111 'o'
BYTE LeadByte[12]	
struct MB2WC mb2wc	
char WC2MB[65536]	

Compare these two files with the C_037.NLS file and you will see the overwritten area:



When NLSScan detects fake NLS files, the tool will ask the user for permission to copy those files to the %TEMP% folder and delete them. If the tool fails to remove it, NLSScan will prompt you to reboot to delete it at the next reboot. When you receive the NLSScan message as above, there is a fake NLS file, you should allow copying and deleting, then reboot Windows immediately, run the tool again. If the tool still cannot be deleted, please restart Windows in Safe Mode, run the tool again or find and delete those fake NLS files manually.

When NLSScan has detected a fake NLS file, it is almost certain that your computer has been infected with some malware of this group. You should disconnect from the Internet, rescan your system with AV programs, change the passwords, review all security factors.

You can send us the fake NLS files that have been copied to the %Temp% folder, and if you need helping with finding, review, etc. please contact us at the email address: malware.report@vincss.net

```
NLSScan - Scan and detect malwares, that fake Windows C_*.nls files
Written by HTC (TQN) - VinCSS(a member of Vingroup)
Version 1.0 - First release

Usage: NLSScan [NLSFileName1] ...[NLSFileNameN]
If run withouth parameter, scan all C_*.nls files in Windows System directories

Computer info: Windows 10 (Version 10.0, Build 19041, 64-bit Edition)
Get C_*.nls files list in C:\WINDOWS\system32\ and C:\WINDOWS\SysWOW64\ directories...done.
Warning: NLS files cannot be placed in C:\WINDOWS\SysWOW64\ directory. They could be malware !!!
Scanning 116 files...
Log file at D:\Git\VinCSS-RE-Tools-Utilities\NLSScan\NLSScan.log
Total files OK: 113
Total files bad: 3
File C:\WINDOWS\SysWOW64\0409\C_877.NLS
    Filename codepage is invalid
    NLS header size is not equal 0xD
File C:\WINDOWS\SysWOW64\0409\C_877.NLS copied to C:\Temp\C_877.NLS
File C:\WINDOWS\SysWOW64\0409\C_877.NLS deleted
File C:\WINDOWS\SysWOW64\0409\C_878.NLS
    Filename codepage is invalid
    NLS header size is not equal 0xD
File C:\WINDOWS\SysWOW64\0409\C_878.NLS copied to C:\Temp\C_878.NLS
File C:\WINDOWS\SysWOW64\0409\C_878.NLS deleted
File C:\WINDOWS\system32\C_892.NLS
    Filename codepage is invalid
    File is a PE (Windows) Executable
    File is a Windows DLL file
    File is PE 64bit
File C:\WINDOWS\system32\C_892.NLS copied to C:\Temp\C_892.NLS
File C:\WINDOWS\system32\C_892.NLS deleted
```

This is a test result by using NLSScan on our Windows 10 machine. Especially, you can see the file C_878.NLS that we mentioned in part II, which is a PE x64 dll file, which Windows Defender has not detected at the time of this article.

NLSScan can also scan each or multiple NLS files (user can enter the path of those NLS files). Currently, NLSScan only supports scanning C_XXXX.NLS files. On Windows there are a number of other NLS files such as: I_intl.nls, locale.nls, normidna.nls, normnfc.nls, normnfd.nls, normnffc.nls, normnffd.nls, SortXXX.nls. But because the format of these files is not announced by Microsoft, we cannot check. When you encounter files with such names, you use the Windows tool **sfc.exe** (System File Checker).

```
C:\Sandbox\New\NLS\C_20253.NLS>sfc /scanfile=C:\Sandbox\New\NLS\C_20253.NLS\C_20253.NLS
Windows Resource Protection could not perform the requested operation.
C:\Sandbox\New\NLS\C_20253.NLS>sfc /scanfile=C:\Windows\System32\C_037.NLS
Windows Resource Protection did not find any integrity violations.
```

The file C_20253.NLS is invalid, so sfc will say “**WRP could not perform....**”, C_037.NLS is a valid file, located in Windows Protection Files, not compromised, so sfc says “**WRP did not find ...**”

We also upload a small bat file **sfcnlis.cmd** in the above repo for you to periodically run and check with NLSscan all .nlis files in the Windows directory. We hope you will share these tools to scan all Windows-based computers in Vietnamese companies, agencies, organizations and economic groups. In our opinion, this group is very dangerous, may have been able to penetrate and lie deep inside with undetected for a long time, causing great harm to Vietnam.

We wish you good health, peace, ... and together overcome this pandemic.

Click here for [Vietnamese version](#).

Sincerely,

Author: Truong Quoc Ngan (HTC), Dang Dinh Phuong

VinCSS (a member of Vingroup)