Darkside is a ransomware notorious for attacking high profile industrial control systems and facilities. It's most famous attack was against the Colonial Pipeline line in the United States between May 6 – May 12 of 2021. This attack show cased how dangerous ransomware can be and what an effect these types of attacks can have on the country and world. This is an analysis on Darkside.

Analysis

Darkside utilizes a lot of dynamic allocation as means to hide its functionality. This is done with the usage of LoadLibrary and GetProcessAddress functions. What makes this so difficult is their usage of encrypted strings which have to go through an decryption process in order to be used first. This added to the complexity and could not be statically resolved. To determine what these strings were dynamic analysis was necessary.

```
int __stdcall CreateDecryptionKey(int a1, int a2, unsigned int a3)
{
  signed int v3; // ecx@1
  int v4; // edx@1
  int v5; // ebx@1
  int v6; // edi@1
  int result; // eax@1
  int v8; // edx@3
  int v9; // ecx@3
  unsigned int v10; // ebx@3

  v3 = 240;
  v4 = *(_DWORD *)a1;
  v5 = *(_DWORD *)(a1 + 4);
  v6 = *(_DWORD *)(a1 + 8);
  result = *(_DWORD *)(a1 + 12);
  do
  {
    *(_DWORD *)&decryption_key[v3 + 12] = v4;
    *(_DWORD *)&decryption_key[v3 + 8] = result;
    *(_DWORD *)&decryption_key[v3 + 4] = v5;
    *(_DWORD *)&decryption_key[v3] = v6;
    v4 -= 269488144;
    result -= 269488144;
    v5 -= 269488144;
    v6 -= 269488144;
```

Figure 1

The decryption process is done entire through the use of the key created through this function, taking in two smaller 16 length keys it creates a 256 byte decryption key which is utilized through the entire malware to dynamically allocate and resolve strings and API. Because of the constant value of the key, the key is a great host base indicator as long as new versions do not change these.

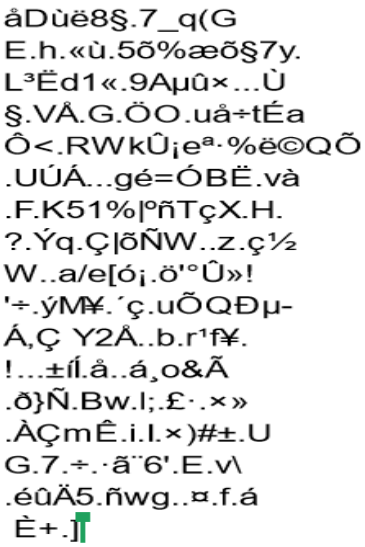

Figure 2

```
decryption_buffer_process_wrapper((int)LibFileName, *(_DWORD *)&LibFileName[-4]);
v0 = LoadLibraryA(LibFileName);
clear_string((_DWORD *)LibFileName, *(_DWORD *)&LibFileName[-4]);
v1 = &LibFileName[*(_DWORD *)&LibFileName[-4]];
function_resolution(v0, (FARPROC *)&wcsicmp, v1);
v2 = *(_DWORD *)v1;
v1 += 4;
decryption_buffer_process_wrapper((int)v1, v2);
v3 = LoadLibraryA(v1);
clear_string(v1, *((_DWORD *)v1 - 1));
v4 = &v1[*((_DWORD *)v1 - 1)];
function_resolution(v3, (FARPROC *)&wcsicmp, v4);
v5 = *(_DWORD *)v4;
v4 += 4;
decryption_buffer_process_wrapper((int)v4, v5);
v6 = LoadLibraryA(v4);
clear_string(v4, *((_DWORD *)v4 - 1));
v7 = &v4[*((_DWORD *)v4 - 1)];
function_resolution(v6, (FARPROC *)&wcsicmp, v7);
v8 = *(_DWORD *)v7;
v7 += 4;
decryption_buffer_process_wrapper((int)v7, v8);
v9 = LoadLibraryA(v7);
clear_string(v7, *((_DWORD *)v7 - 1));
v10 = &v7[*((_DWORD *)v7 - 1)];
```

Figure 3

The API's are resolved here, looping through dword values and setting them accordingly. Afterwards the dword values are as followed:

```
.data:00410D3E ; int (__stdcall *sprintf_ptr)(_DWORD, _DWORD, _DWORD, _DWORD)
.data:00410D3E sprintf_ptr     dd ?                    ; DATA XREF: sub_407EB9+7D↑r
.data:00410D42 ; int (__stdcall *RtlGetVersion_ptr)(_DWORD, _DWORD, _DWORD, _DWORD)
.data:00410D42 RtlGetVersion_ptr dd ?                  ; DATA XREF: sub_404C32+22↑r
.data:00410D42                                         ; sub_404C32+3A↑r
.data:00410D46                 db    ? ;
.data:00410D47                 db    ? ;
.data:00410D48                 db    ? ;
.data:00410D49                 db    ? ;
.data:00410D4A ; int (__stdcall *RtlWow64EnableFsRedirectionEx_ptr)(_DWORD)
.data:00410D4A RtlWow64EnableFsRedirectionEx_ptr dd ?  ; DATA XREF: sub_40380C+B0↑r
.data:00410D4E ; int (__stdcall *NtAllocateVirtualMemory_ptr)(_DWORD)
.data:00410D4E NtAllocateVirtualMemory_ptr dd ?        ; DATA XREF: sub_40380C+E0↑r
.data:00410D52                 db    ? ;
.data:00410D53                 db    ? ;
.data:00410D54                 db    ? ;
.data:00410D55                 db    ? ;
.data:00410D56 ; int (__stdcall *NtProtectVirtualMemory_ptr)(_DWORD, _DWORD, _DWORD)
.data:00410D56 NtProtectVirtualMemory_ptr dd ?         ; DATA XREF: sub_401AEC+18↑r
.data:00410D56                                         ; sub_401E9E+26↑r ...
.data:00410D5A ; int (__stdcall *NtSetInformationThread_ptr)(_DWORD, _DWORD, _DWORD,
```

Figure 4

What really made Darkside ransomware so interesting was its usage of a UAC bypass to encrypt to begin with. A UAC stand for User Access Control, and bypassing it gives a program more control and privileges. In this case Darkside utilized a COM object to achieve these higher privileges. Note the following.

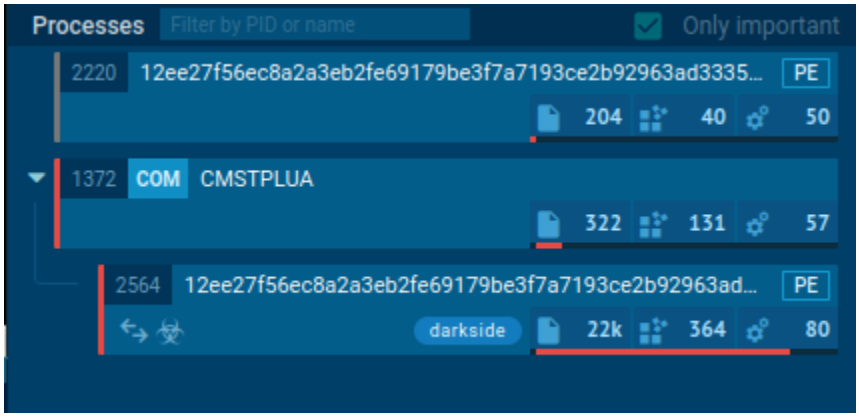

Figure 5

Initially Darkside is execute but almost immediately it executes the COM object CMSTPLUA. This being generated in a very specific way; C:\Windows\system32\DllHost.exe /Processid:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}. Was is important about this COM object is that it is given higher privileges because it is listed under trusted COM by Windows. It then executes a ShellExecute function on Darkside, running it now in higher privileges.

List of Approved COM is listed in the registry:

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\UAC\COMAutoApprovalList

For now, this is all my analysis as taken from this piece of malware. This malware is interesting because of its ability to escalate its privilege and with relative ease and less noise than other means. Like most security exploits, utilizing the trust places in other resources was the downfall here as Darkside was able to capitalize on this and proceed with the encryption process and steal money accordingly.