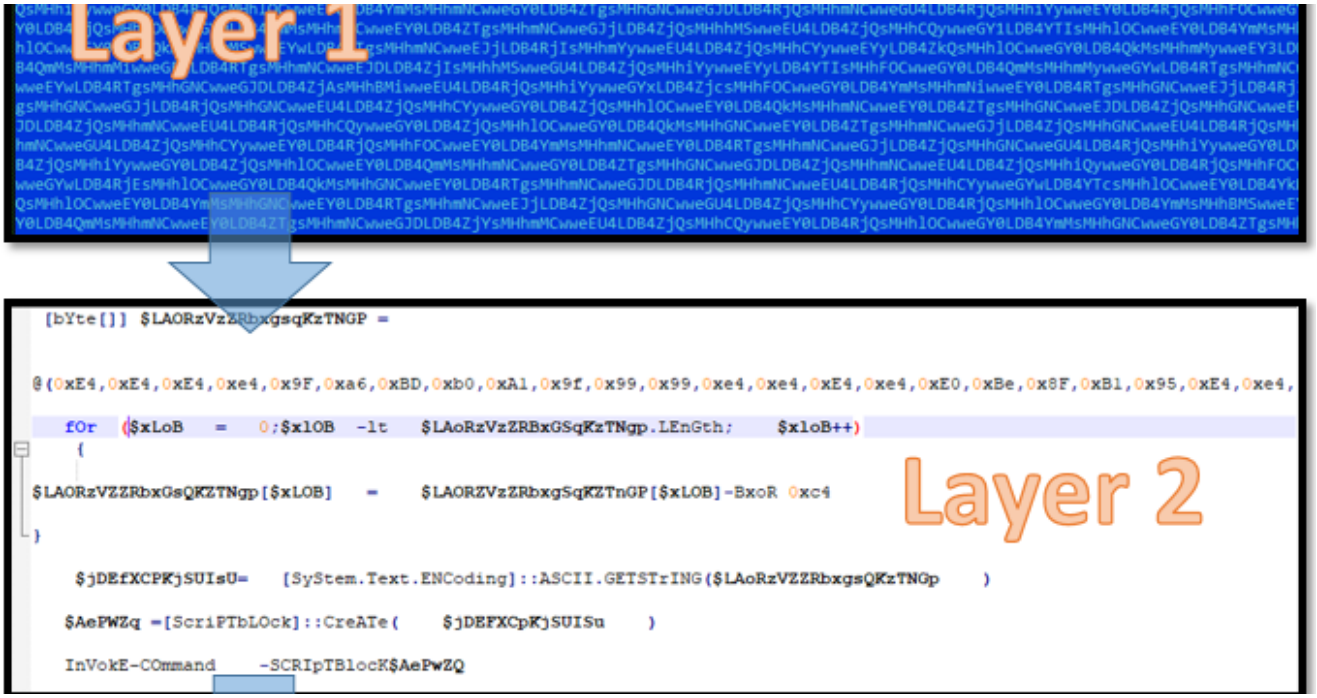
Netwalker Ransomware: [API Call Obfuscation (using Structure) and Evading Memory Forensic]

tccontre.blogspot.com/2020/05/netwalker-ransomware-api-call.html



Today I just want to share some interesting obfuscation and anti memory forensic techniques I've learned from Netwalker Ransomware that makes its code more time consuming and hard to analyze. This also include the first part which is a obfuscated powershell that will serve as the loader of the malware.

Stage 1: Obfuscated Powershell:

This netwalker ransomware variant start with 3 stages as follows:

1st Layer : base 64 encoded powershell

2nd Layer: (after decoding the base64) is an encrypted array of bytes using xor command with decryption key of 0xc4, that will be run in scriptblock command.

3rd Layer : (after the decrypted 2nd layer) is a 2 sets of hex bytes array which is the x86 and 64 version of Netwalker binary files that will be injected in a process by a C# code that will be loaded and compile using powershell.

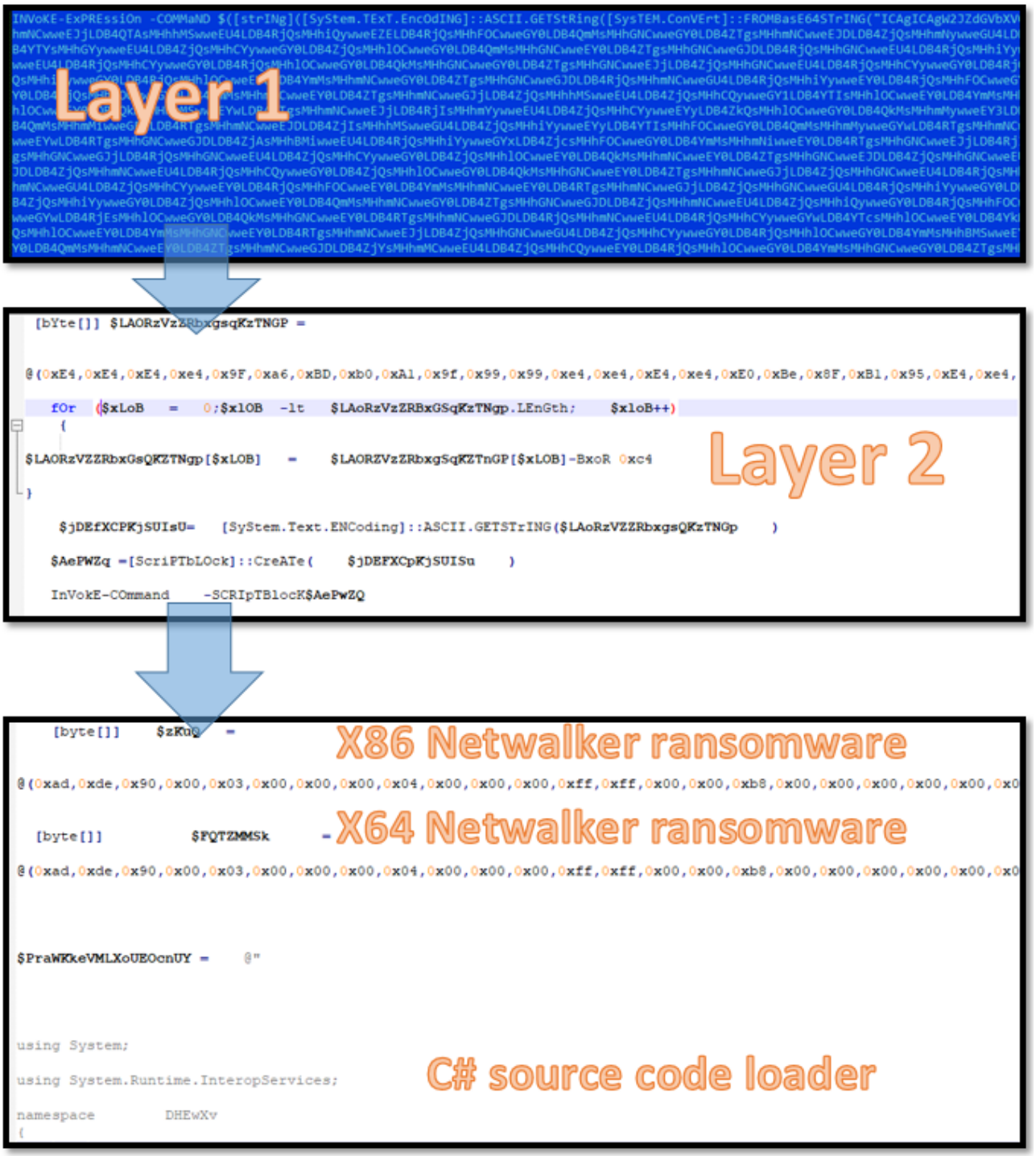


figure 1: the 3 layered powershell script

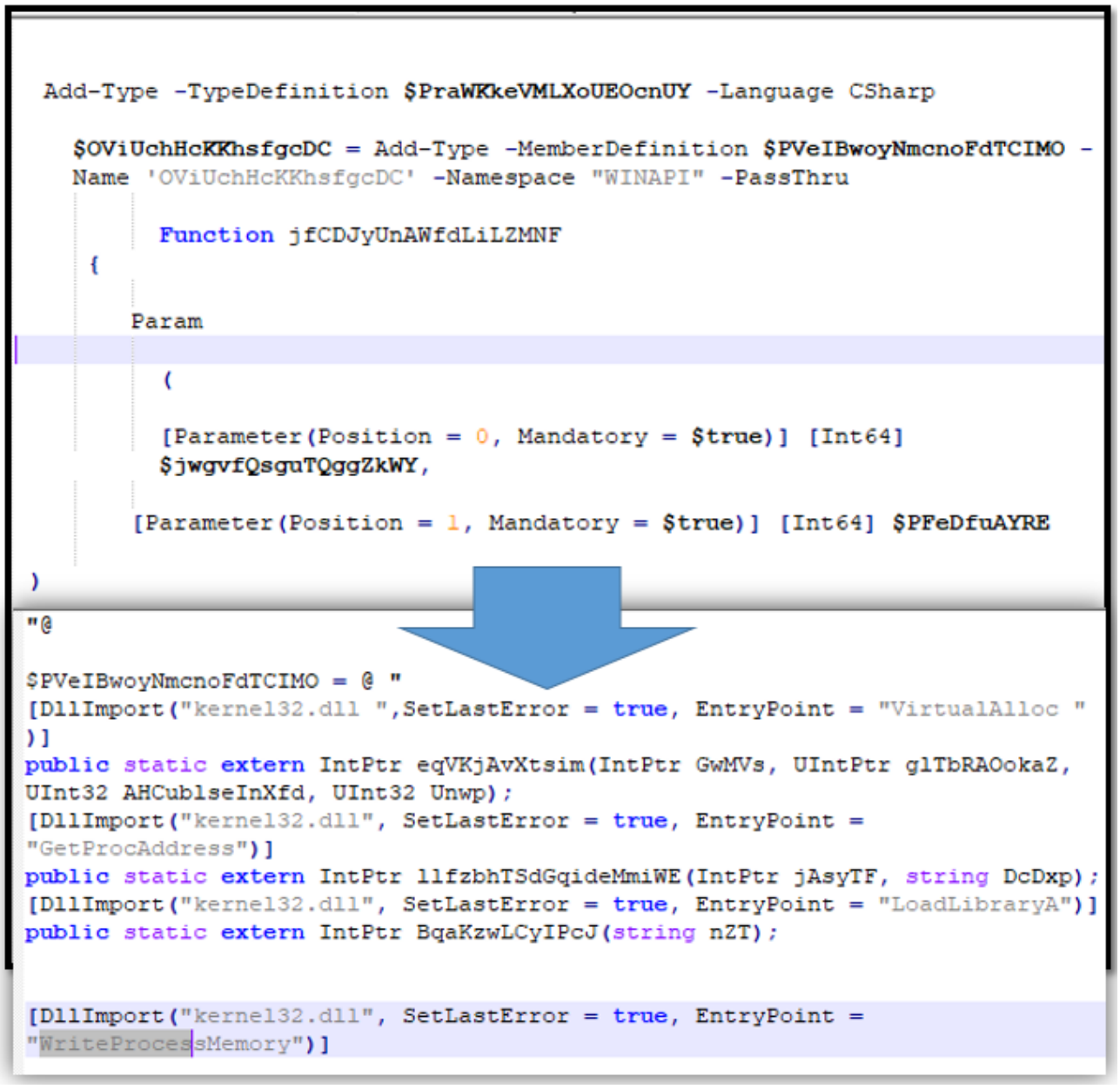


figure 2: the C# loader written in powershell using Add-Type

Stage 2: No MZ Header Binaries

as far as we saw in the last stage of the powershell, it will inject the ransomware (x86 or x64 binaries) to the explorer.exe process. The interesting part is after I decode those hex byte array, I notice that there are no MZ header to the binary file that are one technique to evade memory forensic tools or some quick check for injected executable to a process.

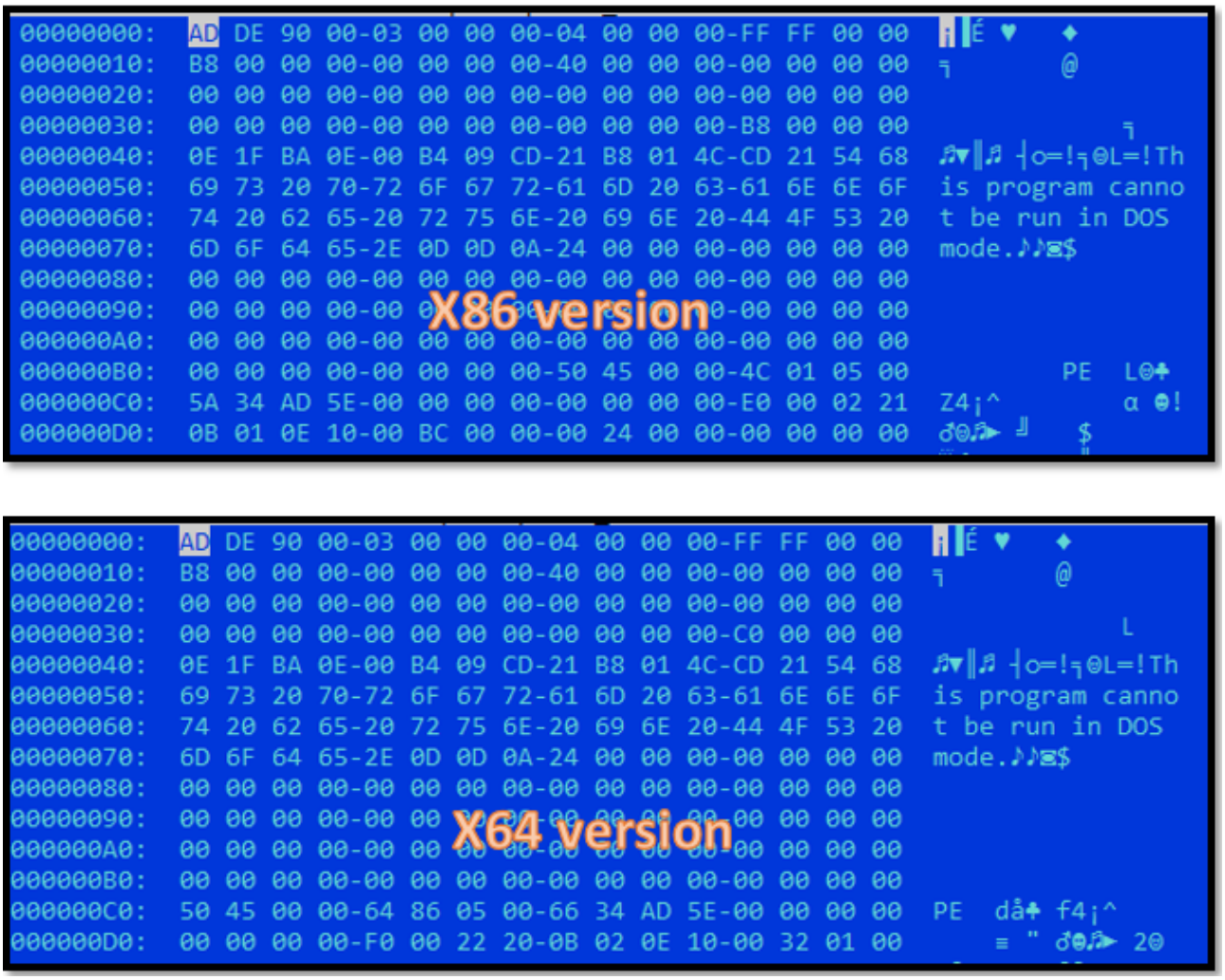


figure 3: NO MZ Header Files

Stage 3: Obfuscated API Call Using Structure

This Netwalker Ransomware has no import table. It will dynamically harvest its needed API using some hashing algorithm search to all export table of all needed DLL modules to executes its malicious code then save it to a structure object. Below is the screenshot how the raw Hexray view of the import harvesting before and after resolving the API hash and the structure Array using Idapython.

```
v0 = sub_10001010(-2067767744);
v1 = v0;
if ( !v0 )
  return dword_1000E1C8;
v2 = (int (__stdcall *)(int, int, int))sub_10001060(v0, 0xA1D45974);
if ( !v2 )
  return dword_1000E1C8;
v3 = sub_10004600(8, 648);
dword_1000E1CC = v2(v3, v14, v15);
if ( !dword_1000E1CC )
  return dword_1000E1C8;
*(_DWORD *)dword_1000E1CC = sub_10001060(v1, 0xA1D45974);
*(_DWORD *)(dword_1000E1CC + 4) = sub_10001060(v1, 0xAF11BC24);
*(_DWORD *)(dword_1000E1CC + 8) = sub_10001060(v1, 0xB973B8DC);
*(_DWORD *)(dword_1000E1CC + 12) = sub_10001060(v1, 0x8463960A);
*(_DWORD *)(dword_1000E1CC + 16) = sub_10001060(v1, 0xD141AFD3);
*(_DWORD *)(dword_1000E1CC + 20) = sub_10001060(v1, 0x57F17B6B);
*(_DWORD *)(dword_1000E1CC + 24) = sub_10001060(v1, 0x23398D9A);
*(_DWORD *)(dword_1000E1CC + 36) = sub_10001060(v1, 0xBD6735C3);
*(_DWORD *)(dword_1000E1CC + 40) = sub_10001060(v1, 0x900F6A6E);
*(_DWORD *)(dword_1000E1CC + 28) = sub_10001060(v1, 0xA8AE7412);
*(_DWORD *)(dword_1000E1CC + 32) = sub_10001060(v1, 0x4896A43);
*(_DWORD *)(dword_1000E1CC + 44) = sub_10001060(v1, 0x4C8A5B22);
*(_DWORD *)(dword_1000E1CC + 48) = sub_10001060(v1, 0x61E2048F);
*(_DWORD *)(dword_1000E1CC + 52) = sub_10001060(v1, 0x52FF8A3F);
*(_DWORD *)(dword_1000E1CC + 56) = sub_10001060(v1, 0xA312E4DE);
*(_DWORD *)(dword_1000E1CC + 60) = sub_10001060(v1, 0xCA3A8F9A);
*(_DWORD *)(dword_1000E1CC + 64) = sub_10001060(v1, 0x958F47AF);
*(_DWORD *)(dword_1000E1CC + 68) = sub_10001060(v1, 0x9AB4737E);
*(_DWORD *)(dword_1000E1CC + 72) = sub_10001060(v1, 0x7EF4BAE5);
*(_DWORD *)(dword_1000E1CC + 76) = sub_10001060(v1, 0x4A5A980C);
*(_DWORD *)(dword_1000E1CC + 80) = sub_10001060(v1, 0x7AA7B69B);
```

```
v0 = func_ResolveDllModuleName(0x84C05E40);
ntdll_ = v0;
if ( !v0 )
  return dword_1000E1C8;
v2 = (int (__stdcall *)(void *, int, int))func_parse_export_table(v0, ntdll_RtlAllocateHeap_HASH);
if ( !v2 )
  return dword_1000E1C8;
v3 = sub_10004600();
dwApImportStruct_1000E1CC = (DWORD *)v2(v3, 8, 0x288);
if ( !dwApImportStruct_1000E1CC )
  return dword_1000E1C8;
*dwApImportStruct_1000E1CC = func_parse_export_table(ntdll_, -1579918988);
dwApImportStruct_1000E1CC[1] = func_parse_export_table(ntdll_, ntdll_RtlFreeHeap_HASH);
dwApImportStruct_1000E1CC[2] = func_parse_export_table(ntdll_, ntdll_RtlReAllocateHeap_HASH);
dwApImportStruct_1000E1CC[3] = func_parse_export_table(ntdll_, ntdll_memset_HASH);
dwApImportStruct_1000E1CC[4] = func_parse_export_table(ntdll_, ntdll_memcpy_HASH);
dwApImportStruct_1000E1CC[5] = func_parse_export_table(ntdll_, ntdll_memcmp_HASH);
dwApImportStruct_1000E1CC[6] = func_parse_export_table(ntdll_, ntdll_sprintf_HASH);
dwApImportStruct_1000E1CC[9] = func_parse_export_table(ntdll_, ntdll_strcpy_HASH);
dwApImportStruct_1000E1CC[0xA] = func_parse_export_table(ntdll_, ntdll_strcat_HASH);
dwApImportStruct_1000E1CC[7] = func_parse_export_table(ntdll_, ntdll_strchr_HASH);
dwApImportStruct_1000E1CC[8] = func_parse_export_table(ntdll_, ntdll_strtol_HASH);
dwApImportStruct_1000E1CC[0xB] = func_parse_export_table(ntdll_, ntdll_wcscpy_HASH);
dwApImportStruct_1000E1CC[0xC] = func_parse_export_table(ntdll_, ntdll_wcscat_HASH);
dwApImportStruct_1000E1CC[0xD] = func_parse_export_table(ntdll_, ntdll_strstr_HASH);
dwApImportStruct_1000E1CC[0xE] = func_parse_export_table(ntdll_, ntdll_wcsstr_HASH);
dwApImportStruct_1000E1CC[0xF] = func_parse_export_table(ntdll_, ntdll_wcscmp_HASH);
dwApImportStruct_1000E1CC[0x10] = func_parse_export_table(ntdll_, ntdll_wcsncmp_HASH);
dwApImportStruct_1000E1CC[0x11] = func_parse_export_table(ntdll_, ntdll_RtlRandomEx_HASH);
dwApImportStruct_1000E1CC[0x12] = func_parse_export_table(ntdll_, ntdll_RtlRandom_HASH);
dwApImportStruct_1000E1CC[0x13] = func_parse_export_table(ntdll_, ntdll_RtlInitAnsiString_HASH);
dwApImportStruct_1000E1CC[0x14] = func_parse_export_table(ntdll_, ntdll_RtlInitUnicodeString_HASH);
dwApImportStruct_1000E1CC[0x15] = func_parse_export_table(ntdll_, ntdll_RtlAnsiStringToUnicodeString_HASH);
dwApImportStruct_1000E1CC[0x16] = func_parse_export_table(ntdll_, ntdll_RtlUnicodeStringToAnsiString_HASH);
dwApImportStruct_1000E1CC[0x17] = func_parse_export_table(ntdll_, ntdll_RtlFreeUnicodeString_HASH);
dwApImportStruct_1000E1CC[0x18] = func_parse_export_table(ntdll_, ntdll_RtlFreeAnsiString_HASH);
```

figure 4: API harvesting Function

The Hashing Algorithm is really looks complicated base on its graph but actually it is just a loop of xor and rotate bit operation with specific keys.

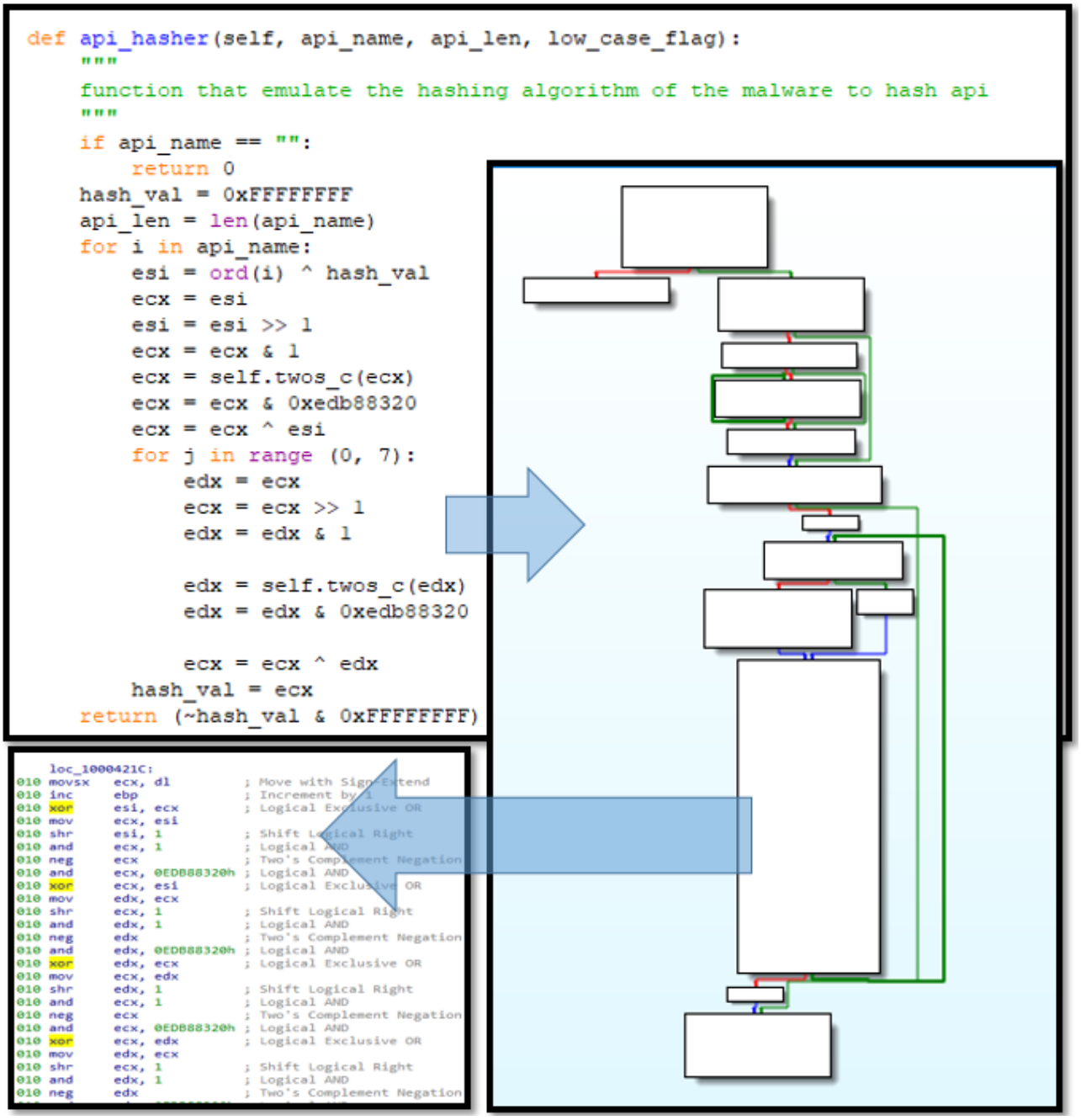


figure 5: Hashing algorithm

But the Obfuscation does not ends here. As we remember that it place the resolved API address into a structure object. Then this structure was initialized to a another variable by a function then do the access the member of the structure out of that which make the analysis more confusing.

figure 6: Declare multiple Structure as a obfuscation

Thanks for IDA Python for helping me in creating a structure out of harvested API it needs to make the static analysis more easily.

```python
def create_struct_sid():
    sid = idc.add_struc(-1, "ApiHashStructList", 0)
    return sid

def main():

    sid = create_struct_sid()

    for api_name in api_member.api_member:
        try:
            idc.add_struc_member(sid, str(api_name), -1, FF_DATA|FF_DWORD, 0, 4)
            print("[+] status: {} added to ApiHashStructList".format(api_name))
        except:
            print (sys.exc_info())
    return
```

```python
api_member = [
"ntdll_RtlAllocateHeap_HASH",
"ntdll_RtlFreeHeap_HASH",
"ntdll_RtlReAllocateHeap_HASH",
"ntdll_memset_HASH",
"ntdll_memcpy_HASH",
"ntdll_memcmp_HASH",
"ntdll_sprintf_HASH",
"ntdll_strchr_HASH",
"ntdll_strtol_HASH",
"ntdll_strcpy_HASH",
"ntdll_strcat_HASH",
"ntdll_wcscpy_HASH",
"ntdll_wcscat_HASH",
"ntdll_strstr_HASH",
"ntdll_wcsstr_HASH",
"ntdll_wcscmp_HASH",
"ntdll_wcsncmp_HASH",
"ntdll_RtlRandomEx_HASH",
"ntdll_RtlRandom_HASH",
"ntdll_RtlInitAnsiString_HASH",
"ntdll_RtlInitUnicodeString_HASH",
"ntdll_RtlAnsiStringToUnicodeString_HASH",
"ntdll_RtlUnicodeStringToAnsiString_HASH",
"ntdll_RtlFreeUnicodeString_HASH",
"ntdll_RtlFreeAnsiString_HASH",
"ntdll_LdrLoadDll_HASH",
"ntdll_RtlAdjustPrivilege_HASH",
"ntdll_NtQuerySystemInformation_HASH",
"ntdll_NtOpenProcess_HASH",
```

figure 7: Add Structure

Lesson Learn:

I learned that the there are so many way to obfuscate code from analysis and even the data structure can be used to make the analysis little bit confusing during analysis like what I experience. :)

IOC:

https://app.any.run/tasks/6bb00be0-cd0a-4d9a-a1ea-72cd275ded0e/

Powershell:

filename: powershell.ps1
md5: 5bec43ea21e95a68abafa8c7f99d1e6c
sha1: 22df933f2b33f3f4ffee22b51b4f8fa0268bb327
sha256: b7c7fa9b74aacf331871a9e5438678bce46002618fa106429225161d94e22e44

x64 Netwalker Ransomware:

filename: x64.bin
md5: bc96c744bd66ddfaa79d467b757b8628
sha1: a379f9e04708d773a2dec897166780b026f4c4ea
sha256: 2c245db9fb9b2c6e84832662dda3dfff3c6b21128d9fec115f5b989fb090841d

x86 Netwalker Ransomware:

filename: x86_raw.bin
md5: de61b852cadac6afe307652b187ca5df
sha1: fa02c1d394bc150d8a62d3f991d0fdc042ee9724
sha256: e8c5c0b70d45a5dc80d678ed7102abf9882efb9cbc2cff20f171d60d5205051d