SysWhispers2 analysis 🙊

blog.krakz.fr/notes/syswhispers2/

March 10, 2024

This helper comes in handy when reversing samples that use SysWhispers2 to recover ntdll call from SysWhispers2 hashes.

Readme.md

SysWhispers github.com/jthuraisamy/SysWhispers2 helps with evasion by generating header/ASM files implants can use to make direct system calls.

Various security products place hooks in user-mode API functions which allow them to redirect execution flow to their engines and detect for suspicious behaviour. The functions in `ntdll.dll` that make the syscalls consist of just a few assembly instructions, so re-implementing them in your own implant can bypass the triggering of those security product hooks. This technique was popularized by @Cn33liz and his blog post has more technical details worth reading.

Analysis

VMray recently tweeted that Pikabot incorporates SysWhispers2 This note offers a step-by-step guide to identify the syscalls made by malware that utilizes SysWhispers2, a technique that can be applied in any situation where SysWhispers2 is present. *NB: Tools: IDA decompiler and xdbg* The analysis began with the sample `PERFERENDISF.jar` shared in VMRay tweet, which is available on Malware Bazaar, with the SHA-256: d26ab01b293b2d439a20d1dffc02a5c9f2523446d811192836e26d370a34d1b4

We skipped to the stage 2 of the Pikabot loader, which employs **SysWhispers2** to load the malware's core. The malware executes the following steps to perform a direct syscall:

1. Saves the return address;
2. Resolves the syscall ID from a hash (a behavior related to SysWhispers2);
3. Retrieves a stub to invoke the syscall based on the host architecture;
4. Executes the syscall and resumes program execution.

```
2  int __cdecl call_direct_syscall(DWORD arg_api_hash)
3  {
4    int v2; // [esp-8h] [ebp-8h]
5    int retaddr; // [esp+0h] [ebp+0h]
6
7    dword_4126168 = v2;
8    saved_ret_address = retaddr;                  // to return the original caller
9    api_hash = (DWORD)&arg_api_hash;
10   syscall_id = SW2_GetSyscallNumber(arg_api_hash);
11   syscall_stub_offset_address = (int)get_stub_offets_addr(NtCurrentTeb()->WOW32Reserved != 0);
12   ((void (*)(void))syscall_stub_offset_address)();
13   return ((int (*)(void))saved_ret_address)();
14 }
```

Figure 1: Function used to made the direct syscall

Here are examples of direct syscalls made by the malware.

```
                        ; int sub_4111097()
                        sub_4111097     proc near               ; COI
68 C4 9C 48 85                          push    85489CC4h
E8 5F FF FF FF                          call    call_direct_syscall
68 B7 7C 22 70                          push    70227CB7h
E8 55 FF FF FF                          call    call_direct_syscall
                        sub_4111097     endp ; sp-analysis failed


                        ; =============== S U B R O U T I N E =======:


                        ; int sub_41110AB()
                        sub_41110AB     proc near               ; COI
68 E8 F0 54 C6                          push    0C654F0E8h
E8 4B FF FF FF                          call    call_direct_syscall
                        sub_41110AB     endp ; sp-analysis failed
```

Figure 2: Example of SW2Syscall stubs

To operate SysWhispers2, it is necessary to populate the `_SW2_SYSCALL_LIST` structure, which is an array containing correspondences between hashes and `ntdll.dll` addresses. According to the file `base.h` jthuraisamy/SysWhispers2/blob/main/data/base.h the two structures are:

```
struct _SW2_SYSCALL_ENTRY
{
    DWORD Hash;
    DWORD Address;
}
```

Code Snippet 1: SysWhispers2 syscall entry

The `Hash` field contains a hash value corresponding to a particular syscall, and the `Address` field contains the address of the corresponding function in `ntdll.dll`.

```
struct _SW2_SYSCALL_LIST
{
    DWORD Count;
    SW2_SYSCALL_ENTRY Entries[SW2_MAX_ENTRIES];
}
```

Code Snippet 2: SysWhispers2 syscall list

The malware stores a pointer to the syscall list as a **global** variable, which is convenient when we later retrieve the populated data with the debugger.

```
.data:04137F1B 00                                      db    0
.data:04137F1C                          ; _SW2_SYSCALL_LIST SW2_SyscallList
.data:04137F1C 00 00 00 00 00 00 00 00 SW2_SyscallList _SW2_SYSCALL_LIST <0, <0, 1>>
.data:04137F1C 01 00 00 00 00 00 00 00                                    ; DATA X
.data:04137F1C 00 00 00 00 00 00 00 00…                                   ; SW2_Po
.data:04138EC0 00                                      db    0
```

Figure 3: Reference of the _SW2_SYSCALL_LIST structure

According to the source code See function `SW2_GetSyscallNumber` line 131. the function used to get the address in ntdll from hash ensure that `_SW2_SYSCALL_LIST` structure is populated.

The most “challenging” task is now to identify a call to `SW2_GetSyscallNumber` and set a breakpoint after the `SW2_PopulateSyscallList` function, at which point a dump of the list can be made.

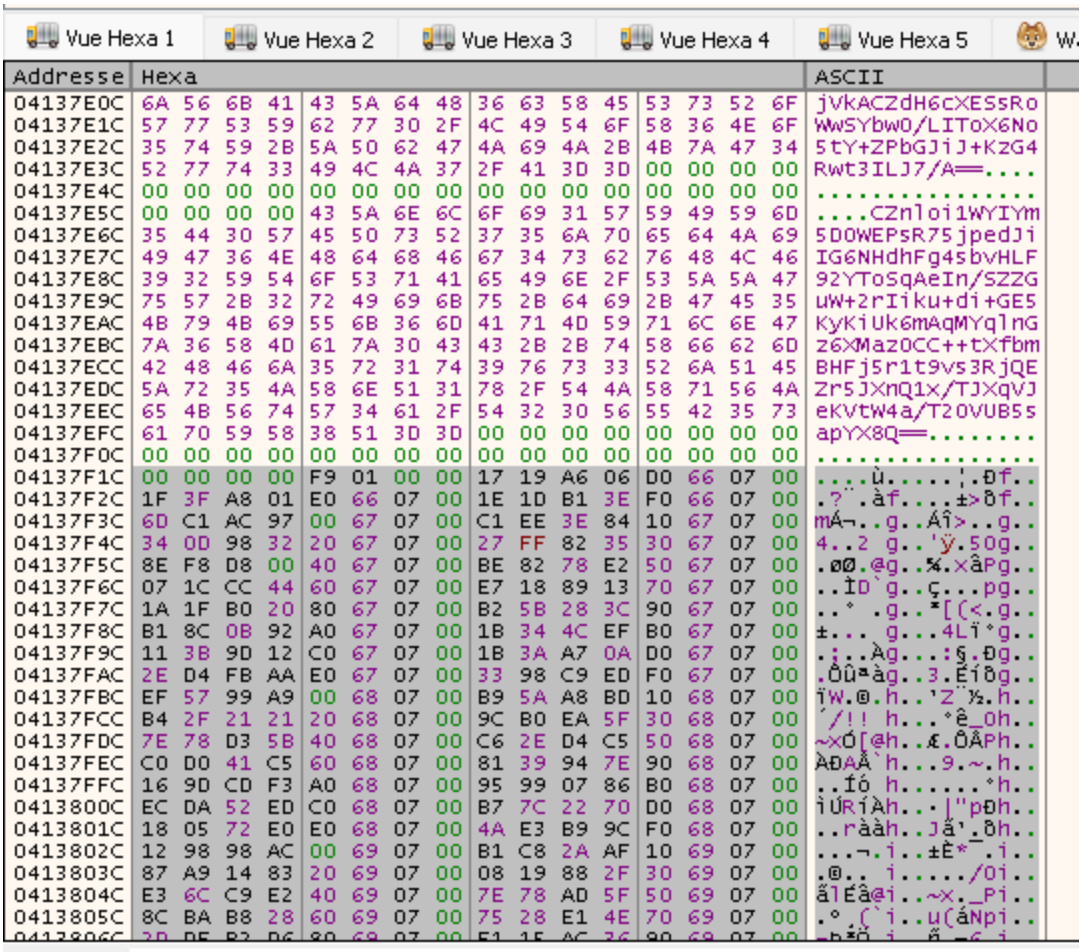

Figure 4: Hex memory view of the _SW2_SYSCALL_LIST structure populated

Here is a clearest visualization of the memory using ImHex.

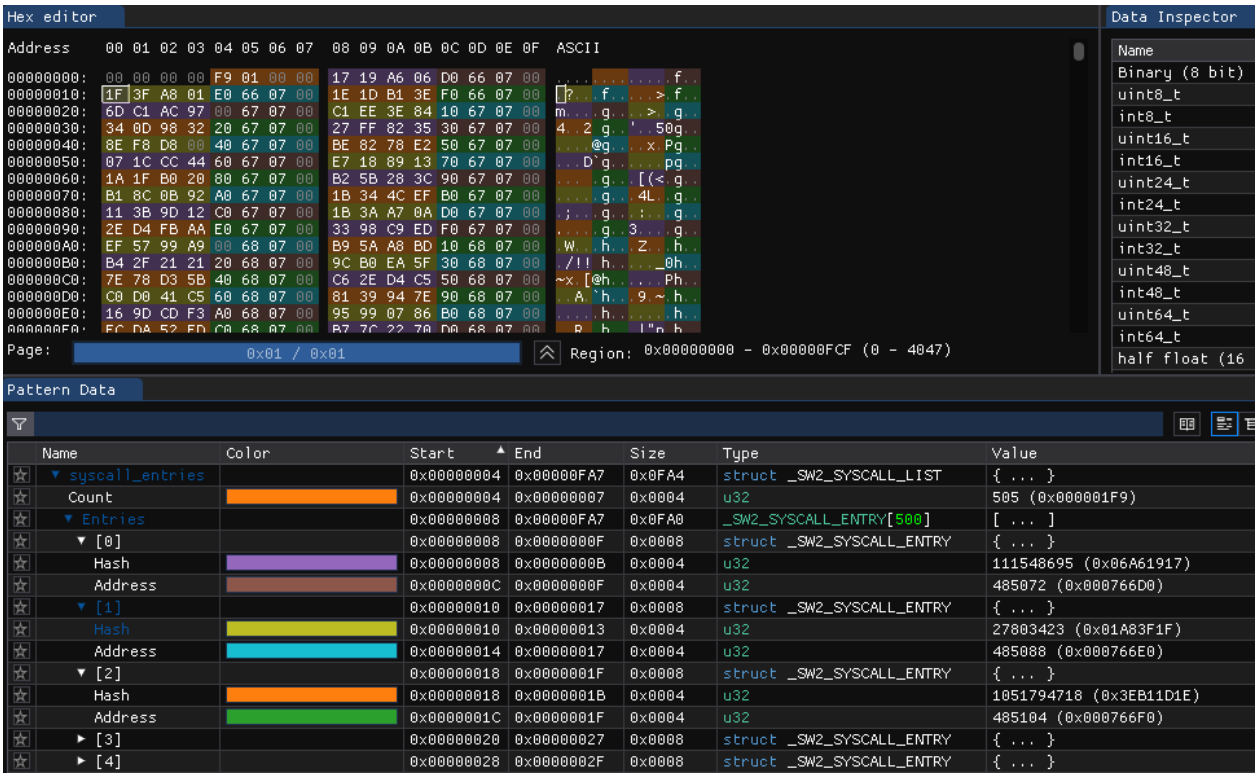

Figure 5: Visualization of the _SW2_SYSCALL_LIST structure populated

Mapping Hashes to Syscalls

First, the hashes (*SW2*) must be listed, and then the hash must be resolved to obtain the syscall number.

The following IDA script lists the hashes by retrieving the first (single one) function argument:

```
s2w_direct_call_addr = 0x04111000

for x in XrefsTo(s2w_direct_call_addr):
    syscall_hash = get_wide_dword(x.frm - 0x4) # First args of the function
    print(f"call to SW2 at:0x{x.frm:x} hash:0x{syscall_hash:x}")
```

Which gives the following hashes: `0x312294161`, `0x228075779`, `0x2553518241`, `0x3309424832`, `0x1605204094`, `0x2236128452`, `0x1881308343`, `0x3327455464`, `0x3319017158`, `0x2249560824`, `0x397169428`, `0x4066245879`, `0x2629212700`.

Subsequently, the `_SW2_SYSCALL_LIST` structure was parsed to obtain the address corresponding to each of the aforementioned hashes.

```
import struct

with open("syscall_entries.dmp", "rb") as f:
    # offset 0x8 is used to remove the DWORD Count of the struct _SW2_SYSCALL_LIST
    SW2_syscallList_raw = f.read()[0x8:]

NTDLL_BASE_ADDRESS = 0x77DA0000 # specifics for each sample
SW2_Entrie = namedtuple("SW2_Entrie", ["hash", "address"])
SW2_syscallList: List = []

for hash, addr_offset in struct.iter_unpack("<Li", SW2_syscallList_raw):
    print(f"0x{hash:x} 0x{addr_offset + NTDLL_BASE_ADDRESS:x}")
    SW2_syscallList.append(SW2_Entrie(hash, addr_offset + NTDLL_BASE_ADDRESS))
```

Next, take a snapshot of ntdll (*to avoid rebasing the DLL base address*) to list the export functions of `ntdll.dll` and their corresponding addresses.

The subsequent step involves taking a snapshot of `ntdll.dll` to obtain a list of its export functions along with their corresponding address. *This approach eliminates the need to rebase the DLL base address.*

```
import pefile

def get_section(pe: pefile.PE, section_name: str) -> pefile.SectionStructure:
    """return section by name, if not found raise KeyError exception."""
    for section in filter(
        lambda x: x.Name.startswith(section_name.encode()), pe.sections
    ):
        return section
    raise KeyError(f"{section_name} not found")

PE_FILE = "ntdll.dll"
pe = pefile.PE(PE_FILE)

text = get_section(pe, ".text")
image_base = pe.OPTIONAL_HEADER.ImageBase
section_rva = text.VirtualAddress

mapping_syscall_id_fn = []
# Build a corresponding address and ntdll function name
for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
    mapping_syscall_id_fn.append((pe.OPTIONAL_HEADER.ImageBase + exp.address,
exp.name))
```

Finally, **map** the addresses populated in the `_SW2_SYSCALL_ENTRIES` structure with the corresponding addresses exported from `ntdll.dll` to obtain their export names.

```
# hashes obtained in IDA
hashes = [
    0x129D3B11,
    0xD982903,
    0x983398A1,
    0xC541D0C0,
    0x5FAD787E,
    0x85489CC4,
    0x70227CB7,
    0xC654F0E8,
    0xC5D42EC6,
    0x861592F8,
    0x17AC5314,
    0xF25DFCF7,
    0x9CB69A1C,
]

def find_syscall_by_hash(hash) -> Optional[SW2_Entrie]:
    for syscall in SW2_syscallList:
        if syscall.hash == hash:
            return syscall

for addr, name in mapping_syscall_id_fn:
    for syscall in map(find_syscall_by_hash, hashes):
        if addr == syscall.address:
            print(f"0x{syscall.hash:x} <-> {name.decode()}")
            break
```

Output for this sample of Pikabot is:

```
0xc5d42ec6 <-> NtAllocateVirtualMemory
0x129d3b11 <-> NtClose
0x85489cc4 <-> NtCreateUserProcess
0x70227cb7 <-> NtFreeVirtualMemory
0x17ac5314 <-> NtGetContextThread
0x5fad787e <-> NtOpenProcess
0xc541d0c0 <-> NtQueryInformationProcess
0x983398a1 <-> NtQuerySystemInformation
0xc654f0e8 <-> NtReadVirtualMemory
0x9cb69a1c <-> NtResumeThread
0xf25dfcf7 <-> NtSetContextThread
0xd982903 <-> NtSystemDebugControl
0x861592f8 <-> NtWriteVirtualMemory
0xc5d42ec6 <-> ZwAllocateVirtualMemory
0x129d3b11 <-> ZwClose
0x85489cc4 <-> ZwCreateUserProcess
0x70227cb7 <-> ZwFreeVirtualMemory
0x17ac5314 <-> ZwGetContextThread
0x5fad787e <-> ZwOpenProcess
0xc541d0c0 <-> ZwQueryInformationProcess
0x983398a1 <-> ZwQuerySystemInformation
0xc654f0e8 <-> ZwReadVirtualMemory
0x9cb69a1c <-> ZwResumeThread
0xf25dfcf7 <-> ZwSetContextThread
0xd982903 <-> ZwSystemDebugControl
0x861592f8 <-> ZwWriteVirtualMemory
```

The full script is available on this gist, along with the *S2W_SyscallList.dmp* file in hexadecimal format. To use the dump, replace lines 32 to 34 with the following:

```
import binascii
with open("SW2_SyscallList_hex.dmp", "r") as f:
    # offset 0x8 is used to remove the DWORD Count of the struct _SW2_SYSCALL_LIST
    SW2_syscallList_raw = binascii.unhexlify(f.read())[0x8:]
```

Resources