SysWhispers2 analysis 🙊

By pbo

Published: 2024-03-09 · Archived: 2026-04-05 13:37:44 UTC

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](https://blog.krakz.fr/notes/syswhispers2/) 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.

```
Vue Hexa 1   Vue Hexa 2   Vue Hexa 3   Vue Hexa 4   Vue Hexa 5   W…
Addresse  Hexa                                             ASCII
04137E0C  6A 56 6B 41 43 5A 64 48 36 63 58 45 53 73 52 6F  jVkACZdH6cXESsRo
04137E1C  57 77 53 59 62 77 30 2F 4C 49 54 6F 58 36 4E 6F  WwSYbw0/LIToX6No
04137E2C  35 74 59 2B 5A 50 62 47 4A 69 4A 2B 4B 7A 47 34  5tY+ZPbGJiJ+KzG4
04137E3C  52 77 74 33 49 4C 4A 37 2F 41 3D 3D 00 00 00 00  Rwt3ILJ7/A==....
04137E4C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
04137E5C  00 00 00 00 43 5A 6E 6C 6F 69 31 57 59 49 59 6D  ....CZnloi1WYIYm
04137E6C  35 44 30 57 45 50 73 52 37 35 6A 70 65 64 4A 69  5D0WEPsR75jpedJi
04137E7C  49 47 36 4E 48 64 68 46 67 34 73 62 76 48 4C 46  IG6NHdhFg4sbvHLF
04137E8C  39 32 59 54 6F 53 71 41 65 49 6E 2F 53 5A 5A 47  92YToSqAeIn/SZZG
04137E9C  75 57 2B 32 72 49 69 6B 75 2B 64 69 2B 47 45 35  uW+2rIiku+di+GE5
04137EAC  4B 79 4B 69 55 6B 36 6D 41 71 4D 59 71 6C 6E 47  KyKiUk6mAqMYqlnG
04137EBC  7A 36 58 4D 61 7A 30 43 43 2B 2B 74 58 66 62 6D  z6XMaz0CC++tXfbm
04137ECC  42 48 46 6A 35 72 31 74 39 76 73 33 52 6A 51 45  BHFj5r1t9vs3RjQE
04137EDC  5A 72 35 4A 58 6E 51 31 78 2F 54 4A 58 71 56 4A  Zr5JXnQ1x/TJXqVJ
04137EEC  65 4B 56 74 57 34 61 2F 54 32 30 56 55 42 35 73  eKVtW4a/T20VUB5s
04137EFC  61 70 59 58 38 51 3D 3D 00 00 00 00 00 00 00 00  apYX8Q==........
04137F0C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
04137F1C  00 00 00 00 F9 01 00 00 17 19 A6 06 D0 66 07 00  ....ù.....¦.Ðf..
04137F2C  1F 3F A8 01 E0 66 07 00 1E 1D B1 3E F0 66 07 00  .?¨.àf....±>ðf..
04137F3C  6D C1 AC 97 00 67 07 00 C1 EE 3E 84 10 67 07 00  mÁ¬..g..Áî>..g..
04137F4C  34 0D 98 32 20 67 07 00 27 FF 82 35 30 67 07 00  4..2 g..'ÿ.50g..
04137F5C  8E F8 D8 00 40 67 07 00 BE 82 78 E2 50 67 07 00  .øØ.@g..¾.xâPg..
04137F6C  07 1C CC 44 60 67 07 00 E7 18 89 13 70 67 07 00  ..ÌD`g..ç...pg..
04137F7C  1A 1F B0 20 80 67 07 00 B2 5B 28 3C 90 67 07 00  ..° .g..²[(<.g..
04137F8C  B1 8C 0B 92 A0 67 07 00 1B 34 4C EF B0 67 07 00  ±... g...4Lï°g..
04137F9C  11 3B 9D 12 C0 67 07 00 1B 3A A7 0A D0 67 07 00  .;..Àg...:§.Ðg..
04137FAC  2E D4 FB AA E0 67 07 00 33 98 C9 ED F0 67 07 00  .Ôûªàg..3.Éíðg..
04137FBC  EF 57 99 A9 00 68 07 00 B9 5A A8 BD 10 68 07 00  ïW.©.h..¹Z¨½.h..
04137FCC  B4 2F 21 21 20 68 07 00 9C B0 EA 5F 30 68 07 00  ´/!! h...°ê_0h..
04137FDC  7E 78 D3 5B 40 68 07 00 C6 2E D4 C5 50 68 07 00  ~xÓ[@h..Æ.ÔÅPh..
04137FEC  C0 D0 41 C5 60 68 07 00 81 39 94 7E 90 68 07 00  ÀÐAÅ`h...9.~.h..
04137FFC  16 9D CD F3 A0 68 07 00 95 99 07 86 B0 68 07 00  ..Íó h......°h..
0413800C  EC DA 52 ED C0 68 07 00 B7 7C 22 70 D0 68 07 00  ìÚRíÀh..·|"pÐh..
0413801C  18 05 72 E0 E0 68 07 00 4A E3 B9 9C F0 68 07 00  ..rààh..Jã¹.ðh..
0413802C  12 98 98 AC 00 69 07 00 B1 C8 2A AF 10 69 07 00  ...¬.i..±È*¯.i..
0413803C  87 A9 14 83 20 69 07 00 08 19 88 2F 30 69 07 00  .©.. i..../0i..
0413804C  E3 6C C9 E2 40 69 07 00 7E 78 AD 5F 50 69 07 00  ãlÉâ@i..~x._Pi..
0413805C  8C BA B8 28 60 69 07 00 75 28 E1 4E 70 69 07 00  .º¸(`i..u(áNpi..
```

Figure 4: Hex memory view of the _SW2_SYSCALL_LIST structure populated

Here is a clearest visualization of the memory using [ImHex](https://blog.krakz.fr/notes/syswhispers2/).

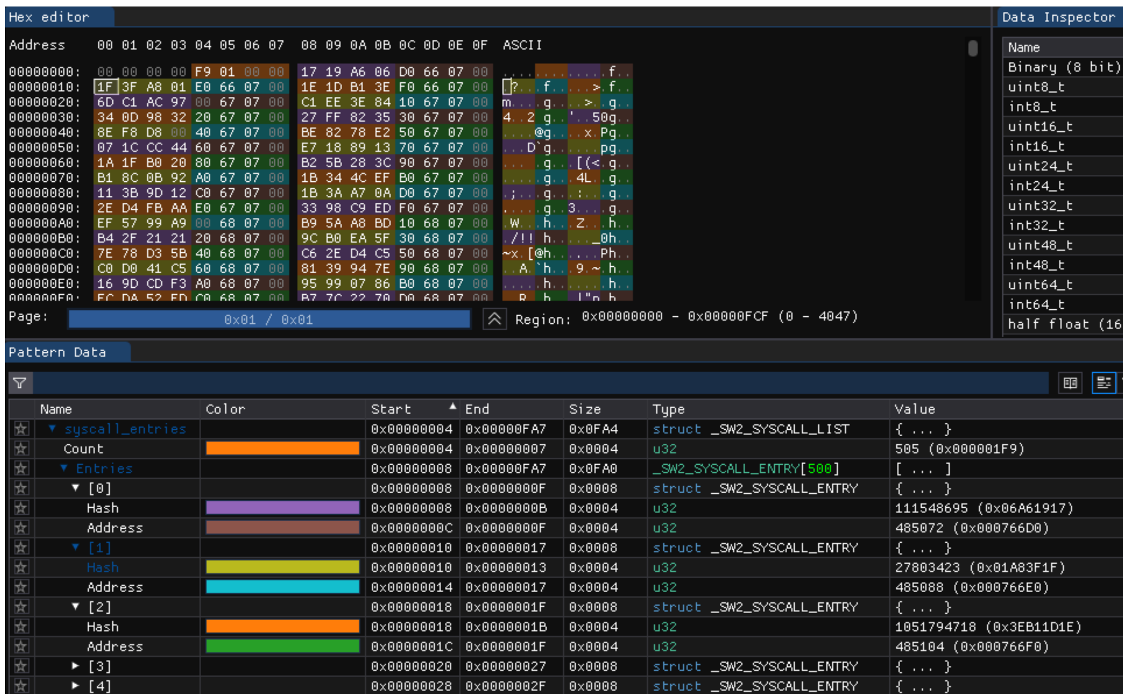

Figure 5: Visualization of the _SW2_SYSCALL_LIST structure populated

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

- https://github.com/jthuraisamy/SysWhispers2
- https://twitter.com/vmray/status/1760647508038947080
- https://gist.github.com/lbpierre/c9c39de0c32bb96a5e12556f75744d42
- https://joshfinley.github.io/posts/2020-04-17-sycall-dumping/

Source: https://blog.krakz.fr/notes/syswhispers2/