Resolving System Service Numbers using the Exception Directory

mdsec.co.uk/2022/04/resolving-system-service-numbers-using-the-exception-directory

19 April 2022

Introduction

While developing new features for Nighthawk C2, we observed that NTDLL contains up to three internal tables with the Relative Virtual Address (RVA) of all system calls. Two of these tables include addresses sorted in ascending order allowing us to obtain the System Service Number (SSN) for a call. The three tables in question are:

1. Export Address Table (**IMAGE_EXPORT_DIRECTORY**)
2. Runtime Function Table (**IMAGE_RUNTIME_FUNCTION_ENTRY**)
3. Guard CF Function Table (**IMAGE_LOAD_CONFIG_DIRECTORY**)

SSNs are required to invoke Windows system calls directly and bypass user-mode hooks installed by third-party applications. The problem is that these numbers differ from XP to Windows 11 and are continually changing with each new release. Some of the options we have to avoid hardcoding values include:

1. Using OS version information to select the correct SSN. (SysWhispers)
2. Reading the syscall stubs from disk or the KnownDlls directory object. (Windows 10 Parallel Loader)
3. Extracting the SSN from NTDLL in memory. (HellsGate)
4. Sorting the address of system calls in ascending order using NTDLL in memory. (SysWhispers2, FreshyCalls)
5. Extracting the SSN from NTDLL in memory. (HalosGate)

Guard CF Function Table

Beginning with Windows 8.1 Update 3 (KB3000850), Control Flow Guard (CFG) was added to mitigate against redirecting the flow of execution, something typically used by malicious code such as an exploit. When CFG is enabled at compile time using */guard:cf switch*, MSVC will wrap all the indirect calls in the resulting binary with a call to *_guard_dispatch_icall* that determines if execution should continue or raise an exception. It will also generate a list of approved targets within the given binary stored in the Load Configuration data directory of the PE header.

We can read the approved list, including the address of all system calls, by manually parsing the PE header or calling the GetImageConfigInformation API that returns a pointer to an **IMAGE_LOAD_CONFIG_DIRECTORY** structure containing a pointer to the table

stored at the **GuardCFFunctionTable** field. Because it remains undocumented, the following structure is taken from the sources to Process Hacker.

```
// Structure for Windows 10
#pragma pack(1)
typedef struct _IMAGE_CFG_ENTRY {
        DWORD Rva;
    struct {
        BOOLEAN SuppressedCall : 1;
        BOOLEAN ExportSuppressed : 1;
        BOOLEAN LangExcptHandler : 1;
        BOOLEAN Xfg : 1;
        BOOLEAN Reserved : 4;
    } Flags;
} IMAGE_CFG_ENTRY, *PIMAGE_CFG_ENTRY;
```

Examining earlier versions of NTDLL indicate revisions have been made. For example, the Flags field doesn't exist on Windows 8.1.

```
// Structure for Windows 8.1
typedef struct _IMAGE_CFG_ENTRY {
        DWORD Rva;
} IMAGE_CFG_ENTRY, *PIMAGE_CFG_ENTRY;
```

As a result of differences in the structure and the availability of CFG before Windows 8.1, it mightn't be safe to use, so let's examine the runtime table instead.

Runtime Function Table

Beginning with the 64-Bit version of Windows XP and Server 2003, exception handling uses a table of **RUNTIME_FUNCTION** entries to safely unwind stacks and recover from a crash. We can parse these entries by reading the Exception Directory, and thankfully, the structure of each entry is the exact same for Windows XP up to Windows 11.

```
typedef struct _IMAGE_RUNTIME_FUNCTION_ENTRY {
  DWORD BeginAddress;
  DWORD EndAddress;

  union {
    DWORD UnwindInfoAddress;
    DWORD UnwindData;
  } DUMMYUNIONNAME;
} RUNTIME_FUNCTION, *PRUNTIME_FUNCTION,
_IMAGE_RUNTIME_FUNCTION_ENTRY, *_PIMAGE_RUNTIME_FUNCTION_ENTRY;
```

First, we set a variable, *SSN* to zero. Then for every entry in the runtime table, search for the value of *BeginAddress* in *AddressOfFunctions*. When we find a match, use the same table index to read a symbol from the *AddressOfNames* table. Then compare this symbol with

what we're searching for. When symbols match, exit the loop by returning the value of *SSN*. When symbols don't match, we check if it begins with "Zw" and if true, increment *SSN*. Then continue searching. Some code should help clarify the process:

```
//
// Resolve System Service Number (SSN) by System Call Name.
//

#include <ltwindows.h>
#include <winternl.h>

int
GetSsnByName(PCHAR syscall) {
    auto Ldr = (PPEB_LDR_DATA)NtCurrentTeb()->ProcessEnvironmentBlock->Ldr;
    auto Head = (PLIST_ENTRY)&Ldr->Reserved2[1];
    auto Next = Head->Flink;

    while (Next != Head) {
        auto ent = CONTAINING_RECORD(Next, LDR_DATA_TABLE_ENTRY, Reserved1[0]);
        Next = Next->Flink;
        auto m = (PBYTE)ent->DllBase;
        auto nt = (PIMAGE_NT_HEADERS)(m + ((PIMAGE_DOS_HEADER)m)->e_lfanew);
        auto rva = nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
        if (!rva) continue; // no export table? skip

        auto exp = (PIMAGE_EXPORT_DIRECTORY)(m + rva);
        if (!exp->NumberOfNames) continue;   // no symbols? skip
        auto dll = (PDWORD)(m + exp->Name);

        // not ntdll.dll? skip
        if ((dll[0] | 0x20202020) != 'ldtn') continue;
        if ((dll[1] | 0x20202020) != 'ld.l') continue;
        if ((*(USHORT*)&dll[2] | 0x0020) != '\x00l') continue;

        // Load the Exception Directory.
        rva = nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION].VirtualAddress;
        if (!rva) return -1;
        auto rtf = (PIMAGE_RUNTIME_FUNCTION_ENTRY)(m + rva);

        // Load the Export Address Table.
        rva = nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
        auto adr = (PDWORD)(m + exp->AddressOfFunctions);
        auto sym = (PDWORD)(m + exp->AddressOfNames);
        auto ord = (PWORD)(m + exp->AddressOfNameOrdinals);

        int ssn = 0;

        // Search runtime function table.
        for (int i=0; rtf[i].BeginAddress; i++) {
            // Search export address table.
            for (int j=0; j<exp->NumberOfFunctions; j++) {
                // begin address rva?
                if (adr[ord[j]] == rtf[i].BeginAddress) {
                    auto api = (PCHAR)(m + sym[j]);
                    auto s1 = api;
                    auto s2 = syscall;
```

```
                    // our system call? if true, return ssn
                    while (*s1 && (*s1 == *s2)) s1++, s2++;
                    int cmp = (int)*(PBYTE)s1 - *(PBYTE)s2;
                    if (!cmp) return ssn;

                    // if this is a syscall, increase the ssn value.
                    if (*(USHORT*)api == 'wZ') ssn++;
                }
            }
        }
    }
    return -1; // didn't find it.
}
```

Summary

Using the above code, we can avoid using dynamic memory allocation or global variables to store information about all the system calls available. This code demonstrates how to retrieve an SSN using the symbol name but could be updated to support using a hash.

This post was written by @modexpblog.

MDSec Research