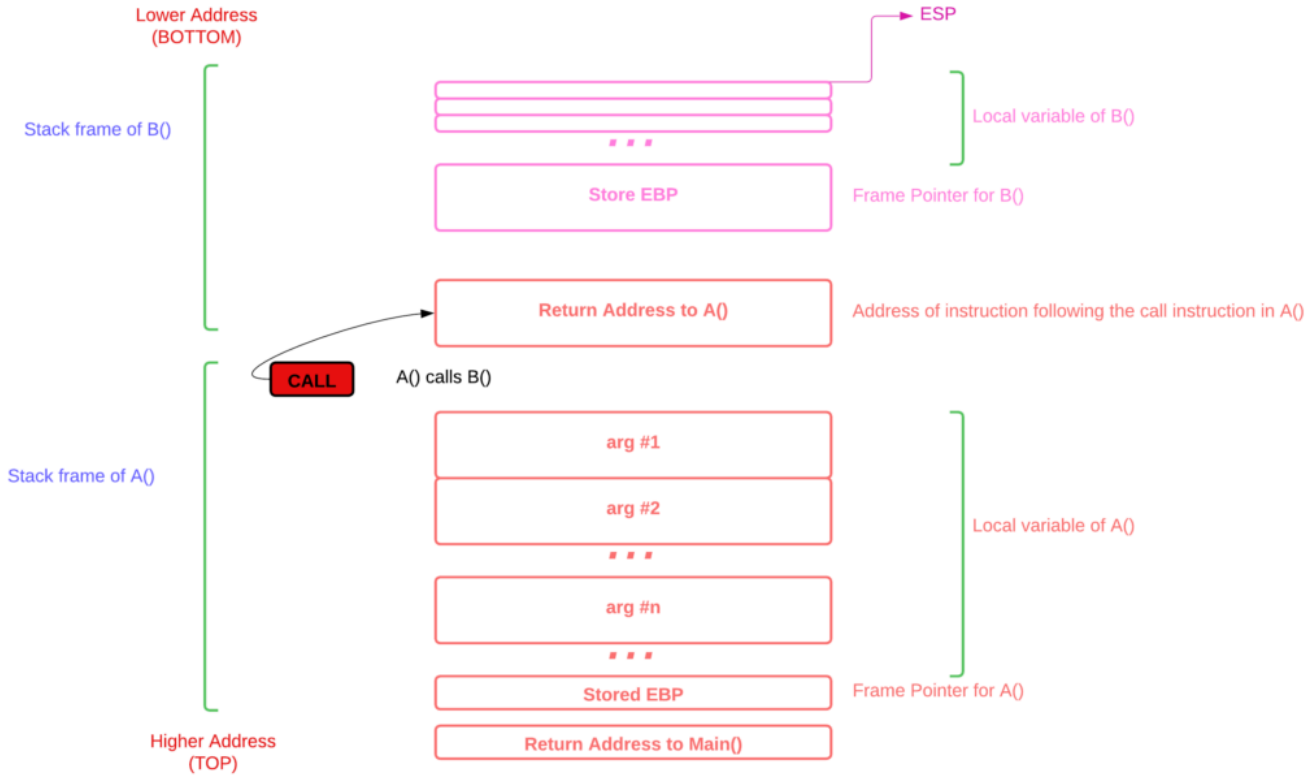
Stack Manipulation via Hardware breakpoint & Direct Syscall Execution

sabotagesec.com/stack-manipulation-via-hardware-breakpoint-direct-syscall-execution

x86 Stack Primer



In this section we will quickly go over x86 stack working. The scenario is main() calls function A(), then A() calls B(). Lets see how this affects the stack. The above image is very generic in nature and calling convention is not taken into consideration.

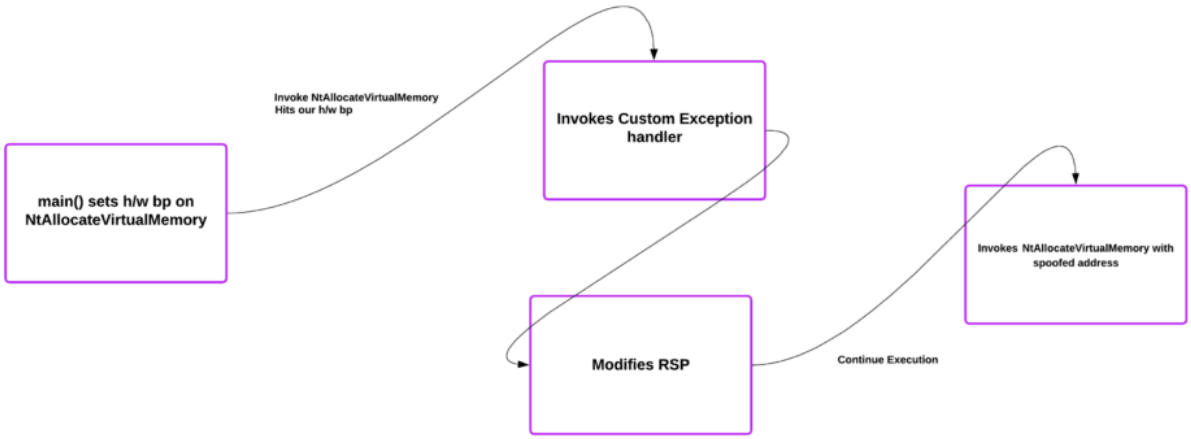
- Simple rule is state of the stack should be restored to state previous to the function call. The should be equal number of “pop” to the number of “push” instructions executed by the function to restore the state. For this reason there are many calling conventions like __cdecl and __stdcall. The windows API follow stdcall convention. Meaning “stack cleaning” or state restoration is the responsibility of the callee. Hence we see RET XX in win32 api disassembly.
- Before program executes a call instruction, it pushes the arguments (right to left) on to the stack. The first argument value will be the last item that gets pushed to the stack(again depends on calling convention, here we assume its stdcall).

- Now its time to execute the CALL instruction. Interestingly this instruction simply pushes the address of the instruction on the stack and jumps to the address of the target function by placing it in the EIP. This is how the system knows where to transfer control after finishing execution of the function. Now the function can access all of the parameter values from the stack, that are pushed by the caller.
- Whenever things get pushed to the stack, the ESP value changes, there needs to be a standard mechanism to keep track of the contents of the stack. Since ESP is very volatile in nature, we use EBP register to keep track of functions. These are called “frames”. The frames are logical boundaries set in the stack to identify a specific function. All the data within a frame belongs to a function. This way we can easily make a stack trace and see the call sequence by simply parsing the EBP chain.
- This is the main reason why all x86 functions have its signature prologue and epilogue as shown below.

```
1 push    ebp
2 mov     ebp, esp
3 mov esp, ebp
4 pop ebp
5 ret               ;ret xx
6
7
8
```

- Right after the call instruction the ESP will be pointing to the return address, the first thing that callee does is to execute the prologue, it pushes(preserves frame pointer of caller, A()) the value in the EBP to the stack and this increments the ESP by 4 bytes and move this new address in stack to EBP. From now on this serves as the base of the stack and we don’t need ESP to keep track of the state of the stack.
- This is called EBP based referencing. Now [EBP+XX] can access parameter values from the stack and [EBP-XX] can access data local to the function.
- When function exits, it executes the epilogue which reverses the effect of the prologue. After execution of “pop” the ESP will point to our return address and ret instruction will simply put return address in the EIP and thus the control gets transferred to caller A(). The value returned by the callee B() will be present in EAX. In case of “ret xx”, the process is same as mentioned below, difference is the callee simply adds the xx value to the ESP and cleans the stack. The xx value is computed by the compiler based on the stack sized required by the function based on the its data. The stack clean up is dictated by the calling convention.

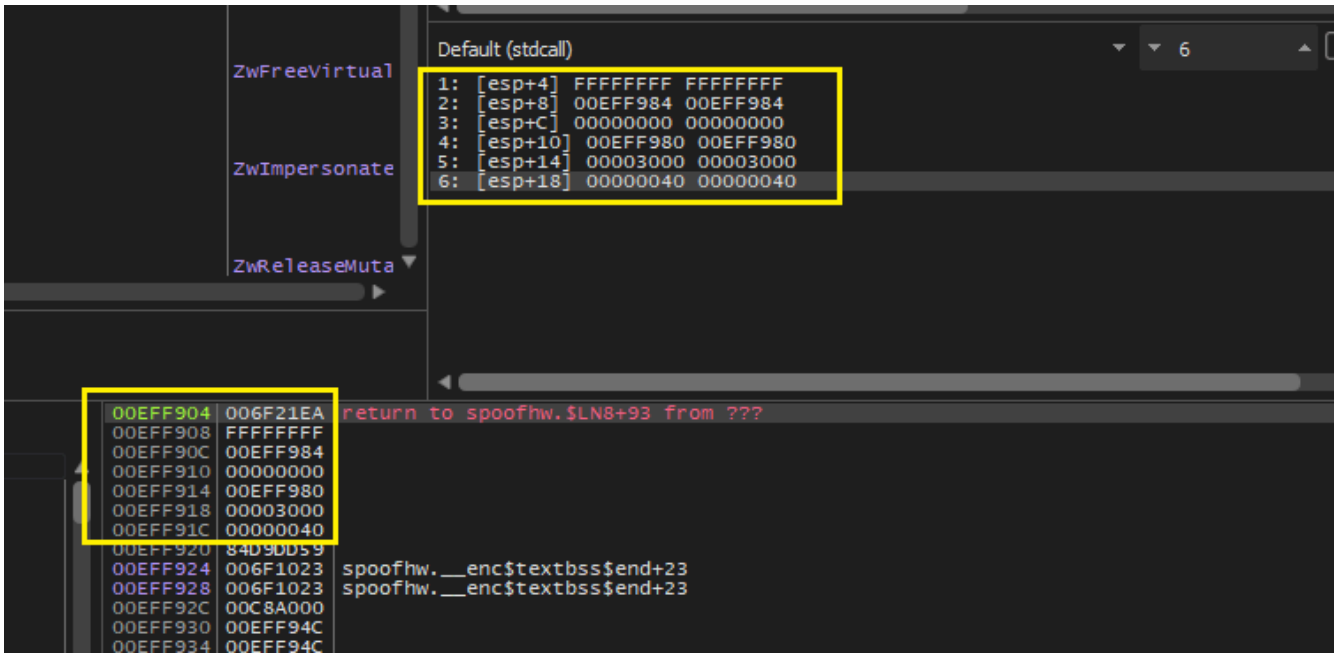
Process Overview



- We will make use of Vectored Exception Handling to define a custom exception handler to perform the stack manipulation.
- We need a trigger to invoke our handler, which is going to be a hardware based breakpoint set on function we want to execute. Lets target assembly stub of NtAllocateVirtualMemory in ntdll.dll module.
- Following exception handling, the execution will resume normally but NtAllocateVirtualMemory gets executed with a spoofed address.

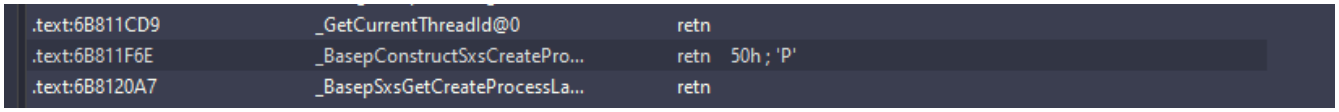
Spoofing return address

Below image shows the state of the stack when NtAllocateVirtualMemory gets directly called from the main(). We can see the arguments passed to the API and the the return address stored in the ESP.

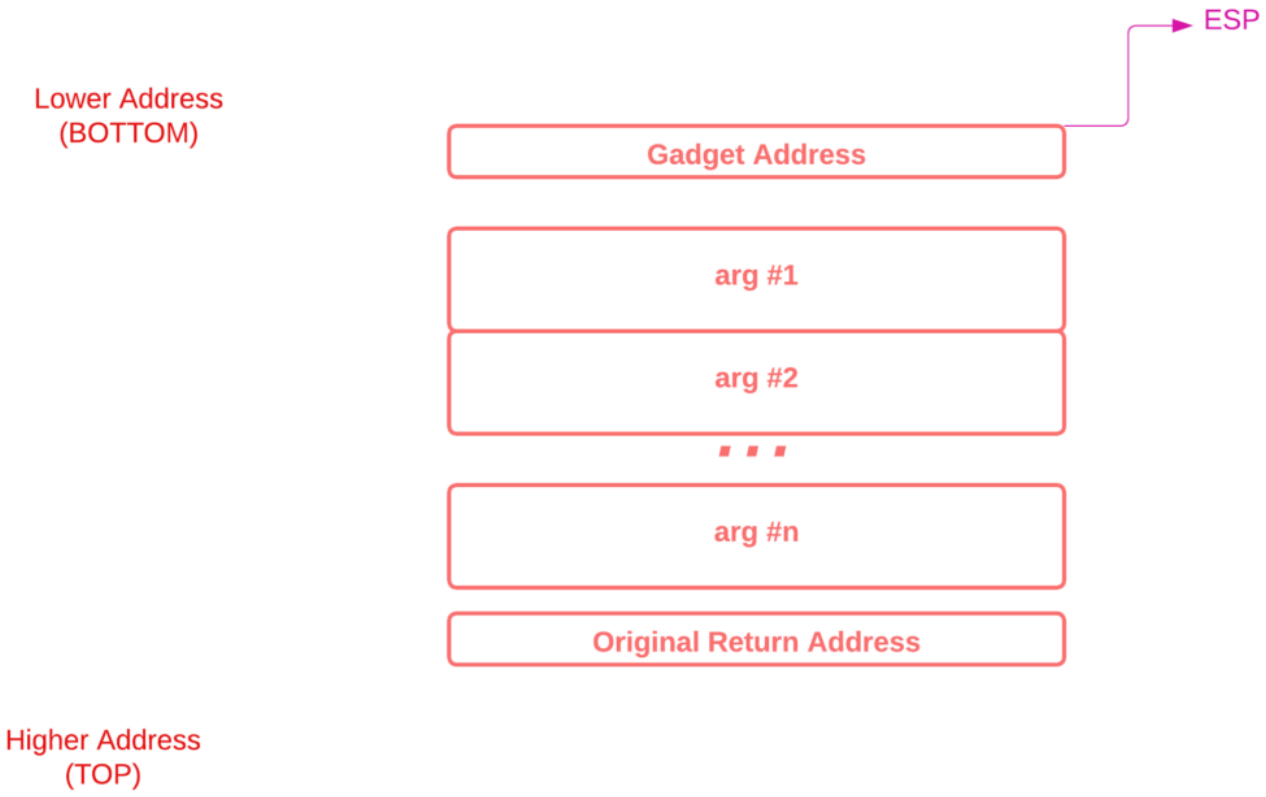


Its time to do some stack manipulation, we will modify the items on the stack from our exception handler.

First find a gadget, lets hunt for a simple “ret” in kernel32.dll.

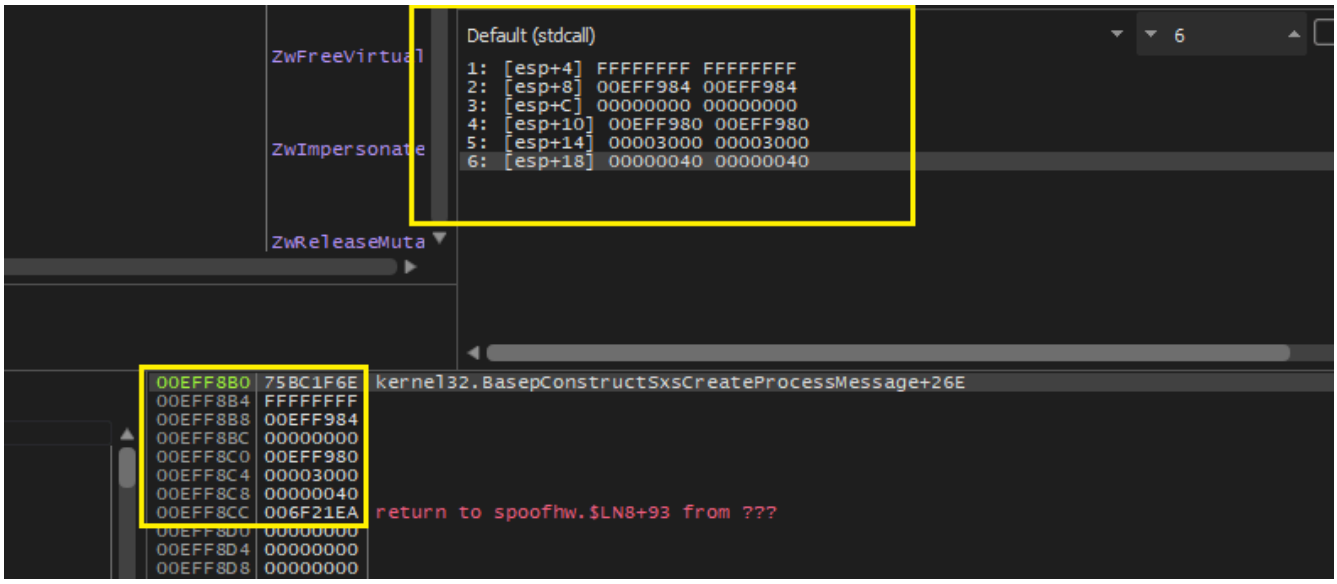


- There is no specific reason but I am just going for “ret 50” as shown above.
- For this to work, we need to allocate enough space on the stack which is in our case 0x50 .
- When we hit H/W BP on the NtAllocateVirtualMemory assembly stub, the state of the stack is that the esp points to the original return address which is our main(). The system registers are accessible through the context structure passed to our handler routine.
- To spoof the return address, we need to setup the stack as shown below.

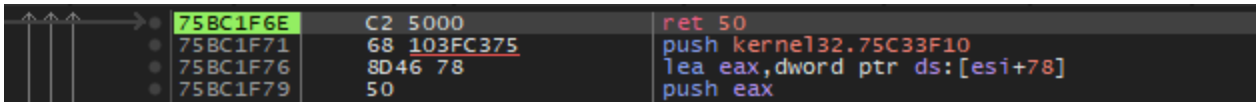


Lets do the spoofing!

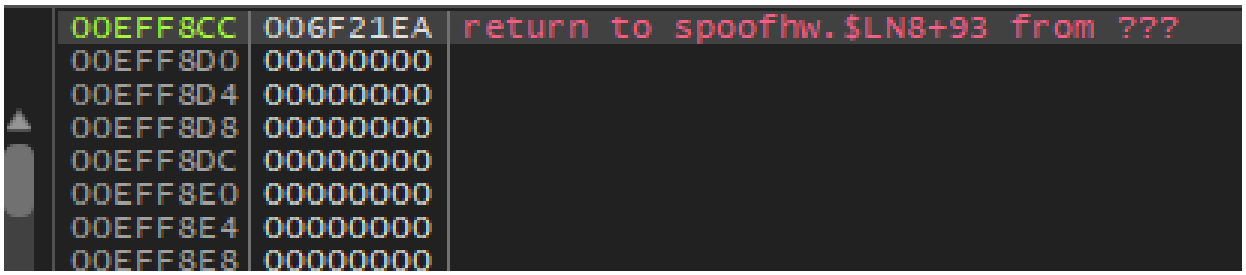
- Increase the stack size by 50, by subtracting the ESP value by 0x50.
- Copy the stack items over to new stack.
- Place the original return address to main() at exact position shown in the image above following the api arguments.
- Finally we can replace the ESP value which is the top with address of our gadget. Now the stack looks exactly as the image shown above. The modified state of the stack is shown below.



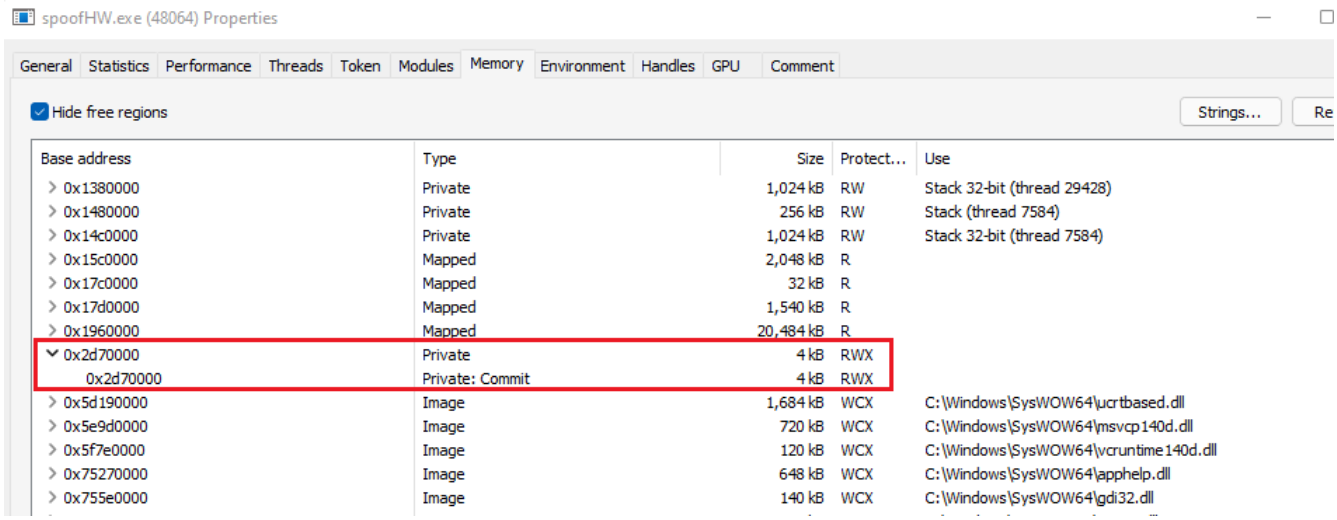
Following the execution of NtAllocateVirtualMemory, the control goes to our gadget as shown below.



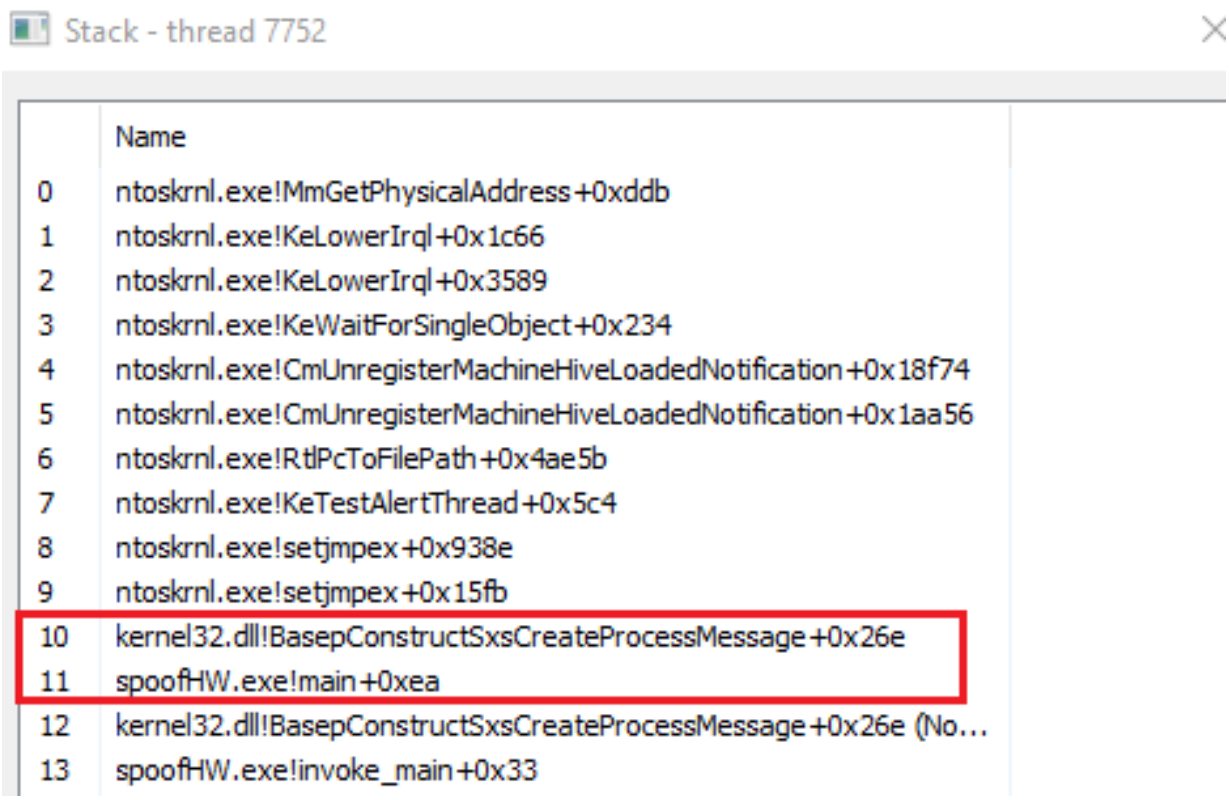
And finally back to our main() as shown below :).



The call to the api was a success as we can see the newly allocated RWX region as shown below.



And here is the stack trace for the main thread!



Handler Code

```
#define STACK_ARGS_LENGTH 5 //total number of args - 1
LONG HWSpoofHandler(EXCEPTION_POINTERS* ExceptionInfo)
{


if (ExceptionInfo->ExceptionRecord->ExceptionCode ==
EXCEPTION_SINGLE_STEP)
{
if (ExceptionInfo->ContextRecord->Eip == (DWORD64)&BreakOnSyscall)
{


ntSyscallStub = CustomGetProc((UINT64)hNtdll, (char*)*(DWORD*)
(ExceptionInfo->ContextRecord->Esp + 4));
ExceptionInfo->ContextRecord->Dr0 = ntSyscallStub ;

}
else if (ExceptionInfo->ContextRecord->Eip == (DWORD64)ntSyscallStub )
{


ExceptionInfo->ContextRecord->Esp -= 0x50;



for (size_t i = 0; i < STACK_ARGS_LENGTH; i++)
{
const size_t offset = i * STACK_ARGS_LENGTH;
*(DWORD64*)(ExceptionInfo->ContextRecord->Esp + offset) = *(DWORD64*)
(ExceptionInfo->ContextRecord->Esp + offset + 0x50);

}
*(DWORD*)(ExceptionInfo->ContextRecord->Esp + (4 * STACK_ARGS_LENGTH +
8)) = *(DWORD*)(ExceptionInfo->ContextRecord->Esp);
*(DWORD*)(ExceptionInfo->ContextRecord->Esp) = retGadgetAddress;

```

```
37  ExceptionInfo->ContextRecord->Eip = ntSyscallStub ;
38
39  ExceptionInfo->ContextRecord->Dr0 = (UINT64)&BreakOnSyscall;
40  }
41  return EXCEPTION_CONTINUE_EXECUTION;
42
43  }
44
45  return EXCEPTION_CONTINUE_SEARCH;
46
47  }
48
49
50
51
52
53
54
```

Defender doesnt care!

A classic vanilla process injection flies under the radar of defender with this vector.

