Lab 10-1

1. **Write a hello world driver in C for Windows 7 32-bit. The driver should simply print “Hello world” to Dbgview. You can either use DriverLauncher to load the driver into the kernel, or write your own binary to load the driver.**

See attached code.

2. **What is the Processor Control Block (PRCB/PCB)? List each field of the PRCB.**

The Process Control Block is a data structure in the kernel that contains the information the CPU needs to manage a process. In Windows this is stored in something called a KPROCESS structure. The fields are as follows

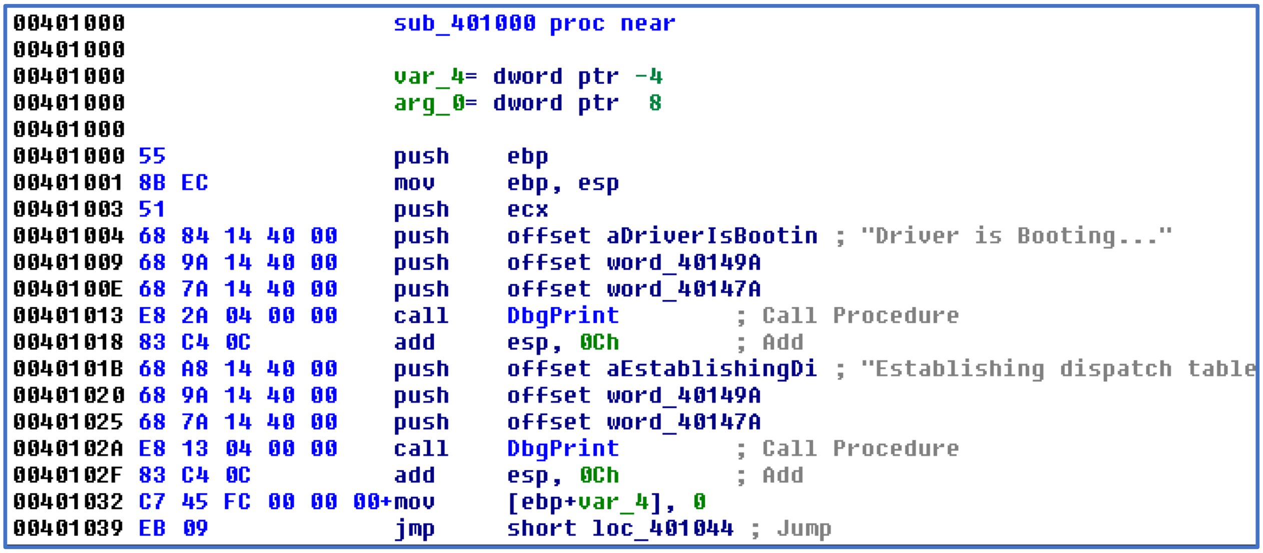
```
DISPATCHER_HEADER Header;
LIST_ENTRY ProfileListHead;
ULONG DirectoryTableBase;
ULONG Unused0;
    KGDTENTRY LdtDescriptor;
    KIDTENTRY Int21Descriptor;
    WORD IopmOffset;
    UCHAR Iopl;
    UCHAR Unused;
    ULONG ActiveProcessors;
    ULONG KernelTime;
    ULONG UserTime;
    LIST_ENTRY ReadyListHead;
    SINGLE_LIST_ENTRY SwapListEntry;
    PVOID VdmTrapcHandler;
    LIST_ENTRY ThreadListHead;
    ULONG ProcessLock;
    ULONG Affinity;
    union
    {
        ULONG AutoAlignment: 1;
        ULONG DisableBoost: 1;
        ULONG DisableQuantum: 1;
        ULONG ReservedFlags: 29;
        LONG ProcessFlags;
    };
    CHAR BasePriority;
    CHAR QuantumReset;
    UCHAR State;
    UCHAR ThreadSeed;
    UCHAR PowerState;
    UCHAR IdealNode;
    UCHAR Visited;
    union
    {
        KEXECUTE_OPTIONS Flags;
        UCHAR ExecuteOptions;
    };
    ULONG StackCount;
    LIST_ENTRY ProcessListEntry;
    UINT64 CycleTime;
```

More information can be found at https://reactos.org/wiki/Techwiki:Ntoskrnl/KPROCESS#ProfileListHead.

Lab 10-2.malware

1. What is the address of the malicious function called by DriverEntry?

The malicious function is at 0x40100.



```
00401000                         sub_401000 proc near
00401000
00401000                         var_4= dword ptr -4
00401000                         arg_0= dword ptr  8
00401000
00401000 55                      push    ebp
00401001 8B EC                   mov     ebp, esp
00401003 51                      push    ecx
00401004 68 84 14 40 00          push    offset aDriverIsBootin ; "Driver is Booting..."
00401009 68 9A 14 40 00          push    offset word_40149A
0040100E 68 7A 14 40 00          push    offset word_40147A
00401013 E8 2A 04 00 00          call    DbgPrint        ; Call Procedure
00401018 83 C4 0C                add     esp, 0Ch        ; Add
0040101B 68 A8 14 40 00          push    offset aEstablishingDi ; "Establishing dispatch table
00401020 68 9A 14 40 00          push    offset word_40149A
00401025 68 7A 14 40 00          push    offset word_40147A
0040102A E8 13 04 00 00          call    DbgPrint        ; Call Procedure
0040102F 83 C4 0C                add     esp, 0Ch        ; Add
00401032 C7 45 FC 00 00 00+mov   [ebp+var_4], 0
00401039 EB 09                   jmp     short loc_401044 ; Jump
```

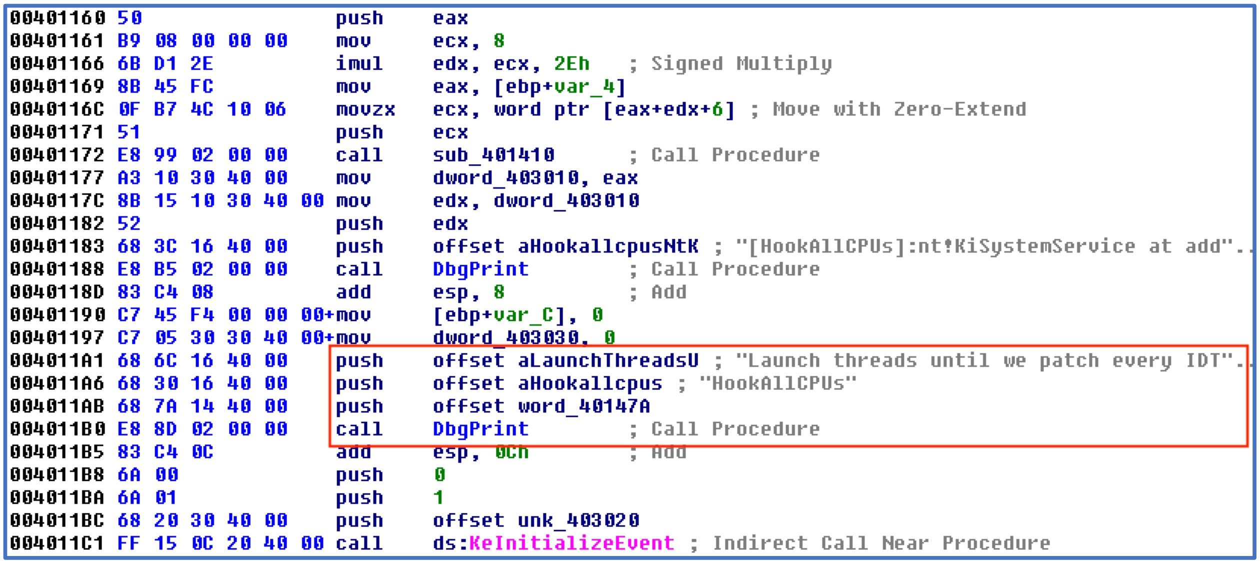
2. Describe the SIDT and LIDT instructions, what are they used for?

The SIDT instruction stores the address of the IDT in the destination operand, and the LIDT loads it from the source operand. They are used by the malware to get the location of IDT so that it can patch each one of the interrupt handlers.

http://resources.infosecinstitute.com/hooking-idt/

3. What is the malicious function doing? What is it creating?

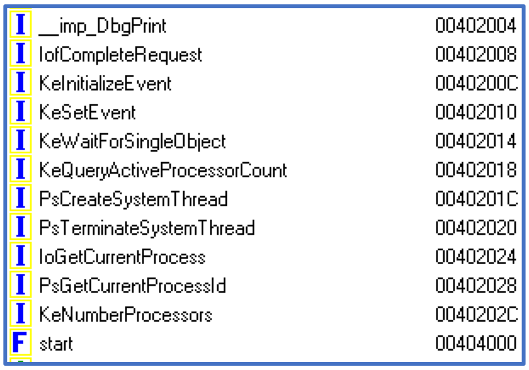
The malicious function is spawning threads to patch every entry in the IDT to point to a custom function. This function performs its own malicious payload, then passes off the execution to the actual interrupt handler.



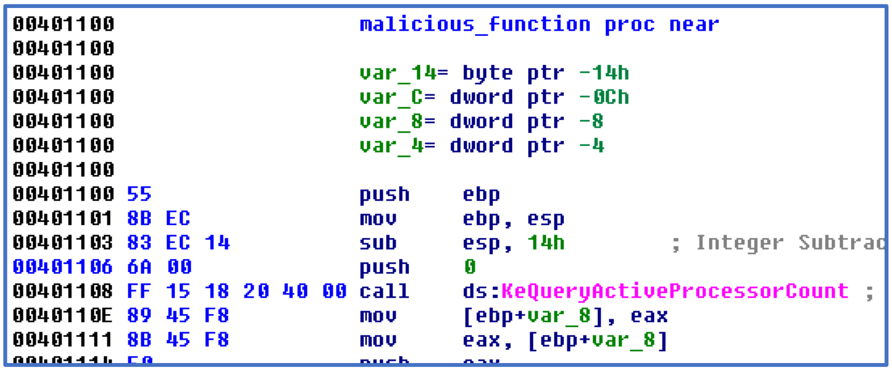
```
00401160 50                push    eax
00401161 B9 08 00 00 00    mov     ecx, 8
00401166 6B D1 2E          imul    edx, ecx, 2Eh    ; Signed Multiply
00401169 8B 45 FC          mov     eax, [ebp+var_4]
0040116C 0F B7 4C 10 06    movzx   ecx, word ptr [eax+edx+6] ; Move with Zero-Extend
00401171 51                push    ecx
00401172 E8 99 02 00 00    call    sub_401410       ; Call Procedure
00401177 A3 10 30 40 00    mov     dword_403010, eax
0040117C 8B 15 10 30 40 00 mov     edx, dword_403010
00401182 52                push    edx
00401183 68 3C 16 40 00    push    offset aHookallcpusNtK ; "[HookAllCPUs]:nt!KiSystemService at add"..
00401188 E8 B5 02 00 00    call    DbgPrint         ; Call Procedure
0040118D 83 C4 08          add     esp, 8           ; Add
00401190 C7 45 F4 00 00 00+mov     [ebp+var_C], 0
00401197 C7 05 30 30 40 00+mov     dword_403030, 0
004011A1 68 6C 16 40 00    push    offset aLaunchThreadsU ; "Launch threads until we patch every IDT"..
004011A6 68 30 16 40 00    push    offset aHookallcpus ; "HookAllCPUs"
004011AB 68 7A 14 40 00    push    offset word_40147A
004011B0 E8 8D 02 00 00    call    DbgPrint         ; Call Procedure
004011B5 83 C4 0C          add     esp, 0Ch         ; Add
004011B8 6A 00             push    0
004011BA 6A 01             push    1
004011BC 68 20 30 40 00    push    offset unk_403020
004011C1 FF 15 0C 20 40 00 call    ds:KeInitializeEvent ; Indirect Call Near Procedure
```

4. What is this sample doing?

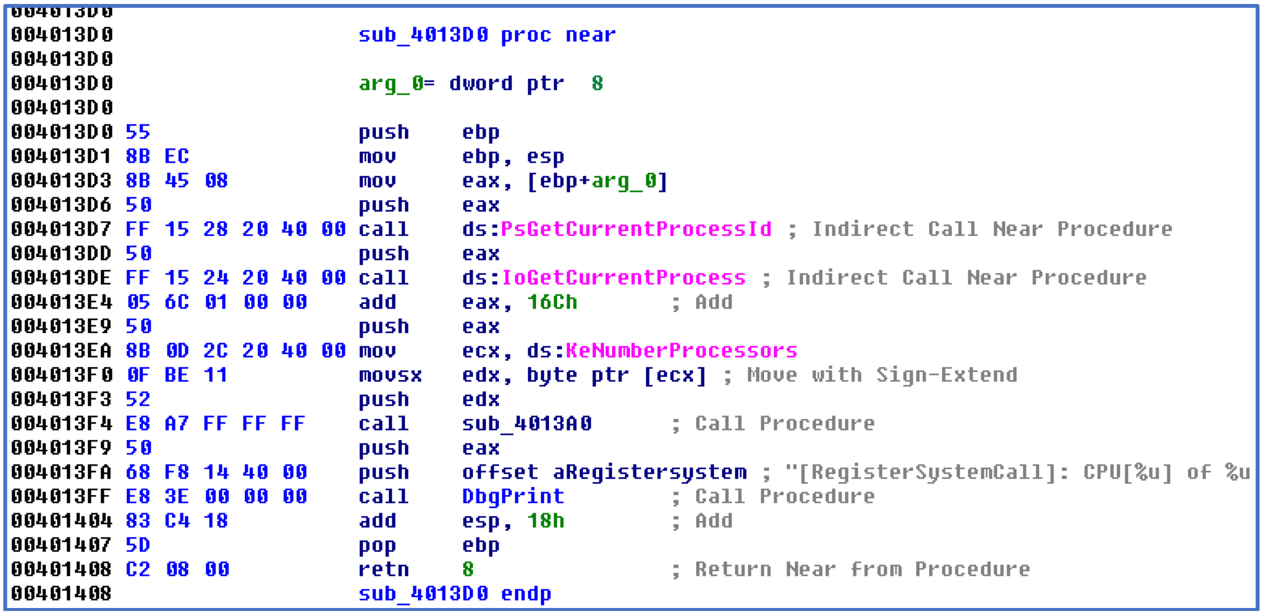
The custom function logs (to debug output) information like CPU processor count and PID to debug output. Examples of these function calls and information logging can be seen below.



I	__imp_DbgPrint	00402004
I	IofCompleteRequest	00402008
I	KeInitializeEvent	0040200C
I	KeSetEvent	00402010
I	KeWaitForSingleObject	00402014
I	KeQueryActiveProcessorCount	00402018
I	PsCreateSystemThread	0040201C
I	PsTerminateSystemThread	00402020
I	IoGetCurrentProcess	00402024
I	PsGetCurrentProcessId	00402028
I	KeNumberProcessors	0040202C
F	start	00404000



```
00401100                   malicious_function proc near
00401100
00401100                   var_14= byte ptr -14h
00401100                   var_C= dword ptr -0Ch
00401100                   var_8= dword ptr -8
00401100                   var_4= dword ptr -4
00401100
00401100 55                push    ebp
00401101 8B EC             mov     ebp, esp
00401103 83 EC 14          sub     esp, 14h         ; Integer Subtrac
00401106 6A 00             push    0
00401108 FF 15 18 20 40 00 call    ds:KeQueryActiveProcessorCount ;
0040110E 89 45 F8          mov     [ebp+var_8], eax
00401111 8B 45 F8          mov     eax, [ebp+var_8]
00401114 50                push    eax
```



```
004013D0
004013D0                      sub_4013D0 proc near
004013D0
004013D0                      arg_0= dword ptr  8
004013D0
004013D0 55                   push    ebp
004013D1 8B EC                mov     ebp, esp
004013D3 8B 45 08             mov     eax, [ebp+arg_0]
004013D6 50                   push    eax
004013D7 FF 15 28 20 40 00 call    ds:PsGetCurrentProcessId ; Indirect Call Near Procedure
004013DD 50                   push    eax
004013DE FF 15 24 20 40 00 call    ds:IoGetCurrentProcess ; Indirect Call Near Procedure
004013E4 05 6C 01 00 00       add     eax, 16Ch       ; Add
004013E9 50                   push    eax
004013EA 8B 0D 2C 20 40 00 mov     ecx, ds:KeNumberProcessors
004013F0 0F BE 11             movsx   edx, byte ptr [ecx] ; Move with Sign-Extend
004013F3 52                   push    edx
004013F4 E8 A7 FF FF FF       call    sub_4013A0      ; Call Procedure
004013F9 50                   push    eax
004013FA 68 F8 14 40 00       push    offset aRegistersystem ; "[RegisterSystemCall]: CPU[%u] of %u
004013FF E8 3E 00 00 00       call    DbgPrint        ; Call Procedure
00401404 83 C4 18             add     esp, 18h        ; Add
00401407 5D                   pop     ebp
00401408 C2 08 00             retn    8               ; Return Near from Procedure
00401408                      sub_4013D0 endp
```