

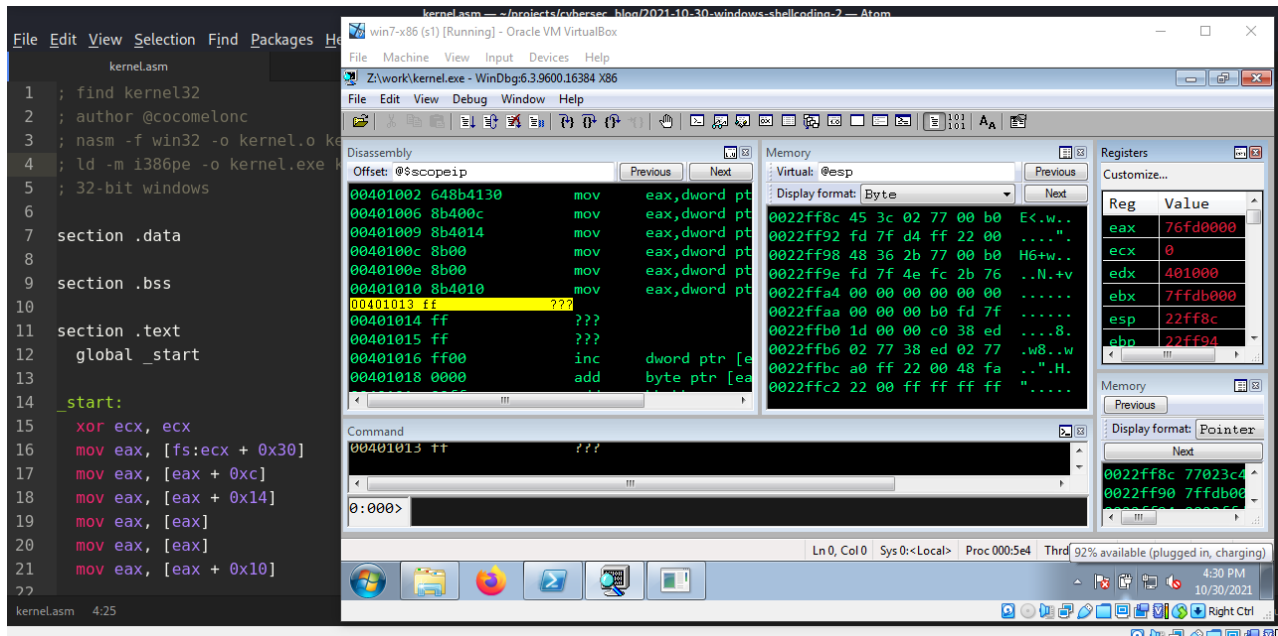
Windows shellcoding - part 2. Find kernel32 address

cocamelonc.github.io/tutorial/2021/10/30/windows-shellcoding-2.html

October 30, 2021

5 minute read

Hello, cybersecurity enthusiasts and white hackers!



In the first part of my post about windows shellcoding we found the addresses of `kerne132` and functions using the following logic:

```

/*
getaddr.c - get addresses of functions
(ExitProcess, WinExec) in memory
*/
#include <windows.h>
#include <stdio.h>

int main() {
    unsigned long Kernel32Addr;    // kernel32.dll address
    unsigned long ExitProcessAddr; // ExitProcess address
    unsigned long WinExecAddr;     // WinExec address

    Kernel32Addr = GetModuleHandle("kernel32.dll");
    printf("KERNEL32 address in memory: 0x%08p\n", Kernel32Addr);

    ExitProcessAddr = GetProcAddress(Kernel32Addr, "ExitProcess");
    printf("ExitProcess address in memory is: 0x%08p\n", ExitProcessAddr);

    WinExecAddr = GetProcAddress(Kernel32Addr, "WinExec");
    printf("WinExec address in memory is: 0x%08p\n", WinExecAddr);

    getchar();
    return 0;
}

```

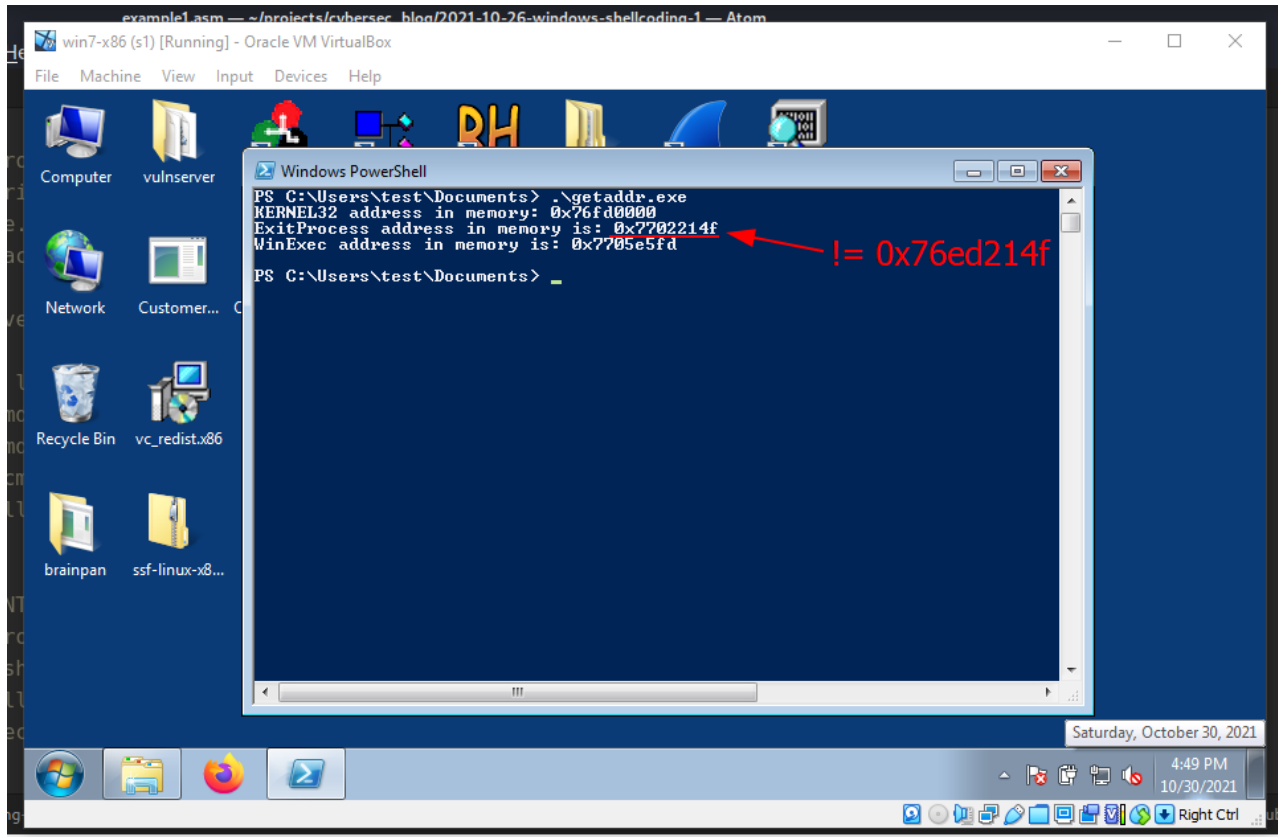
Then we entered the found address into our shellcode:

```

; void ExitProcess([in] UINT uExitCode);
xor  eax, eax          ; zero out eax
push eax              ; push NULL
mov  eax, 0x76ed214f   ; call ExitProcess function addr in kernel32.dll
jmp  eax              ; execute the ExitProcess function

```

The caveat is that the addresses of all DLLs and their functions change upon reboot and differ in each system. For this reason, we cannot hard-code any addresses in our ASM code:



First of all, how do we find the address of `kernel32.dll`?

TEB and PEB structures

Whenever we execute any exe file, the first thing that is created (at least to my knowledge) in the OS are PEB:

```

typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PPEB_LDR_DATA Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    PVOID Reserved4[3];
    PVOID AtlThunkSListPtr;
    PVOID Reserved5;
    ULONG Reserved6;
    PVOID Reserved7;
    ULONG Reserved8;
    ULONG AtlThunkSListPtr32;
    PVOID Reserved9[45];
    BYTE Reserved10[96];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE Reserved11[128];
    PVOID Reserved12[1];
    ULONG SessionId;
} PEB, *PPEB;

```

and TEB:

```

typedef struct _TEB {
    PVOID Reserved1[12];
    PPEB ProcessEnvironmentBlock;
    PVOID Reserved2[399];
    BYTE Reserved3[1952];
    PVOID TlsSlots[64];
    BYTE Reserved4[8];
    PVOID Reserved5[26];
    PVOID ReservedForOle;
    PVOID Reserved6[4];
    PVOID TlsExpansionSlots;
} TEB, *PTEB;

```

PEB - process structure in windows, filled in by the loader at the stage of process creation, which contains the information necessary for the functioning of the process.

TEB is a structure that is used to store information about threads in the current process, each thread has its own TEB.

Let's open some program in the windbg debugger and run command:

```
dt _teb
```

```
Command
77210541 cc int 5
0:000> dt _teb
ntdll!_TEB
+0x000 NtTib : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId : _CLIENT_ID
+0x028 ActiveRpcHandle : Ptr32 Void
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
+0x034 LastErrorValue : Uint4B
```

As we can see, PEB has an offset of 0x030. Similarly, we can see the contents of the PEB structure using command:

dt _peb

```
Command
0:000> dt _peb
ntdll!_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 BitField : UChar
+0x003 ImageUsesLargePages : Pos 0, 1 Bit
+0x003 IsProtectedProcess : Pos 1, 1 Bit
+0x003 IsLegacyProcess : Pos 2, 1 Bit
+0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
+0x003 SkipPatchingUser32Forwarders : Pos 4, 1 Bit
+0x003 SpareBits : Pos 5, 3 Bits
+0x004 Mutant : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 RTL_USER_PROCESS_PARAMETERS
0:000> dt _peb
```

We now need to look at the member that is at an offset of 0x00c from the base of the PEB structure, which is the PEB_LDR_DATA. PEB_LDR_DATA contains information about the loaded modules for the process.

Then, we can also examine PEB_LDR_DATA structure via windbg:

dt _PEB_LDR_DATA

```
Command
+0x240 TracingFlags : Uint4B
+0x240 HeapTracingEnabled : Pos 0, 1 Bit
+0x240 CritSecTracingEnabled : Pos 1, 1 Bit
+0x240 SpareTracingBits : Pos 2, 30 Bits
0:000> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
+0x000 Length : Uint4B
+0x004 Initialized : UChar
+0x008 SsHandle : Ptr32 Void
+0x00c InLoadOrderModuleList : _LIST_ENTRY
+0x014 InMemoryOrderModuleList : _LIST_ENTRY
+0x01c InInitializationOrderModuleList : _LIST_ENTRY
+0x024 EntryInProgress : Ptr32 Void
+0x028 ShutdownInProgress : UChar
+0x02c ShutdownThreadId : Ptr32 Void
0:000>
```

Here we can see that the offset of `InLoadOrderModuleList` is `0x00c`, `InMemoryOrderModuleList` is `0x014`, and `InInitializationOrderModuleList` is `0x01c`. `InMemoryOrderModuleList` is a doubly linked list where each list item points to an `LDR_DATA_TABLE_ENTRY` structure, so Windbg suggests the structure type is `LIST_ENTRY`.

Before we continue let's run the command:

```
!peb
```

```
Command
0:000> !peb
PEB at 7ffdf000
  InheritedAddressSpace: No
  ReadImageFileExecOptions: No
  BeingDebugged: Yes
  ImageBaseAddress: 00400000
  Ldr: 77328880
  Ldr.Initialized: Yes
  Ldr.InInitializationOrderModuleList: 00511a90 . 00512868
  Ldr.InLoadOrderModuleList: 005119f0 . 00512858
  Ldr.InMemoryOrderModuleList: 005119f8 . 00512860
  Base TimeStamp      Module
  400000 617955e8 Oct 27 19:36:40 2021 Z:\work\exit.exe
  77250000 57c99842 Sep 02 21:18:26 2016 C:\Windows\SYSTEM32\ntdll.dll
  76fd0000 4ce7b8ef Nov 20 18:02:55 2010 C:\Windows\system32\kernel32.dll
0:000>
```

As we can see, LDR (PEB structure) address is - `77328880`.

Now to see the addresses of the `InLoadOrderModuleList`, `InMemoryOrderModuleList` and `InInitializationOrderModuleList` run the command:

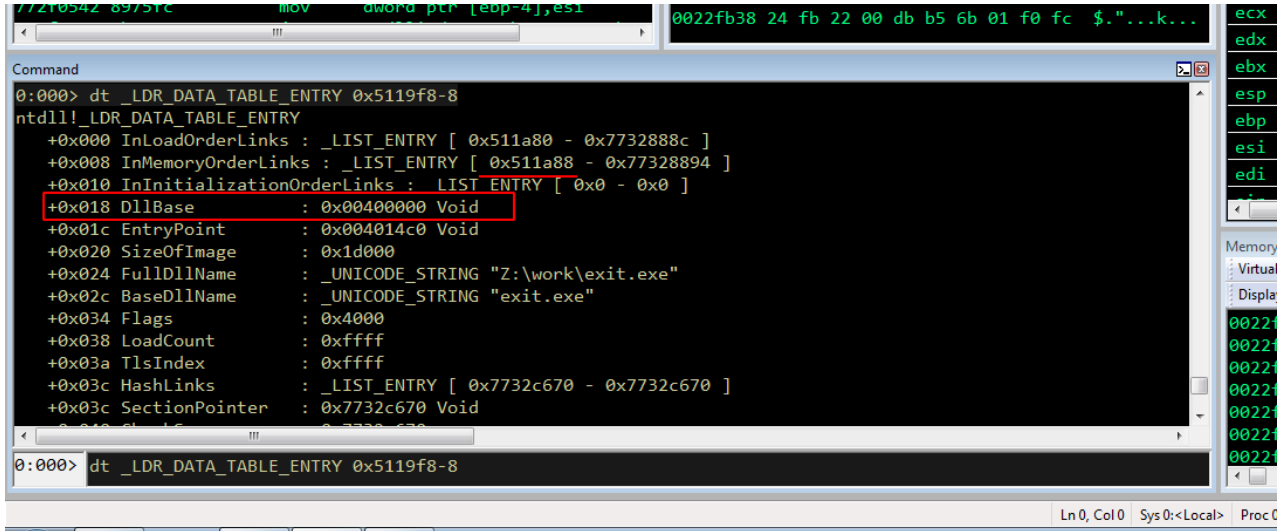
```
dt _PEB_LDR_DATA 77328880
```

This will show us the corresponding start addresses and end addresses of linked lists:

```
Command
windir=C:\Windows
windows_tracing_flags=3
windows_tracing_logfile=C:\BVTBin\Tests\installpackage\csilogfile.log
0:000> dt _PEB_LDR_DATA 77328880
ntdll!_PEB_LDR_DATA
+0x000 Length : 0x30
+0x004 Initialized : 0x1 ''
+0x008 SsHandle : (null)
+0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x5119f0 - 0x512858 ]
+0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x5119f8 - 0x512860 ]
+0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x511a90 - 0x512868 ]
+0x024 EntryInProgress : (null)
+0x028 ShutdownInProgress : 0 ''
+0x02c ShutdownThreadId : (null)
0:000>
```

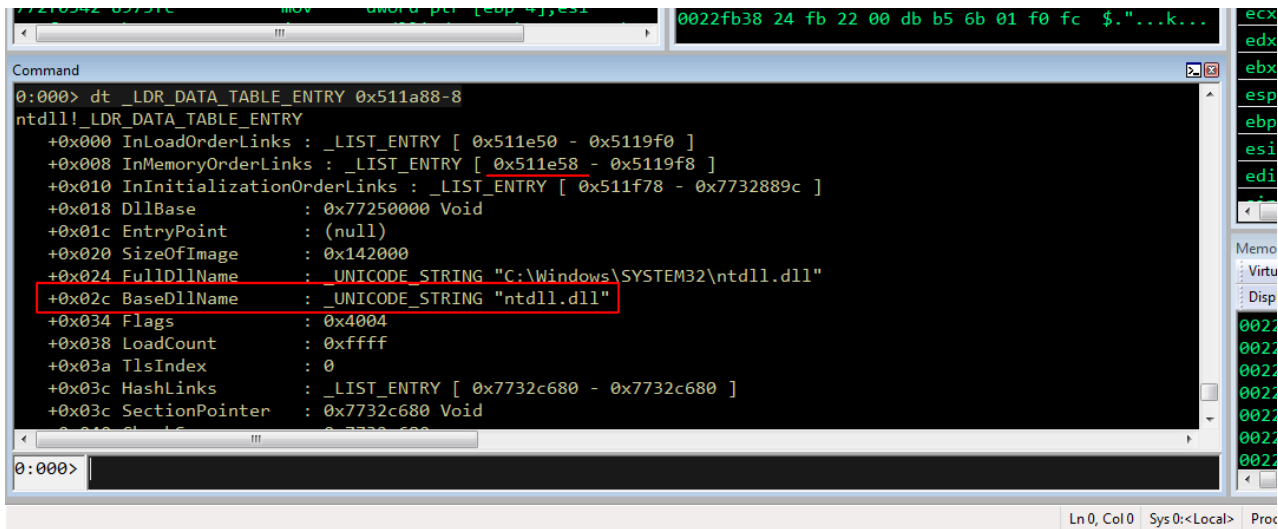
Let's try to view the modules loaded into the `LDR_DATA_TABLE_ENTRY` structure, and we will also indicate the starting address of this structure at `0x5119f8` so that we can see the base addresses of the loaded modules. Remember that `0x5119f8` is the address of this structure, so the first entry will be 8 bytes less than this address:

```
dt _LDR_DATA_TABLE_ENTRY 0x5119f8-8
```



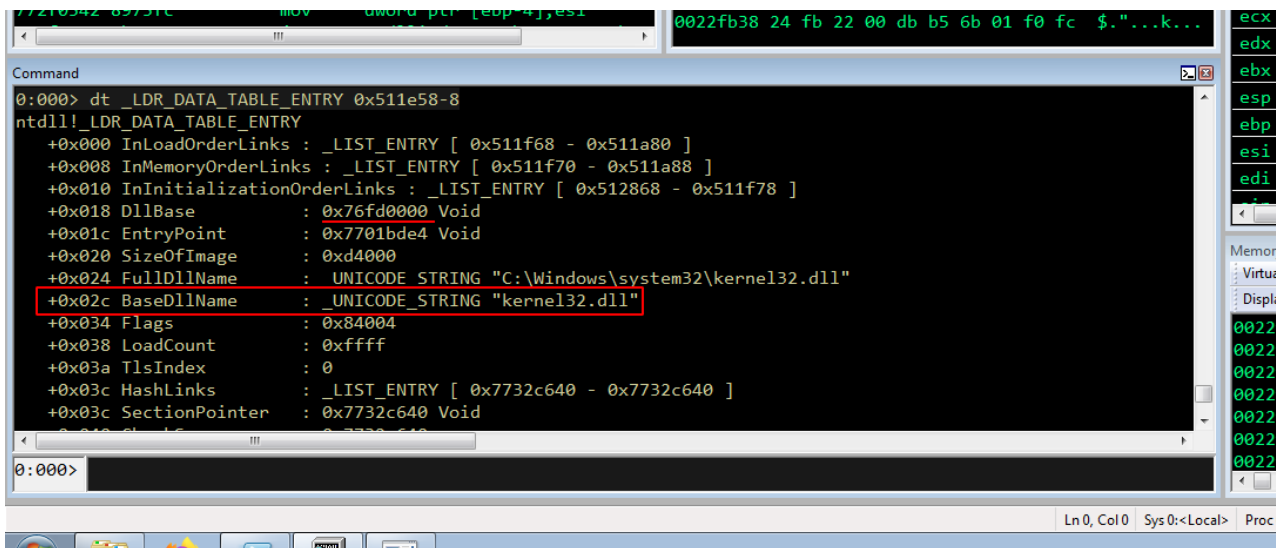
As you can see `BaseDllName` is our `exit.exe`. This is exe I executed. Also, you can see that the `InMemoryOrderLinks` address is now `0x511a88`. `DllBase` at offset `0x018` contains the base address `BaseDllName`. Now our next loaded module should be 8 bytes away from `0x511a88`, namely `0x5119f8-8`:

```
dt _LDR_DATA_TABLE_ENTRY 0x5119f8-8
```

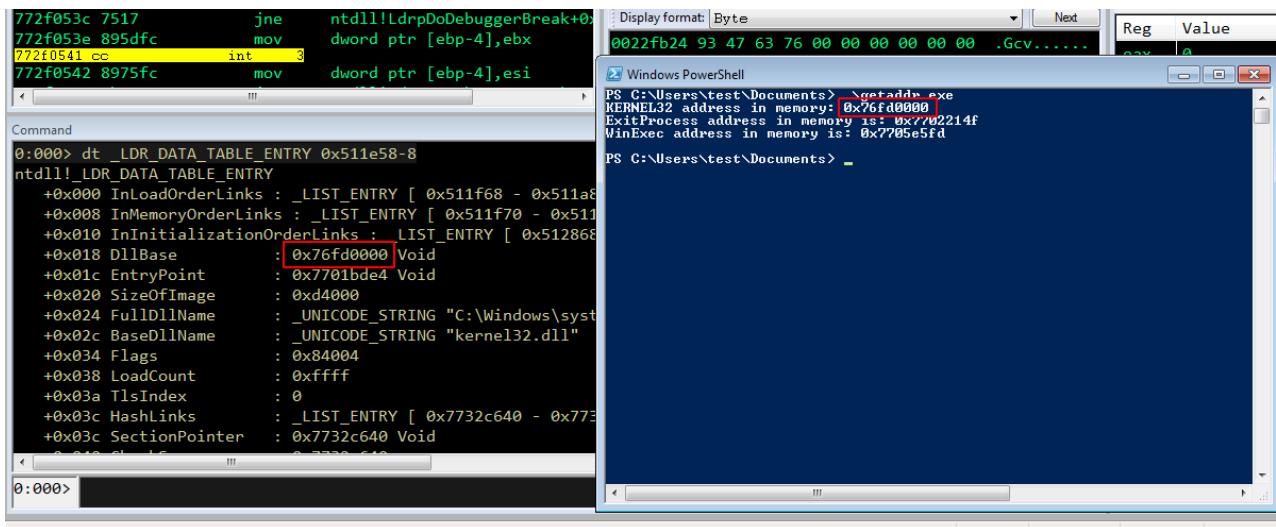


As you can see `BaseDllName` is `ntdll.dll`. It's address is `0x77250000` and the next module is 8 bytes after `0x511e58`. So, then:

```
dt _LDR_DATA_TABLE_ENTRY 0x511e58-8
```



As you can see our third module is `kernel32.dll` and its address is `0x76fd0000`, offset is `0x018`. To make sure that it is correct, we can run our `getaddr.exe`:



This module loading order will always be fixed (at least to my knowledge) for Windows 10, 7. So when we write in ASM, we can go through the entire PEB LDR structure and find the `kernel32.dll` address and load it into our shellcode.

As I wrote in the [first part](#), The next module should be `kernelbase.dll`. Just for experiment, to make sure that it is correct, we can run:

```
dt _LDR_DATA_TABLE_ENTRY 0x511f70-8
```



```

Command
0:000> dt _LDR_DATA_TABLE_ENTRY 0x511f70-8
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x512858 - 0x511e50 ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x512860 - 0x511e58 ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x511e60 - 0x511a90 ]
+0x018 DllBase : 0x754f0000 Void
+0x01c EntryPoint : 0x754f7de0 Void
+0x020 SizeOfImage : 0x4a000
+0x024 FullDllName : UNICODE_STRING "C:\Windows\system32\KERNELBASE.dll"
+0x02c BaseDllName : UNICODE_STRING "KERNELBASE.dll"
+0x034 Flags : 0x84004
+0x038 LoadCount : 0xffff
+0x03a TlsIndex : 0
+0x03c HashLinks : _LIST_ENTRY [ 0x7732c690 - 0x7732c690 ]
+0x03c SectionPointer : 0x7732c690 Void
0:000> dt _LDR_DATA_TABLE_ENTRY 0x511f70-8

```

Thus, the following is obtained:

1. offset to the PEB struct is 0x030
2. offset to LDR within PEB is 0x00c
3. offset to InMemoryOrderModuleList is 0x014
4. 1st loaded module is our .exe
5. 2nd loaded module is ntdll.dll
6. 3rd loaded module is kernel32.dll
7. 4th loaded module is kernelbase.dll

In all recent versions of the Windows OS (at least to my knowledge), the FS register points to the TEB. Therefore, to get the base address of our kernel32.dll (kernel.asm):

```

; find kernel32
; author @cocomelonc
; nasm -f win32 -o kernel.o kernel.asm
; ld -m i386pe -o kernel.exe kernel.o
; 32-bit windows

section .data

section .bss

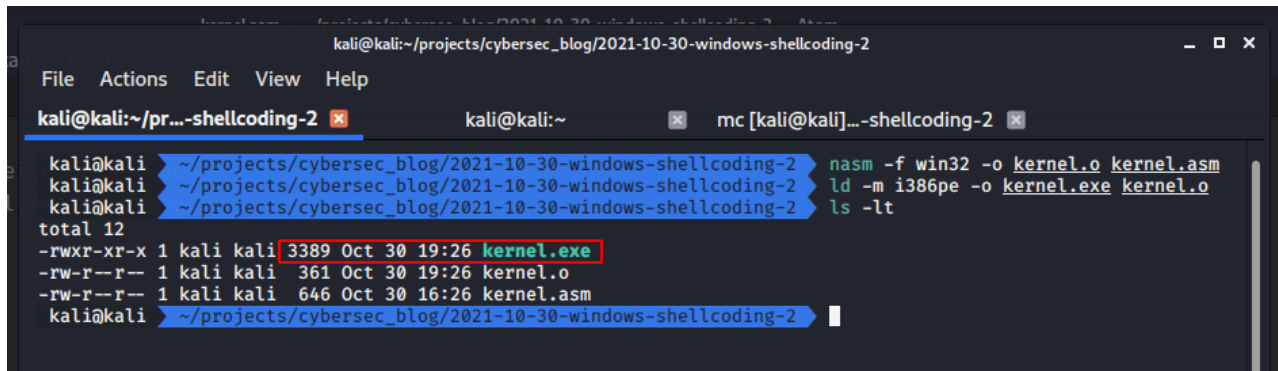
section .text
    global _start                ; must be declared for linker

_start:
    mov eax, [fs:ecx + 0x30]      ; offset to the PEB struct
    mov eax, [eax + 0xc]         ; offset to LDR within PEB
    mov eax, [eax + 0x14]        ; offset to InMemoryOrderModuleList
    mov eax, [eax]               ; kernel.exe address loaded in eax (1st module)
    mov eax, [eax]               ; ntdll.dll address loaded (2nd module)
    mov eax, [eax + 0x10]        ; kernel32.dll address loaded (3rd module)

```

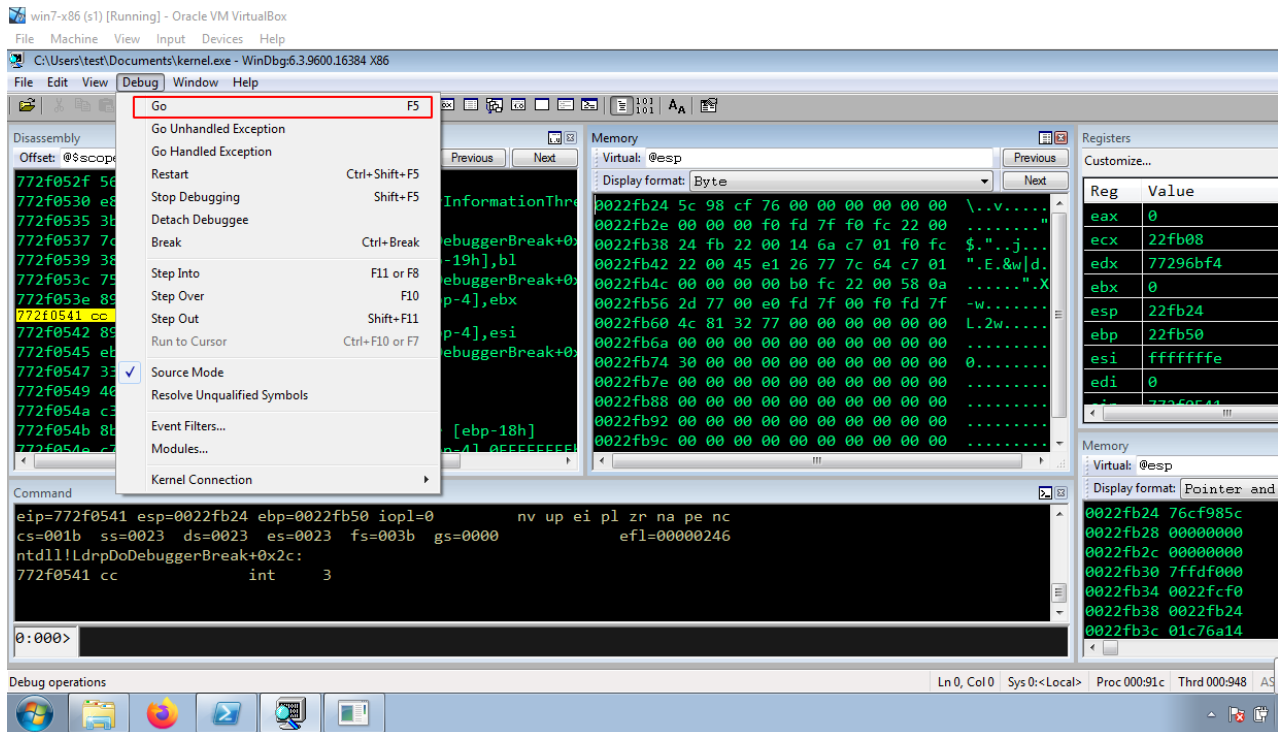
With this assembly code we can find the `kernel32.dll` address and store it in `EAX` register, so compile it:

```
nasm -f win32 -o kernel.o kernel.asm
ld -m i386pe -o kernel.exe kernel.o
```

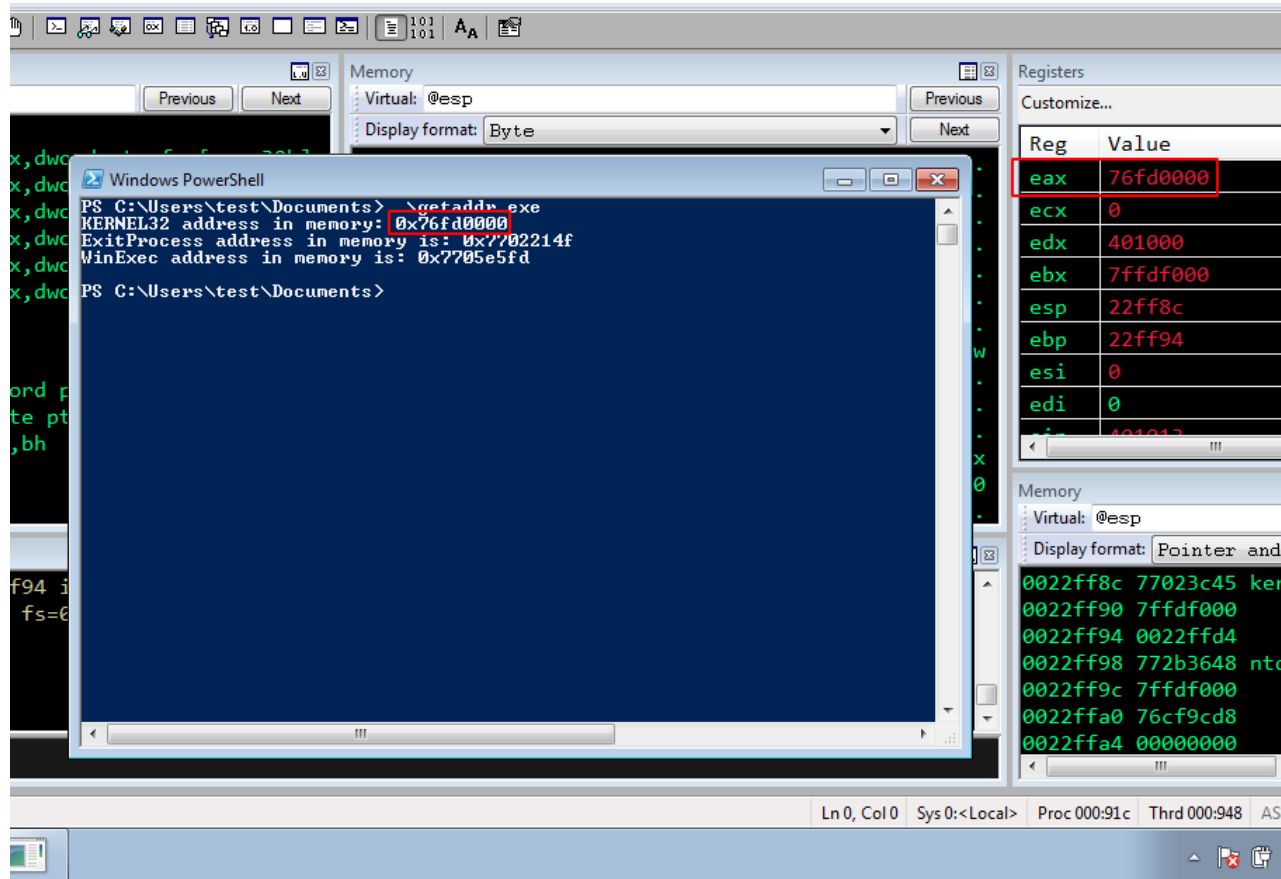


```
kali@kali:~/projects/cybersec_blog/2021-10-30-windows-shellcoding-2
File Actions Edit View Help
kali@kali:~/pr...-shellcoding-2 x kali@kali:~ x mc [kali@kali]...-shellcoding-2 x
kali@kali ~/projects/cybersec_blog/2021-10-30-windows-shellcoding-2 nasm -f win32 -o kernel.o kernel.asm
kali@kali ~/projects/cybersec_blog/2021-10-30-windows-shellcoding-2 ld -m i386pe -o kernel.exe kernel.o
kali@kali ~/projects/cybersec_blog/2021-10-30-windows-shellcoding-2 ls -lt
total 12
-rwxr-xr-x 1 kali kali 3389 Oct 30 19:26 kernel.exe
-rw-r--r-- 1 kali kali 361 Oct 30 19:26 kernel.o
-rw-r--r-- 1 kali kali 646 Oct 30 16:26 kernel.asm
kali@kali ~/projects/cybersec_blog/2021-10-30-windows-shellcoding-2
```

Copy it and run it in debugger on windows 7:



run:



As you can see everything is worked perfectly!

The next step is to find the address of function (for example `ExitProcess`) using `LoadLibraryA` and call the function. This will be in the next part.

| This is a practical case for educational purposes only.

[History and Advances in Windows Shellcode](#)

[PEB structure](#)

[TEB structure](#)

[PEB_LDR_DATA structure](#)

[The Shellcoder's Handbook](#)

[windows shellcoding_part 1](#)

[Source code in Github](#)

Thanks for your time, happy hacking and good bye!

PS. All drawings and screenshots are mine