Module 9

Defense

Rootkit detectors

Having focused on attacks, let's now also tackle protection against rootkits. This module teaches you how to detect suspicious software and its activity. We'll write a program that checks the current hooks in the IAT and checks the libraries pointed to in function addresses.

Detecting an IAT hook is really simple. If you know that GetProcAddress is contained in the kernel32 library, it becomes obvious that the function's address must fall in the range of <ImageBase; ImageBase+ImageSize>. If the address you check points to a memory area in a different library or a memory space where no library is linked, this means the function has been hooked.

First, we'll declare the global parameters that intercept a process handle, the PID of the checked process and the ImageBase of the main module. In the main function, we assign to the PID its value returned from the console as the first argument.

```
HANDLE hProc;
DWORD pid;
DWORD MainImageBase;
...
int main(int argc,char** argv)
{
pid=atoi(argv[1]);
...
```

The next step is finding the address of the program and the size of the program.

```
DWORD IB=0;
DWORD ImgSize=0;
FindImageBase(pid,&IB,&ImgSize);
printf("ImageBase: 0x%.8x\nImageSize: 0x%.8x\n\n",IB,ImgSize);
```

The `FindImageBase` function:

```
DWORD FindImageBase(DWORD pid,DWORD* IB,DWORD* ImgSize)
{
PROCESSENTRY32 lppe32;
char buf[260];
memset(buf,0,260);
 HANDLE hSnapshot;
 hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
   lppe32.dwSize = sizeof(PROCESSENTRY32);

   Process32First(hSnapshot, &lppe32);
      do
       {
                        if(lppe32.th32ProcessID==pid)
                        {
                        strcpy(buf,lppe32.szExeFile);
                        break;
                        }

      }
       while(Process32Next(hSnapshot, &lppe32));

 CloseHandle(hSnapshot);
if(buf[0]==0)
{
return 0;
}
MODULEENTRY32 mod32;
std::string x;

   hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pid);
   mod32.dwSize = sizeof(MODULEENTRY32);

    Module32First(hSnapshot, &mod32);
      do
```

```
    {
            x=mod32.szExePath;
            if(x.find(buf)!=0xFFFFFFFF)
                {
                *IB=(DWORD)mod32.modBaseAddr;
                *ImgSize=mod32.modBaseSize;
            }
    }
    while(Module32Next(hSnapshot, &mod32));
    CloseHandle(hSnapshot);
return 1;
}
```

The function looks long and complicated, but in truth it's relatively basic: it scans the process list and then the module list of a found process. The next step involves opening the process and loading the program file to the buffer.

```
hProc=OpenProcess(PROCESS_VM_READ|PROCESS_QUERY_INFORMATION,FALSE,pid);
if(hProc==0)
{
printf("can`t open process\n");
return 0;
}

char* buf=LoadMod(IB);
MainImageBase=IB;
```

The `LoadMod` function:

```
char* LoadMod(DWORD IB)
{
char ProcName[260];
GetModuleFileNameEx(hProc,(HMODULE)IB,ProcName,260);
std::ifstream f(ProcName,std::ios::binary);
char* buf;
f.seekg(0,std::ios::end);
int size=f.tellg();
f.seekg(0,std::ios::beg);
buf=(char*)malloc(size);
memset(buf,0,size);
int i=0;
char ch;
while(i<size)
{
```

```
f.get(ch);
buf[i]=ch;
i++;
}
f.close();
return buf;
}
```

Again, we are using the `PE_file` class in the code, this time however making use of a constructor that takes the address of the file buffer as an argument.

```
PE_file PE((HMODULE)buf);
IAT(&PE);
free(buf);
```

Checking for hooks takes place entirely inside the `IAT` function.

```
void IAT(PE_file* PE)
{
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;
DWORD dw;
PCHAR ptr;
DWORD IB;
char* buf;
DWORD ImgSize;
PIMAGE_OPTIONAL_HEADER32 opt;
DWORD address;
DWORD IAT_adr;
DWORD read;

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

```
            buf=LoadMod(IB);
            PE_file PE2((HMODULE)buf);
            opt=PE2.GetOptionalHeader();
            ImgSize=opt->SizeOfImage;
            free(buf);

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)
                                    {
                                    LPSTR x=MAKEINTRESOURCE(LOWORD(pthunkdatain->u1.Ordinal));

                                    address=MainImageBase+(pimportdescriptor->FirstThunk+(i*4));


                                    ReadProcessMemory(hProc,(LPCVOID)address,&IAT_adr,4,&read);


                                                if(IAT_adr<IB || IAT_adr>(IB+ImgSize))
                                                {
                                                DWORD HookBase=FindHookModule(IAT_adr);
                                                char modname[260];
                                                if(HookBase==0)
                                                {
                                                strcpy(modname,"Virtual Memory");
                                                }
                                                else
                                                {

            GetModuleFileNameEx(hProc,(HMODULE)HookBase,modname,260);
                                                }
printf("Ord: %x(%s) --- Hooked by %s(0x%.8x)\n",x,ptr,modname,IAT_adr);
                                                }
                                    } else {

pimportbyname = (PIMAGE_IMPORT_BY_NAME)(PE->RVA_to_RAW((DWORD)
```

```
                        pthunkdatain->u1.AddressOfData) + (DWORD)hInstance);
address=MainImageBase+(pimportdescriptor->FirstThunk+(i*4));

        ReadProcessMemory(hProc,(LPCVOID)address,&IAT_adr,4,&read);
                                if(IAT_adr<IB || IAT_adr>(IB+ImgSize))

                                {
                DWORD HookBase=FindHookModule(IAT_adr);
                                char modname[260];
                                if(HookBase==0)
                                {
                                strcpy(modname,"Virtual Memory");
                                }
                                else
                                {
        GetModuleFileNameEx(hProc,(HMODULE)HookBase,modname,260);
                                }

 printf("%s(%s) --- Hooked by %s(0x%.8x)\n",
(char*)pimportbyname->Name,ptr,modname,IAT_adr);
                                }
                        }
    i++;
                        pthunkdatain++;
                        pthunkdataout++;
                }

pimportdescriptor++;
}
}
```

The `IAT` function uses two other functions. One finds a module address, and the other checks which library the address points to.

```
void str_tolower(char* str)
{
int i=0;
int size=strlen(str);

while(i<size)
{
str[i]=tolower(str[i]);
i++;
}
}
```

```
DWORD FindModule(char* mod_name)
{
MODULEENTRY32 mod32;
std::string x;
HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pid);
mod32.dwSize = sizeof(MODULEENTRY32);
str_tolower(mod_name);
Module32First(hSnapshot, &mod32);
    do
    {
            str_tolower(mod32.szExePath);
            x=mod32.szExePath;

            if(x.find(mod_name)!=0xFFFFFFFF)
            {
                    return (DWORD)mod32.modBaseAddr;
            }
            }
    while(Module32Next(hSnapshot, &mod32));
    CloseHandle(hSnapshot);
            return 0;
}

DWORD FindHookModule(DWORD Address)
{
MODULEENTRY32 mod32;
HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pid);
mod32.dwSize = sizeof(MODULEENTRY32);

  Module32First(hSnapshot, &mod32);
        do
        {
            if(Address>=(DWORD)mod32.modBaseAddr &&
            Address<=(DWORD)(mod32.modBaseAddr+mod32.modBaseSize))
            {
                    return (DWORD)mod32.modBaseAddr;
            }

            }
        while(Module32Next(hSnapshot, &mod32));
        CloseHandle(hSnapshot);
        return 0;
}
```

As you can see, checking the set `hooks` is quite effortless. All it takes is comparing whether a function address belongs to the right library.

Rootkit detection features are also provided by ready-made software. One of the most popular and best detectors is `GMER`. `GMER` can reveal both user-mode and kernel-mode `hooks` and in addition is able to pick up many other suspicious behaviors and occurrences, for instance writable data sections or code modifications.

To detect malicious code, it's best to start with running an application similar to the one presented in this chapter, and use a `GMER`-like program. When you have identified an attacked application, attach a debugger and see the loaded libraries. Also, be sure to check the locations of the potentially spoofed libraries. It can turn out that even if the filename is correct, the original library has not been loaded. If you aren't yet able to pinpoint the harmful application, take a look at autoruns. Make sure you can identify all started applications. Remember no antivirus is a fully effective tool: it's just software and has repeatedly been shown as vulnerable to deception.

Usually malware attempts to set up an Internet connection. Check all applications that use the Internet through network traffic analyzers. When you get the list, see if you trust all applications on it. Also the use of a `sniffer` like `Wireshark` can produce some good results. Provided the connection is not encoded, you might be able to see if your computer receives or sends suspicious commands. When you notice strange activity, try to associate a specific transmission with a port, and then with a currently executed program.

Practice: video module transcript

Welcome to the ninth module of the training. In this module we'll learn more about defense methods. So far we've analysed the offense. We've managed to get to know the enemy, so it's high time that we focused on effective methods of countering the threats we've learned about. First, we'll deal with the detection of IAT hooking, the technique we used in the third module to hide files and registry entries.

Detecting this technique is relatively easy if we know how to start. Let's think about it for a while. If a hook is used in IAT, the function address points to a different dll library than the one it originally pointed to. We'll create a program which will detect this type of hooking and test it. We'll also learn how to track a suspicious application based on its network activity. As we can assume, the vast majority of harmful applications connect with the Internet at some point to send or receive data. We'll see how to track an application which generates suspicious network activity.

Let's move on to the demonstration. First, we'll deal with hooks in the Import Address Table, IAT for short. We can see a couple of applications here. The first one is GMER. It's a program which detects various types of hooks and modifications in the system. Another application is hidder_dll. It's a library which we wrote in the third module in order to be able to hook. We used the injector to inject the dll library to the process as well as the keylogger on which we'll test detecting suspicious network activity. There is also list file, our application for listing files which comes from the third module. First, let's start the list file program in one of the consoles. As for now, it's not relevant what the application is displaying. Our task is only to check the hooks. We have to get the process identifier.

```
cmd.exe
C:\Users\Grzonu\Desktop\Modules\9\Hooks_search\Release>tasklist | grep "list_fil
e"
list_file.exe                 4844 Console                    1      1,436 K

C:\Users\Grzonu\Desktop\Modules\9\Hooks_search\Release>
```

For this purpose, we'll use the auxiliary tasklist program. We see that the id is 4844. Now we'll check the hooks, using our own application created for the needs of this module.

```
cmd.exe
C:\Users\Grzonu\Desktop\Modules\9\Hooks_search\Release>Hooks_search.exe 4844

IAT Hook detector v. 1.0 by Grzonu

ImageBase: 0x00400000
ImageSize: 0x00011000

EncodePointer(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cca2
25)
DecodePointer(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cccc
a0)
EnterCriticalSection(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0
x77cb7720)
LeaveCriticalSection(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0
x77cb76e0)
HeapAlloc(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cc2d66)
DeleteCriticalSection(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(
0x77cc9a55)
HeapSize(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cc9b7c)
HeapReAlloc(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cdff2f
)

C:\Users\Grzonu\Desktop\Modules\9\Hooks_search\Release>
```

As we can see, the application has found a couple of hooks. However, not all of them are real. For instance, we can see that a hook was used by the ntdll library. We don't have to worry about that, it's a sort of false positive because not all functions are actually present in their libraries, some are simply redirected to others, just as the case with kernel32, which is redirected to ntdll. Now let's hook. We choose the hidder_dll library and inject it into the process.

```
cmd.exe
IAT Hook detector v. 1.0 by Grzonu

ImageBase: 0x00400000
ImageSize: 0x00011000

FindFirstFileExA(kernel32.dll) --- Hooked by C:\Users\Grzonu\Desktop\Modules\9\h
idder_dll.dll(0x68e31320)
FindNextFileA(kernel32.dll) --- Hooked by C:\Users\Grzonu\Desktop\Modules\9\hidd
er_dll.dll(0x68e31410)
EncodePointer(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cca2
25)
DecodePointer(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cccc
a0)
EnterCriticalSection(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0
x77cb7720)
LeaveCriticalSection(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0
x77cb76e0)
HeapAlloc(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cc2d66)
DeleteCriticalSection(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(
0x77cc9a55)
HeapSize(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cc9b7c)
HeapReAlloc(kernel32.dll) --- Hooked by C:\Windows\SYSTEM32\ntdll.dll(0x77cdff2f
)
```

Let's check again. We can see that this time there appeared two new hooks on the list. The first one concerns the FindFirstFileExA function, while the other, FindNextFileA. At this point we may have certain suspicions, because these functions are used to get file lists. We may assume that an application tries to hide a file on the drive. We can also see that the address points to the hidder_dll library, which isn't a standard system library. It seems our suspicions are even more grounded.

Now let's check what GMER has to tell us about that. The screen turned black for a moment. Let's refresh it. Let's scan the system right away. We see that GMER has already started scanning. First, it found hooks in SSDT. These hooks are used by the anti-virus software. The next ones are kernel hooks. Most probably they are from the anti-virus, so we shouldn't worry about them. We have to learn how to distinguish potentially dangerous entries from the normal ones that are created by the anti-virus software. Let's wait until GMER finishes the scanning. It should also show us the hook which we've used on the ZwQuerySystemInformation function, because as we remember, the task of this library is to hide a process and it hooks this particular function.

We can see that the program detected a modification in the list_file process and informs us that the ZwQuerysSystemInformation has been modified. We can also see which bytes have been modified and at which address they are. At the moment, we're searching for hooks in the import table. There appeared

some new items on the list, but they are entirely normal, used by the operating system in the GDI+ library.

We see that GMER has already finished the IAT scan and is searching other areas. Curiously, it didn't find our hooks used on the import table. It's probably caused by the fact that it considered our hook safe, or it simply uses a different method. That's one of the reasons why we shouldn't completely trust the widely available tools. As we can see, even the ones considered the best ones, such as GMER, aren't able to detect all threats. We should create our own tools and periodically test the system on our own.

Now we can stop the scanning. Let's discuss the application source code which we've used to detect hooks in IAT. As it seems, when it comes to this particular case, our program is better than GMER. Again, we are to use our class to handle PE files. The code is pretty long, but still, we'll try to discuss it. Let's start from the main function and then move on to the remaining program methods.

```
int main(int argc, CHAR* argv[])
{
printf("\n\nIAT Hook detector v. 1.0 by Grzonu\n\n");

if(argc!=2)
{
    printf("usage: %s <pid>\n\n",argv[0]);
    ProcList();
    return 0;
}
pid=atoi(argv[1]);
DWORD IB=0;
DWORD ImgSize=0;
FindImageBase(pid,&IB,&ImgSize);
printf("ImageBase: 0x%.8x\nImageSize: 0x%.8x\n\n",IB,ImgSize);

hProc=OpenProcess(PROCESS_VM_READ|PROCESS_QUERY_INFORMATION,FALSE,pid);
```

We can see that the list of all processes is displayed if no calling argument is provided. We didn't use this option and checked the process identifier using the tasklist application. The first parameter is the process number and it's changed to a variable of type int using the atoi function. Further, we have the

call of the function which searches for the ImageBase, and the size of the application the number of which we provide. We'll jump to this function. It takes the process_id as well as two pointers under which we have the relevant values. First, it gets the process list. Once it finds a process with the specific identifier, it copies the file name to the buffer. If the given process isn't found, the function returns 0 because further execution won't succeed.

```
DWORD FindImageBase(DWORD pid,DWORD* IB,DWORD* ImgSize)
{
PROCESSENTRY32 lppe32;
char buf[260];
    memset(buf,0,260);
    HANDLE hSnapshot;
   hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    lppe32.dwSize = sizeof(PROCESSENTRY32);

    Process32First(hSnapshot, &lppe32);
        do
        {
        if(lppe32.th32ProcessID==pid)
        {
        strcpy(buf,lppe32.szExeFile);
        break;
        }
```

However, if the operation is executed correctly, we get the list of modules of the process pointed to and search for the module with the name we previously copied to the buffer. Once the module is found, we set the pointed values of our two pointers to process ImageBase and image size respectively. After exiting the function, we have the base address of the main module in the IB, and the size of the main module in ImgSize.

```
pid=atoi(argv[1]);
DWORD IB=0;
DWORD ImgSize=0;
FindImageBase(pid,&IB,&ImgSize);
printf("ImageBase: 0x%.8x\nImageSize: 0x%.8x\n\n",IB,ImgSize);

hProc=OpenProcess(PROCESS_VM_READ|PROCESS_QUERY_INFORMATION,FALSE,pid);
if(hProc==0)
{
printf("can`t open process\n");
return 0;
}
char* buf=LoadMod(IB);
MainImageBase=IB;
PE_file PE((HMODULE)buf);
IAT(&PE);
free(buf);
```

In the next step, we open the process. For that purpose, we just need the reading rights and the rights to obtain the process information. If the returned value is different than 0, it means that the process was opened correctly. Now we load the module using the LoadMod function. Let's jump to it. This function takes the imagebase of the module as a parameter, but in the body it gets its name thanks to the GetModuleFileNameEx function, after which it reads the file to the buffer using the ifstream class and returns the buffer with the file contents.

In the main function, we assign IB to the global variable MainImageBase and then the buffer is loaded to the PE_file class. Further, we call the IAT function, which checks whether any hooks have been used. As a parameter, we pass an object of the PE_file class. Now let's jump to the IAT function. It's a function the base of which we've already seen multiple times. First, we get the library name. Then, using the FindModule function, we get the module ImageBase. The function gets the module list, searches for the module with the name which was passed in the parameter, and subsequently returns the ImageBase of this module.

```
int i=0;
while ( pimportdescriptor->TimeDateStamp != 0 ||pimportdescriptor->Name != 0)
{
    ptr = (PCHAR)((DWORD)hInstance+ PE->RVA_to_RAW((DWORD)pimportdescriptor->Name));
    //lib name
    i=0;
    IB=FindModule(ptr);//function return module imagebase
    buf=LoadMod(IB);//loading module
    PE_file PE2((HMODULE)buf);//module to class
    opt=PE2.GetOptionalHeader();
    ImgSize=opt->SizeOfImage;//size of module
    free(buf);

    //check imports
        pthunkdataout = (PIMAGE_THUNK_DATA)((DWORD)hInstance + PE->RVA_to_RAW((DWORD)
        if (pimportdescriptor->Characteristics == 0)
        {
            pthunkdatain = pthunkdataout;
        }
```

As we remember, the LoadMod function loads the module from the file to the buffer. Hooks modify only the memory, and not physical binaries. Hence, we'll have a reference material which might be used for detection. Next, we create a PE2 object, we assign a buffer to it, then we get an Optional Header and from it, in turn SizeOfImage, that is the size of the image in the memory. We can release the buffer now. Once we already have all the information, we can move on to look for hooks in the import table. This time we can't skip the imports via the ordinal. We create a variable x and assign to it the result of the operation of the MAKEINTRESOURCE macro. The macro will convert the value present in IAT to the relevant variable.

We calculate the address at which the function address is located. We perform it by adding the FirstThunk field from the pimportdescriptor structure to the program ImageBase, and next adding the function number on the list multiplied by the size of the int type, that is 4 bytes. The next step is reading the memory from this address so as to get the address of the function located in the IAT table. We read it to the IAT_adr variable.
The next step is checking whether the address which is present in IAT is within the range spanning from the ImageBase of the module it should point to, to the ImageBase + ImageSize. In other words, we check whether the address points to the right module. If, for instance, the function should be imported from kernel32, we check whether its address points to the memory occupied by

the kernel32 module. If the address points outside this range, we may assume that a hook was used.

```
DWORD FindHookModule(DWORD Address)
{
MODULEENTRY32 mod32;
HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, pid);
//module list
mod32.dwSize = sizeof(MODULEENTRY32);

    Module32First(hSnapshot, &mod32);
         do
        {
        if(Address>=(DWORD)mod32.modBaseAddr && Address<=(DWORD)(mod32.modBaseAddr+mod32
        {
return (DWORD)mod32.modBaseAddr;
        }

        }
        while(Module32Next(hSnapshot, &mod32));
        CloseHandle(hSnapshot);
        return 0;
```

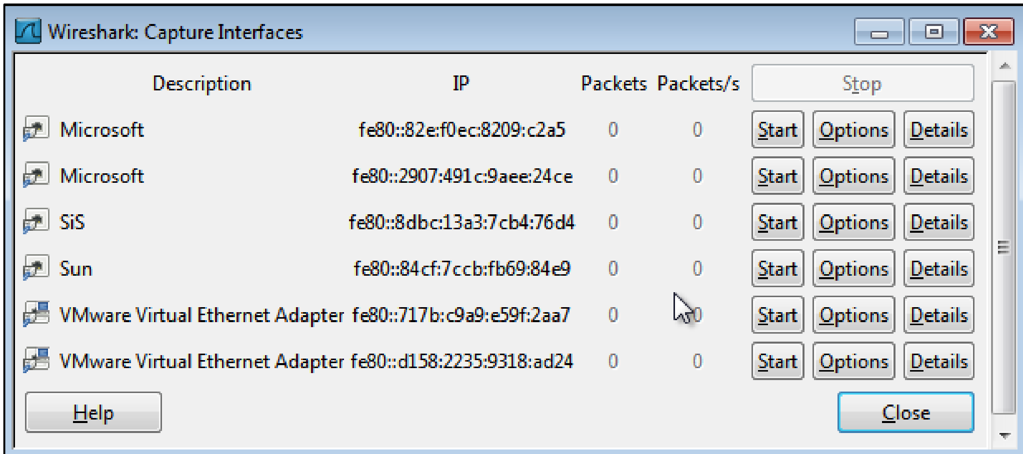
If the condition was fulfilled, we have to get to know which module the address points to. For this purpose, we'll use the FindHookModule function. Let's get acquainted with it. We see that it takes a parameter with the address. The function gets the list of modules and compares the passed address with the memory occupied by other modules. If the address is within the range from the module ImageBase to ImageBase+ImageSize, it means that it belongs to this module and most probably it's the module that hooked.

```
                    //calculate pointer to function
                    ReadProcessMemory(hProc,(LPCVOID)address,&IAT_adr,4,&read);
                    //read address of function from process
                    if(IAT_adr<IB || IAT_adr>(IB+ImgSize))
                    //check
                    {
                    DWORD HookBase=FindHookModule(IAT_adr);
                    char modname[260];
                    if(HookBase==0)
                    {
                    strcpy(modname,"Virtual Memory");
                    }
                    else
                    {
                    GetModuleFileNameEx(hProc,(HMODULE)HookBase,modname,260);
                    }
printf("Ord: %x(%s) --- Hooked by %s(0x%.8x)\n",x,ptr,modname,IAT_adr);
                    }
```

If the address doesn't match any module, it means that the function code is present in the memory allocated by VirtualAlloc and we can't determine which module used the hook. In such a situation, the function returns 0. Let's return again to our IAT function. If the FindHookModule function returns 0, we copy the Virtual Memory string to the modname variable. We know that the function is hooked, but the address doesn't belong to any of the loaded modules. However, if an address was returned, we get the module name using the GetModuleFileNameEx function, and print the results on the screen.

Here, we're analysing the case of an ordinal, but in the case of an import via the name, the procedure looks almost identical. The difference lies in the method of getting the function name. In such a situation, instead of a number, we would simply have a character string. We see that both the code and the method are relatively simple, because they are based on a simple comparison whether the address in IAT points to the relevant module.

That's basically all about detecting hooks. Now let's move on to check potentially dangerous network connections. For this purpose, we'll use a free application named Wireshark, that is a popular network sniffer, which allows us to see what packets are sent and received from the Internet.



We choose the web interface which we'll be listening on. Now we can launch our keylogger and check what our sniffer will show us. We have to wait for the packets to appear. We see that a broadcast packet appeared. We remember that the keylogger sends one log per minute. Let's write something using the keyboard, so that the program has something to send. We see that many packets appeared, which means that our keylogger has just sent a log.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.0.103	192.168.0.255	BROWSER	Domain/Workgroup Announcement WORKG
2	32.401681	Azurewav_e7:c4:8a	Broadcast	ARP	Who has 192.168.0.1? Tell 192.168.
3	32.405173	D-LinkIn_ac:21:32	Azurewav_e7:c4:8a	ARP	192.168.0.1 is at 1c:bd:b9:ac:21:32
4	32.405369	192.168.0.103	178.63.220.15	TCP	cma > http [SYN] Seq=0 Win=8192 Len
5	35.402598	192.168.0.103	178.63.220.15	TCP	cma > http [SYN] Seq=0 Win=8192 Len
6	35.437962	178.63.220.15	192.168.0.103	TCP	http > cma [SYN, ACK] Seq=0 Ack=1 W
7	35.438130	192.168.0.103	178.63.220.15	TCP	cma > http [ACK] Seq=1 Ack=1 Win=17
8	35.441118	192.168.0.103	178.63.220.15	HTTP	POST /keylogger/recv.php HTTP/1.1 [
9	35.741407	192.168.0.103	178.63.220.15	HTTP	[TCP Retransmission] POST /keylogge
10	35.774534	178.63.220.15	192.168.0.103	TCP	http > cma [ACK] Seq=1 Ack=544 Win=
11	35.775496	178.63.220.15	192.168.0.103	HTTP	HTTP/1.1 200 OK
12	35.795711	192.168.0.103	178.63.220.15	TCP	cma > http [RST, ACK] Seq=544 Ack=1
13	36.051745	192.168.0.103	178.63.220.15	TCP	optima-vnet > http [SYN] Seq=0 Win=

Let's browse these packets. We see that the address 192.168.0.103 is a typical address in the local network and it's the address of our computer. The packet was sent to the address 178.63.220.15, that is some external server. However, we don't see which application is the packet sender. Of course, we can inspect such a packet. We can see that there is an error there, but it's not an issue, because the server deals well with such situations. Now, let's see how to track the program which sent the packet. For this purpose, we need a console started with admin rights.

Let's use the standard netstat tool. We'll get acquainted with its help. It includes the description of all its parameters. We'll use the "b" parameter, thanks to which we'll see the executable file which establishes the connection. We'll also use "t" with the parameter "1", so that the program refreshes our results each second. We enter netstat –b –t 1 and again, we have to wait for a while until the keylogger sends us some data.

```
Active Connections

  Proto  Local Address          Foreign Address        State

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    192.168.0.103:1053     my_hidden_host:http    SYN_SENT
 [Opera.exe]

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    192.168.0.103:1053     my_hidden_host:http    SYN_SENT
 [Opera.exe]
```

We can close the two previous consoles because we won't need them anymore. A connection should appear in a moment. Then, we'll halt the netstat execution using the CTRL+C combination.

Our keylogger hasn't sent us anything for awhile, so let's check whether it's actually in the process list. Unfortunately, it seems that it closed. Let's start it and wait for the data. We can see a connection appeared. We've stopped the action and we see that the opera.exe process is sending a packet. As we remember, our keylogger launches the browser in order to bypass the firewall. Obviously, we didn't launch the browser, so some application had to use it to send a packet. However, we still don't know which application is responsible for that.

explorer.exe	1688	0.21	52,136 K	72,184 K	Windows Explorer	Microsoft Corporation
RtHDVCpl.exe	3212		6,528 K	25,148 K	HD Audio Control Panel	Realtek Semiconductor
DataCardMonitor.exe	3224		1,352 K	22,856 K	DataCardMonitor MFC Applic...	Huawei Technologies Co.,...
egui.exe	3272	0.02	4,132 K	28,920 K	ESET GUI	ESET
PPTVIEW.EXE	5600	2.04	59,592 K	77,880 K	Microsoft Office PowerPoint ...	Microsoft Corporation
CamRecorder.exe	5764	20.69	99,456 K	120,672 K		
TscHelp.exe	5876		808 K	21,516 K	TechSmith HTML Help Helper	TechSmith Corporation
cmd.exe	5240		1,900 K	2,484 K		
keylogger.exe	4448		1,556 K	4,928 K		
opera.exe	5484	39.50	544 K	2,104 K	Opera Internet Browser	Opera Software
procexp.exe	2980	5.43	9,536 K	15,136 K	Sysinternals Process Explorer	Sysinternals - www.sysinter...
devenv.exe	6060	0.11	178,628 K	246,808 K	Microsoft Visual Studio 2010	Microsoft Corporation
vcpkgsrv.exe	4552	< 0.01	9,968 K	85,328 K	Microsoft (R) Visual C++ Pac...	Microsoft Corporation

Now let's launch the ProcessExplorer application to figure that out. We see that the keylogger already launched the opera.exe process and based on this we may conclude that keylogger.exe is the application which starts the browser and communicates with the Internet without our permission. Once we have this information, we can safely close the process and remove the harmful file.

Let's sum up what we've learned in this module. We've just seen how to detect IAT hooking. Thanks to that, we've learnt which module in our system is responsible for hooking. On this basis we can determine which application uses this module. It can help us link the situation to the actual aggressor. We've also seen how to identify a suspicious program based on its network activity. Using the netstat application we've managed to establish that it was a browser that was used to send the packets. Eventually we've managed to trace the harmful code using the Process Explorer, which demonstrated that the keylogger is responsible for unauthorised data transmission. That is everything when it comes to this module. Thank you for your attention and please go to the next module, where we'll summarise the knowledge we've learned so far. See you there.