A network hooking application

epokh @ UIC audit
Personal website: www.epokh.org/drupy
UIC website: www.quequero.org

December 1, 2006

Abstract

A network hooking application, from this moment we call it SkypeLogger, that can control the UDP traffic of the Skype famous program. For control it means it can:

- log the UDP traffic in different format: XML and ARFF The logged traffic could be used for latter network analysis
- filter the UDP traffic: the user can block some IP address communication defining some firewall rules
- other amazing stuff the user want: i.e. inserting some "junk traffic" in the data stream. *This can be useful for audit purposes, like finding potential buffer overflow in a client-server application.*

The same application could also be used, with some work, to log the traffic of other kind of program using the TCP/IP stack.

Contents

Chapter 1

Introduction

The article is divided in two main chapter:

1. Chapter 1: Hooking Engine
   Describe the hooking framework used to build the SkypeLogger
2. Chapter 2: The Application Describe how to use the framework to build the network traffic logger.
3. Chapter 3: The data analysis
   Show some data analysis with the Weka software
4. Chapter 4: Skype Reference

The reader should have have a basic knowledge about the Skype protocol. Reference to the reveresed protocol is included in the chapter 4.

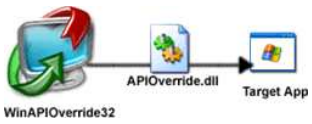
Chapter 2

Hooking engine

The framework used for the hooking engine is called *WinAPIOverride32* developed by Jacquelin Poitier, website is *http://jacquelin.potier.free.fr/*, released under GPL license. Following a basic introduction to this engine, reported from the author s website.

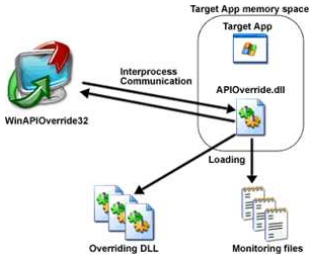
2.1 How it works

We first inject APIOverride dll in the target process using the Injlib code written by Jeffrey Richter. (Creating a remote thread with as start address the LoadLibrary func and as parameter the name of dll to be injected)



Injecting APIOverride dll The DllMain of APIOverride.dll then initialize interprocess communication so injected dll can be managed by a remote application. The injected dll job is to install hooks, load dll and monitoring files, and execute functions in target process (see How remote call works)

Loading monitoring files or overriding dll Target memory space after overriding dll load So project is composed of tow main parts : - WinApiOverride32.exe : user interface sample for the CAPIOverride class (class to easily manage injected dll) - APIOverride.dll : the injected dll.



CAPIOverride object allows you to do all requiered stuff to manage the injected dll. CAPIOverride object main methods are the following, (they are reported in a sequential order):

```
//-------------------------------------------------------------------------------
// Name: CApiOverride
// Object: Constructor
//         use SetMonitoringCallback or SetMonitoringListview next to monitor
//         hooks
// Parameters :
//     in :
// Return :
//-------------------------------------------------------------------------------
CApiOverride::CApiOverride()


//-------------------------------------------------------------------------------
// Name: CApiOverride
// Object: Constructor. Listview will be configured automatically, and it will be
// filled by monitoring events
// Parameters :
//     in : HWND hListView: Handle to a list view
// Return :
//-------------------------------------------------------------------------------
CApiOverride::CApiOverride(HWND hListView)

//-------------------------------------------------------------------------------
// Name: CApiOverride
// Object: Constructor to manage yourself logging event
// Parameters :
//     in : tagCallBackLogFunc pCallBackLogFunc : monitoring callback
//          warning we use mail slot so callback can be called few seconds after
//          real function call
//          for real time function hooking just use a dll (see fake API dll sample)
// Return :
//-------------------------------------------------------------------------------
CApiOverride::CApiOverride(tagCallBackLogFunc pCallBackLogFunc)


//-------------------------------------------------------------------------------
// Name: GetProcessID
// Object: return the process ID with which CApioverride is working or has
// worked at last
// Parameters :
// Return : PID if CAPIOverride
//-------------------------------------------------------------------------------
DWORD CApiOverride::GetProcessID()

//-------------------------------------------------------------------------------
// Name: Start
// Object: inject API Override dll in selected process ID to allow monitoring
```

```
// and faking
// Parameters :
//      in : DWORD dwPID
// Return : FALSE on error, TRUE if success
//-----------------------------------------------------------------------------
BOOL CApiOverride::Start(DWORD dwPID)

//-----------------------------------------------------------------------------
// Name: Start
// Object: start the software specified by pszFileName, inject API Override dll
// at start up
// Parameters :
//      in : TCHAR* pszFileName : path of software to start
// Return : FALSE on error, TRUE if success
//-----------------------------------------------------------------------------
BOOL CApiOverride::Start(TCHAR* pszFileName)

//-----------------------------------------------------------------------------
// Name: LoadMonitoringFile
// Object: start monitoring API hooked by the file (multiple files can be hooked
// at the same time)
// Parameters :
//      in : TCHAR* pszFileName: api monitoring file
// Return : TRUE if file is partially loaded (even if some non fatal errors occurs)
//           and so needs a call to UnloadMonitoringFile to restore all hooked func,
//           FALSE if file not loaded at all
//-----------------------------------------------------------------------------
BOOL CApiOverride::LoadMonitoringFile(TCHAR* pszFileName)

//-----------------------------------------------------------------------------
// Name: UnloadMonitoringFile
// Object: stop monitoring API hooked by the file
// Parameters :
//      in : TCHAR* pszFileName: api monitoring file
// Return : FALSE on error, TRUE if success
//-----------------------------------------------------------------------------
BOOL CApiOverride::UnloadMonitoringFile(TCHAR* pszFileName)

//-----------------------------------------------------------------------------
// Name: LoadFakeAPI
// Object: load dll and start faking api/func specified in the specified in the dll
//          (see the Fake API sample for more infos on specifying API in dll)
//          multiple fake library can be hooked at the same time
// Parameters :
//      in : TCHAR* pszFileName: fake api dll name
// Return : TRUE if library is loaded (even if some non fatal errors occurs)
//           and so needs a call to UnloadFakeAPI to restore all hooked func,
//           FALSE if library not loaded
//-----------------------------------------------------------------------------
BOOL CApiOverride::LoadFakeAPI(TCHAR* pszFileName)

//-----------------------------------------------------------------------------
// Name: UnloadFakeAPI
// Object: stop faking api/func hooked by the dll before unloading this dll
// Parameters :
//      in : TCHAR* pszFileName: fake api dll name
// Return : FALSE on error, TRUE if success
//-----------------------------------------------------------------------------
BOOL CApiOverride::UnloadFakeAPI(TCHAR* pszFileName)


//-----------------------------------------------------------------------------
// Name: StartMonitoring
// Object: restore monitoring until the next StopMonitoring
//          API Override dll do monitoring by default (at start up)
//          So you only need to call this function after a StopMonitoring call
// Parameters :
//      in :
// Return : TRUE on Success
//-----------------------------------------------------------------------------
BOOL CApiOverride::StartMonitoring()


//-----------------------------------------------------------------------------
```

```
// Name: StopMonitoring
// Object: Temporary stop monitoring until the next StartMonitoring call
// Parameters :
//      in :
// Return : TRUE on Success
//-----------------------------------------------------------------------------
BOOL CApiOverride::StopMonitoring()


//-----------------------------------------------------------------------------
// Name: StartFaking
// Object: restore faking until the next StopFaking
//         API Override dll do faking by default (at start up)
//         So you only need to call this function after a StopFaking call
// Parameters :
//      in :
// Return : TRUE on Success
//-----------------------------------------------------------------------------
BOOL CApiOverride::StartFaking()

//-----------------------------------------------------------------------------
// Name: StopFaking
// Object: Temporary stop faking until the next StartFaking call
// Parameters :
//      in :
// Return : TRUE on Success
//-----------------------------------------------------------------------------
BOOL CApiOverride::StopFaking()
```

2.2 How hook works

Given param for hooking are dll name and function name, or direct function address. Doing GetModuleHandle or LoadLibrary with dll name, followed by GetProcAddress with function name, we get function address to hook. So now in the two way of giving param, you have function address to hook.

To create the hook we have to modify first asm instruction at function address. For example for MessageBoxA function in User32.dll, GetProcAddress return 0x77d3add7 on a XP OS. So to make our hook we just replace byte at address 0x77d3add7 by "call OurHandlerAddr" with WriteProcessMemory. If we want to give some parameters to our handler function, you can change code by "push OurHandlerParam1 push OurHandlerParam2 ... call OurHandlerAddr"

Entering the hook, the instruction history is :

1. push HookedFunc params
2. call HookedFunc → push return address of HookedFunc
3. code modification of first bytes of API
    - push OurHandlerParam
    - call OurHandlerAddr → push return address of our asm call `(HookedFuncAddr+OPCODE_REPLACEMENT_SIZE)`

So the stack is in following state

- return address of our asm call
- OurHandlerParam
- return address of HookedFunc

- HookedFunc params

To get all these informations, at the begining of our hook handler we use the following asm code :

```
_asm
{
// <-- return address of our asm call (==HookedFuncAddr+OPCODE_REPLACEMENT_SIZE)
    Mov EAX, [EBP + 4]
    Mov [pbAfterCall], EAX

    Mov EAX, [EBP + 8]
    Mov [pAPIInfo], EAX

    Mov EAX, [EBP + 12] // <-- return address of hooked func
    Mov [pvReturnAddr], EAX

    Lea EAX, [EBP + 16]
    Mov [pdwParam], EAX
}
```

To call original function inside the hook, we first have to restore original asm instruction. Next, as we know parameters number (they are given in configuration file), we can push them on the stack again and call original function or another one. We can put a security option by pushing more data than declared parameters in configuration file to avoid crash in case of bad config file. To log parameters, we just copy them from stack before or/and after function call.

We next need to restore the hook for next function call. Finally we have to put the stack in the same state as it should be after a call without the hook. We have to split the case of StdCall (callee remove params form stack) and Cdecl (caller remove params form stack)

```
//////////////////////////////////////////////////////////////////////
// Restore Stack (make the same job real Api do without hooking)
//////////////////////////////////////////////////////////////////////
if (bStackCleanedByCallee)
{
    _asm
    {
        Mov EAX, [dwReturnValue]// restore API return value
        Mov ECX, [dwParamSize]
        Mov EDX, [pvReturnAddr]

        Pop EDI
        Pop ESI
        Pop EBX

        Mov ESP, EBP
        Pop EBP       // from now you can't access your function local var
        Add ESP, 8    // remove frame pointer and return addr
        Add ESP, 4    // remove our pAPIInfo param from stack
        // now esp points to second ret address --> ret will use return address
         of hooked func

        Add ESP, ECX // remove params from stack

        Push EDX      // push return addr

        Ret
    }
```

2.3 How hook at startup works ?

There are two different ways of hooking :

- When we use the "Attach to all new processes" options, we don't create process, so we don't now when the procmon driver callback appends. So we

can't hook entry point of the process (it has maybe already been reached); and we can't inject dll as soon as we enter the callback, because the NT loader may don't has finished its job (and if we try to inject at this moment, the application crashs like it appends in version of WinAPIOverride before the 2.1) So we have to do pooling on the end of the NT Loader. As soon as there's a module loaded in the process, we can do injection. So to detect the existence of the first module, we do pooling on the Module32First API, and as soon as the result of API is TRUE, we inject our library.

- When we use the "Attach at application startup", we create process in a suspended state, so we know that entry point as not been reached. We could put a breakpoint and debug process, but debugging a process changes some of its attributes. So we use a different way: we install a hook at the entry point. So the entry point content is replaced by jmp OurHookAddress. Where OurHook is a function that loads the APIOverride dll, restore original content of entry point and suspend the process again to allow user to load his monitoring and faking files. Of course this function don't exists in the process we want to hook, and we have to inject code into it by a call to VirtualAllocEx and WriteProcessMemory.

2.4 How remote call works ?

To call a function from another process, we have to transmit parameter by any way (shared memory, mailslot, pipe, tcp ...), and then push them to the stack before calling function address. We next transmit parameters and return back to our process. The following code allow to call function with any parameter/struct and only SIMPLE pointer to any parameter/struct

```
typedef struct _STRUCT_FUNC_PARAM
{
    BOOL bPassAsRef;     // true if param is pass as ref
    DWORD dwDataSize;    // size in byte
    PBYTE pData;         // pointer to data
}STRUCT_FUNC_PARAM, *PSTRUCT_FUNC_PARAM;

//-----------------------------------------------------------------------------
// Name: ProcessInternalCallRequest
// Object: call function at address pFunc with parameters specified in pParams
//          and store function return (eax) in pRet
// Parameters :
//      in: FARPROC pFunc : function address
//          int NbParams : nb params in pParams
//          PSTRUCT_FUNC_PARAM pParams : array of STRUCT_FUNC_PARAM. Can be null
//          if no params
//      out : DWORD* pRet : returned value
// Return :
//-----------------------------------------------------------------------------
void ProcessInternalCallRequest(FARPROC pFunc,int NbParams,

{
    int cnt;
    int cnt2;
    DWORD dw;
    BYTE b;
    USHORT us;
    PSTRUCT_FUNC_PARAM pCurrentParam;
    DWORD dwOriginalESP;

    _asm
    {
        // store esp to restore it without carring about calling convention
        mov [dwOriginalESP],ESP
    }
    // for each param
    for (cnt=NbParams-1;cnt>=0;cnt--)
```

```
    {
        pCurrentParam=&pParams[cnt];

        // if params should be passed as ref
        if (pCurrentParam->bPassAsRef)
        {
            // push param address
            dw=(DWORD)pCurrentParam->pData;
            _asm
            {
                mov eax,dw
                push eax
            }
        }
        else // we have to push param value
        {
            // byte
            if (pCurrentParam->dwDataSize==1)
            {
                b=pCurrentParam->pData[0];
                _asm
                {
                    mov al,b
                    push eax
                }
            }
            // short
            else if (pCurrentParam->dwDataSize==2)
            {
                memcpy(&us,pCurrentParam->pData,2);
                _asm
                {
                    mov ax,us
                    push eax
                }
            }
            // dword
            else if (pCurrentParam->dwDataSize==4)
            {
                memcpy(&dw,pCurrentParam->pData,4);
                _asm
                {
                    mov eax,dw
                    push eax
                }
            }
            // more than dword
            else
            {
                for (cnt2=pCurrentParam->dwDataSize-4;cnt2>=0;cnt2-=4)
                {
                    memcpy(&dw,&pCurrentParam->pData[cnt2],4);
                    _asm
                    {
                        mov eax,dw
                        push eax
                    }
                }
            }
        }
    }
    // now all params are pushed in stack --> just make call
    _asm
    {
        // call func
        call pFunc
        // put pointer to return address in ecx
        mov ecx,[pRet]
        // put return value in the adress pointed by ecx --> *pRet=eax
        mov [ecx],eax
        // restore esp (works for both calling convention)
        mov ESP,[dwOriginalESP]
    }
}
```

By the way, for GetTempPathA(DWORD nBufferLength,LPSTR lpBuffer) func in kernell32, the STRUCT_FUNC_PARAM struct can be filled like

```
STRUCT_FUNC_PARAM pParams[2];
DWORD dwParam=255;
char pc[255];
pParams[0].bPassAsRef=FALSE;
pParams[0].dwDataSize=4;
pParams[0].pData=(PBYTE)&dwParam;
pParams[1].bPassAsRef=TRUE;
pParams[1].dwDataSize=dwParam;
pParams[1].pData=(PBYTE)pc;
```

More example of use of remote process call are done in file main.cpp of WinApiOverride32.exe project inside function GetRemoteWindowInfos().

2.5 How to override an API or internal function ?

The injected dll quite do all the thing for you. You just need to use the CAPIOverride-¿LoadFakeAPI(char* pszFileName); were pszFileName is a name of a dll having the following 3 specifics points : (you can take a look at FakeMsgBox sample project given in source code) You have to

1) include GenericFakeAPI.h (located in Tools\Process\APIOverride) define an array of functions (where handler is your overriding function)

Sample for API or dll overriding

```
STRUCT_FAKE_API pArrayFakeAPI[]=
{
// library name ,function name, function handler, number of params
(requiered to allocate enougth stack space)
{_T("User32.dll"),_T("MessageBoxW"),(FARPROC)mMessageBoxW,4},
{_T("User32.dll"),_T("MessageBoxA"),(FARPROC)mMessageBoxA,4},
{_T(""),_T(""),NULL,0}// last element for ending loops
};
```

Sample for exe internal call overriding

```
STRUCT_FAKE_API pArrayFakeAPI[]=
{
// library name ,function name, function handler,
number of params (requiered to allocate enougth stack space)
{_T("EXE_INTERNAL@0x401000"),_T("add"),(FARPROC)mInternaAddSample,2},
{_T(""),_T(""),NULL,0}// last element for ending loops
};
```

Notice : It's better to specify good number of parameters, but if it is wrong, as said in How hook works, a security is added for stack so your soft won't crash. To get API definition best ways are to use MSDN and next microsoft visual c++ SDK headers. Speedest way in Visual studio is to enter the func name, right click, and next hit "Go to Definition". **3) add the following funcs which export your array and gives encoding type To export your functions array:**

```
//-----------------------------------------------------------------
// Name: GetFakeAPIArray
// Object: allow calling module to get fake api array pointer
// Parameters :
// in :
// out :
// return : pArrayFakeAPI array struct previously defined
//-----------------------------------------------------------------
extern "C" __declspec(dllexport) void* __stdcall GetFakeAPIArray()
```

```
    {
        return (void*)pArrayFakeAPI;
    }
```

To specify encoding type (Ansi or Unicode) of dll

```
1.
    //--------------------------------------------------------------------------
    // Name: GetFakeAPIEncoding
    // Object: allow calling module to know character encoding of pArrayFakeAPI
    // Parameters :
    // in :
    // out :
    // return :
    //--------------------------------------------------------------------------
    extern "C" __declspec(dllexport) int __stdcall GetFakeAPIEncoding()
    {
        #if (defined(UNICODE)||defined(_UNICODE))
            return FakeAPIEncodingUNICODE;
        #else
            return FakeAPIEncodingANSI;
        #endif
    }
```

Notice: if the function is not found your faking dll is concidered as being Ansi encoded.
And That's all !
Warning: It's better to check your code before using it as an injected dll, because it's avoid to crash target. See How to debug a fake API dll for more debugging informations.

Full MessageBox faking code example is the follwing:

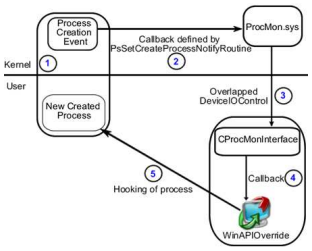
```
#include "../../GenericFakeAPI.h"
// You just need to edit this file to add new fake api
// WARNING YOUR FAKE API MUST HAVE THE SAME PARAMETERS AND CALLING CONVENTION
/
//                    ELSE YOU WILL GET STACK ERRORS

////////////////////////////////////////////////////////////////////////////
// fake API prototype MUST HAVE THE SAME PARAMETERS
// for the same calling convention see MSDN :
// "Using a Microsoft modifier such as __cdecl on a data declaration is an
// outdated practice"
////////////////////////////////////////////////////////////////////////////

int WINAPI mMessageBoxW(HWND hWnd,LPCWSTR lpText,LPCWSTR lpCaption,UINT uType);
int WINAPI mMessageBoxA(HWND hWnd,LPCSTR lpText,LPCSTR lpCaption,UINT uType);

////////////////////////////////////////////////////////////////////////////
// fake API array. Redirection are defined here
////////////////////////////////////////////////////////////////////////////
STRUCT_FAKE_API pArrayFakeAPI[]=
{
    // library name ,function name, function handler,
    number of params (requiered to allocate enougth stack space)
    {_T("User32.dll"),_T("MessageBoxW"),(FARPROC)mMessageBoxW,4},
    {_T("User32.dll"),_T("MessageBoxA"),(FARPROC)mMessageBoxA,4},
    {_T(""),_T(""),NULL,0}// last element for ending loops
};

////////////////////////////////////////////////////////////////////////////
// Only the 2 following functions requiere to be exported (GetFakeAPIArray and
//  GetFakeAPIEncoding)
////////////////////////////////////////////////////////////////////////////

//--------------------------------------------------------------------------
// Name: GetFakeAPIArray
// Object: allow calling module to get fake api array pointer
// Parameters :
//     in  :
//     out :
//     return : pArrayFakeAPI array struct previously defined
```

```
//-----------------------------------------------------------------------------
extern "C" __declspec(dllexport) void* __stdcall GetFakeAPIArray()
{
    return (void*)pArrayFakeAPI;
}

//-----------------------------------------------------------------------------
// Name: GetFakeAPIEncoding
// Object: allow calling module to know character encoding of pArrayFakeAPI
// Parameters :
//     in  :
//     out :
//     return : encoding type
//-----------------------------------------------------------------------------
extern "C" __declspec(dllexport) int __stdcall GetFakeAPIEncoding()
{
#if (defined(UNICODE)||defined(_UNICODE))
    return FakeAPIEncodingUNICODE;
#else
    return FakeAPIEncodingANSI;
#endif
}

////////////////////////////////////////////////////////////////////////////////
//////////////////////////// NEW API DEFINITION ////////////////////////////////
////////////////////// You don't need to export these functions ////////////////
////////////////////////////////////////////////////////////////////////////////

int WINAPI mMessageBoxA(HWND hWnd,LPCSTR lpText,LPCSTR lpCaption,UINT uType)
{
    UNREFERENCED_PARAMETER(lpCaption);

    char szMsg[2*MAX_PATH];
    strcpy(szMsg,"Oups !! It's not your text !!!\r\n[Begin of] Original Text :\r\n");
    strncat(szMsg,lpText,MAX_PATH-strlen(szMsg)-1);
    szMsg[MAX_PATH-1]=0;// assume end of string
    return MessageBoxA(NULL,szMsg,"API_Override:MessageBoxA",uType);
}


int WINAPI mMessageBoxW(HWND hWnd,LPCWSTR lpText,LPCWSTR lpCaption,UINT uType)
{
    UNREFERENCED_PARAMETER(lpCaption);
    wchar_t szMsg[2*MAX_PATH];

    wcscpy(szMsg,L"Oups !! It's not your text !!!\r\n[Begin of] Original Text :\r\n");
    wcsncat(szMsg,lpText,MAX_PATH-wcslen(szMsg)-1);
    szMsg[MAX_PATH-1]=0;// assume end of string

    return MessageBoxW(NULL,szMsg,L"API_Override:mMessageBoxW",uType);
}
```

2.6 How process creation monitoring is done ?

All is done by a call to PsSetCreateProcessNotifyRoutine function. This function defines a callback called at each process creation and giving as parameters process ID and parent process ID.
Ok it's easy so ?
Not really as this function can be called only in kernel mode, that means we have to make a driver (ProcMon.sys) All sources of this driver are located in the Tools\Process\ProcessMonitor\Src\ProcMon directory, but as the windows DDK (Driver Development Kit) is required to build it, the debug and release version of the driver are included in sources. To easely manage this driver, the class CProcMonInterface as been written (located in Tools\Process\ProcessMonitor) It allows to define a callback (in user mode of course), start and stop the driver,

and start or stop process creation monitoring



2.7 How to debug a fake API dll ?

You have created a fake dll, but host process crashs we you try to use it. First of all, assume the monitoring of the function you want to hook works. Make a monitoring file as describe here. As Monitoring and faking use same algorithm, if monitoring works, faking must work (else don't try continue, it's no use; you have to make monitoring work before). So if monitoring works, that means there's bug inside your faking code. As you are in a remote process, and inside a hook done with asm instruction, you can't debug your dll using your targeted application.

First, a good thing to do is to determine which parameters where used during the crashing call. To do it, as monitoring and faking can be mixed, just monitor the function you fake specifying the InNoRet as parameter directions in the created monitoring file. So when your fake dll crashs, the last trace in WinAPIOverride32 interface shows you which parameters make your fake function crash. As this point, you get more informations to debug your dll. So to debug it, just make an exe debug project. From here you can use classical dll debugging technics by building your faking dll in debug mode, and then, load it from your debug app (or more dirty (but some time speedest) copy code from your dll sources and paste it in your debug app). From here just use your classical debugging technics or some great softs like Numega Devpartner or BoundsChecker to resolve memory troubles.

2.8 How to Monitor API or functions

To load a monitoring file, just choose the file and click "Load" in "API Monitoring Configuration" panel. To unload it, just select it in monitoring file list and click "Unload". If hook is in use, a warning appears to ask you for not unhooking now. Just wait more or make action that can unlock the hook. All monitored call will be displayed in listview at the bottom of the application. You can

start/stop monitoring all functions by clicking “Start Monitoring”/”Stop Monitoring”
Notice : you can monitor overrided functions.
Monitoring file syntax

1. **Commented lines** : all lines begining with ”;” will be considered as comments
2. **API or dll function :**

```
LibraryName|[ReturnType]FuncName(ParamType [paramName])[;][|Dir]
where all in [] is optional, and Dir can be one of the following:
"In", "Out", "InOut", "InNoRet" (default parameter direction is "In")
InNoRet allows to send log to WinAPIOverride before the function is called,
so even function crash we get logs
```

Examples

```
KERNEL32.DLL|HMODULE LoadLibraryW(PWSTR)
KERNEL32.DLL|LoadLibraryExA(LPCSTR lpFileName,HANDLE hFile,DWORD dwFlags);|In
```

Samples of monitoring files can be found in the “monitoring file” directory
Notice : To get API definition best ways are to use MSDN and next microsoft visual c++ SDK headers. Speedest way in Visual studio is to enter the func name, right click, and next hit “Go to Definition”.
Monitoring function inside process :

```
EXE_INTERNAL@0xHexaAddress|[ReturnType]FuncName(ParamType [paramName])[;][|Dir]
where HexaAddress is the address of the function, all in [] is optional,
and Dir can be one of the following: "In", "Out", "InOut"
(default parameter direction is "In")
```

Example (see fig 2.8)

```
EXE_INTERNAL@0x401000|Add(int x,int y)|In
```

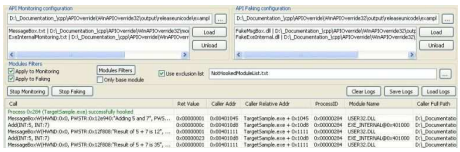


Figure 2.1: Filter example

2.9 What is the HideMe.dll ?

This dll is an example, a proof of concept, to show how to hide your process from your targeted application. This dll only fakes some wellknown functions used to list processes and retreive the parent Id of a process. Don't believe that WinAPIOverride32 is fully hidden: no events nor handles are hidden, as previously said, it's only an example. To concretely see the result of this dll, just open two windows task manager (or two Process Explorer from wininternals). Next attach one of the two processes with WinAPIOverride32, and load HideMe.dll ("Load" button in the "API Faking Configuration" panel). Finally see the results between the two applications: some processes are leaking in the injected process. You can play with the "Start Faking" "Stop Faking" button too to see these differences.

Chapter 3

The SkypeLogger

3.1 Define the hooking functions

The first step to build the logger is to find the function that the application, in our case Skype, will call to send and receive data from the other application. Because we want so catch all the udp traffic the two key functions are:

- **sendto**
  The sendto function sends data to a specific destination.

- **recvfrom**
  The recvfrom function receives a datagram and stores the source address.

Both of them are defined in the Windows Socket 2 system and are defined in Ws2_32.dll, while the header file for the definition is Winsock2.h.
So in our fake dll we must define our fake functions that will be called at the place of the original ones. In the GenericFakeAPI.h we had the following including istructions:

```
#include <windows.h>
#pragma comment(lib,"ws2_32.lib")
```

And in FakeApi.cpp we define the prototype of our handling functions and add them to the FakeAPIArray, as following:

```
//prototype definitions

int  WINAPI mysendto(
  SOCKET s,
  const char* buf,
  int len,
  int flags,
  const struct sockaddr* to,
  int tolen
);

int WINAPI  myrecvfrom(
  SOCKET s,
  char* buf,
```

```
  int len,
  int flags,
  struct sockaddr* from,
  int* fromlen
);

////////////////////////////////////////////////////////////////////////////
// fake API array. Redirection are defined here
////////////////////////////////////////////////////////////////////////////
STRUCT_FAKE_API pArrayFakeAPI[]=
{
    // library name ,function name, function handler, number of params
    //(requiered to allocate enougth stack space)
        {_T("ws2_32.dll"),_T("sendto"),(FARPROC)mysendto,6},
        {_T("ws2_32.dll"),_T("recvfrom"),(FARPROC)myrecvfrom,6},
  {_T(""),_T(""),NULL,0}// last element for ending loops
};
```

3.2 Description of the used class

In this section I describe the class,data structure and algorithm I used to proper handle the network data stream traffic. The classes are:

- ipaddr_block
- IPelement
- IPstream
- PacketManip
- LoadSaveClass
- XMLNode,XMLResults

I'm describing now the classes starting from the basic ones to the advanced one.

3.2.1 LoadSaveClass

The LoadSaveClassi,in filtersLoadSave.h, is defined in this way:

```
class LoadSaveClass{

public:
        //load the IP list we want to filter from a configuration file
        bool static loadIPfilter(string);
        //load the supernode list from the skype configuration file
        bool static loadHostCache(string);
        //save the UDP traffic in Arff format
        bool static saveUDPstreamAsArff(string);
        bool static saveTCPstreamAsArff(string);
        //save the UDP stream in a XML document
        bool static saveUDPstreamAsXML(string);

};
```

The loadIPfilter method load the IPs we want to filter from a text configuration file, called ip2filter with a simple structure, like:

```
127.0.0.1:123
207.81.237.124:456
172.212.123.242:170
```

This list is loaded in a map data structure defined as `{map<ipaddr_block,IPelement> filterlist` where the key is the an ippaddr_block element and the value is an IPelement, these object will be described later. We want to store the ip list in an hashmap because the most important operation is to find an ip quickly. The loadHost-Cache is a method which load the supernode list from the skype configuration file. For example on my Pc the file is located at `string skypeXML("C:\\Documents and Settings\\epokh\\Dati applicazioni\\Skype\\shared.xml");`. The host cache is loaded in another hashmap defined as `map<ipaddr_block,IPelement> supernodelist;` for the same reason I explained above. The supernode list is locate in the XML file in this position: config→lib→connection→HostCache, so we need to explore the XMl tree document in such a way:

```
bool LoadSaveClass::loadHostCache(string fileName)
{

        // this open and parse the XML file:
  XMLNode xMainNode=XMLNode::openFileHelper(fileName.c_str(),"config");

  if(xMainNode.isEmpty())
  {
          return false;
  }
  else
  {
  XMLNode xNode=xMainNode.getChildNode("lib");
  xNode=xNode.getChildNode("Connection");
  //select the HostCache node
  XMLNode xNodeHost=xNode.getChildNode("HostCache");
  //count the number of the child elements, in our case the
  //number of supernodes
  int nhosts=xNodeHost.nChildNode();

  for(int i=1;i<=nhosts;i++)
  {
          char temp[100];
          string node_name("_");
          itoa(i,temp,10);
          node_name.append(temp);
          //get every supernode and add it in the map
          xNode=xNodeHost.getChildNode(node_name.c_str());
          IPelement *IPelem=new IPelement(string(xNode.getText()));
          supernodelist.insert(pair<ipaddr_block,IPelement>(IPelem->get_IPnum(),*IPelem));
          OutputDebugStr("Element host cache added to the list: ");
          OutputDebugStr(IPelem->get_IPchar());

          delete IPelem;

  }
   return true;
  }

}
```

An example of the IPs in my current list is:

```
<HostCache>
  <_1>82.34.50.47:42131,10</_1>
```

```
  <_10>131.251.20.47:29498,10</_10>
  <_100>81.31.127.150:49710,10</_100>
  <_101>86.9.68.84:51118,10</_101>
  <_102>84.104.197.9:9260,10</_102>
  <_103>88.115.247.135:56495,10</_103>
  <_104>80.177.219.68:2892,10</_104>
  <_105>81.173.255.230:40101,10</_105>
  .... others ....
 </HostCache>
```

The method `saveUDPstreamAsXML(string fileName)` save the UDP stream in an XML file name. The UDP stream is saved during the hooking process in another hashmap, defined as `map<ipaddr_block,IPstream> udpstream;`. To traffic is grouped by the IP address and divided between incoming and outcoming, a bit flag tell if the peer host is a supernode or not. A traffic log example is:

```
<?xml version="1.0"?>
<Skype_UDP_log timestamp="16/06/2006">
<udp_stream_info ip="24.30.19.125" total_packets="2" is_supernode="1" total_packets_in="1" total_packets_out="1"/>
<udp_stream_info ip="65.93.113.51" total_packets="1" is_supernode="1" total_packets_in="0" total_packets_out="1"/>
<udp_stream_info ip="68.98.188.131" total_packets="1" is_supernode="1" total_packets_in="0" total_packets_out="1"/>
<udp_stream_info ip="69.116.165.113" total_packets="1" is_supernode="1" total_packets_in="0" total_packets_out="1"/>
<udp_stream_info ip="81.67.60.222" total_packets="1" is_supernode="1" total_packets_in="0" total_packets_out="1"/>
<udp_stream_info ip="82.68.97.46" total_packets="1" is_supernode="1" total_packets_in="0" total_packets_out="1"/>
      <udp_stream_info ip="82.68.34.56" total_packets="23" is_supernode="1" total_packets_in="20" total_packets_out="3"/>
      ...others...
</Skype_UDP_log>
```

The classes described are used when the dll is loaded and when the dll is unloaded, in this way:

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{

switch (fdwReason)
{
        case DLL_PROCESS_ATTACH:
                OutputDebugStr("My dll attach");

          if(LoadSaveClass::loadHostCache(skypeXML))
                {
                        OutputDebugStr("HostCache loaded!");

                }
                if(LoadSaveClass::loadIPfilter("C:\\Programmi\\Skype\\Phone\\ip2filter.txt"))
                {
      OutputDebugStr("IP filter list loaded!");
                }
                if(filterlist.empty())
                   OutputDebugStr("The hashmap ipfilter is empty");
            if(supernodelist.empty())
              OutputDebugStr("The hashmap supernode is empty");
                        break;

        case DLL_PROCESS_DETACH:
                LoadSaveClass::saveUDPstreamAsArff("C:\\Programmi\\Skype\\Phone\\udp_stats.arff");
                LoadSaveClass::saveTCPstreamAsArff("C:\\Programmi\\Skype\\Phone\\tcp_stats.arff");
                LoadSaveClass::saveUDPstreamAsXML("C:\\Programmi\\Skype\\Phone\\udp_stats.xml");
                OutputDebugStr("My dll detach");
                break;
        default:
                break;
}
        return TRUE;
}
```

It means saving the traffic when the dll is unloaded by the event `DLL_PROCESS_DETACH` and loading the lists when the dll is loaded by the event `DLL_PROCESS_ATTACH`.

3.2.2 Ipaddr block class

The class ipaddr_block is a basic element describing an IP address in an easy way.

```
 class ipaddr_block
{
 public:
   int byte1;
   int byte2;
   int byte3;
   int byte4;
   int port;
   ipaddr_block();
   ipaddr_block(string ip);
   friend bool operator==(const ipaddr_block o1,const ipaddr_block o2);
   friend bool operator<(const ipaddr_block o1,const ipaddr_block o2);

};
```

The constructor `ipaddr_block::ipaddr_block(string ip)` build the the block element from a string ip format that could be in two possible formats:

- b1.b2.b3.b4
- b1.b2.b3.b4:port

The constructor is useful to load the address from the configuration file and the ip filter list, where the ip are formatted in such a way. The other two overloaded operators are useful for ordering the element in the hashmap and to compare two ip block elements. Two ip block element are equal if they have the same address not considering the port number. One block element is less than another block element if the decimal number composed by the whole ip number is less than the other.

3.2.3 IPelement class

The Ipelement class is composed by the previous class, with additional fields describing if the element is a supernode.

```
class IPelement{
public:
  ipaddr_block *IPnum;
  string IPstring;
  bool isSuperNode;
public:
        IPelement();
        IPelement(string ip);
        IPelement(const struct sockaddr_in* to);
    string get_IPstring(){return IPstring;}
        const char* get_IPchar(){return IPstring.c_str();}
        ipaddr_block get_IPnum(){return *IPnum;}
        string num2str(long int n);

};
```

The constructor `IPelement::IPelement(const struct sockaddr_in* to)` build the ipelement from a sockaddr_in structure, defined as:

```
struct sockaddr_in{
   short sin_family;
   unsigned short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];
};
```

the method is defined as:

```
IPelement::IPelement(const struct sockaddr_in* to)
{
   IPnum=new ipaddr_block();
   IPnum->byte1=to->sin_addr.S_un.S_un_b.s_b1;
   IPnum->byte2=to->sin_addr.S_un.S_un_b.s_b2;
   IPnum->byte3=to->sin_addr.S_un.S_un_b.s_b3;
   IPnum->byte4=to->sin_addr.S_un.S_un_b.s_b4;
   IPnum->port=to->sin_port;

   IPstring=string(inet_ntoa(to->sin_addr));
}
```

3.2.4 infoPacket and Ipstream

The infoPacket class, contain information about the packet, in our case the packet length,the port number destination and packet's direction.This class could be extended to include other fields.

```
enum mode {in=1,out=0};
class infoPacket
{
private:
        int packet_len;
        mode packet_mode;
        int port;
        string num2str(long int n);
public:
        infoPacket(int len,string mode,int port)
        {
                packet_len=len;
                this->port=port;
                if(mode=="in")packet_mode=in;
                else packet_mode=out;

        }
        string get_mode();
        string get_port_str(){return num2str(port);}
        string get_len_str(){return num2str(packet_len);}


};
```

The class IPstream, is composed by a list of packet_infos and some information about the stream, like the received packet and the sended packet. The user could add some other statistic field, like the average packet lengths and so on.

```
class IPstream:public IPelement
{
private:
        long int num_packets;
        long int num_packets_out;
        long int num_packets_in;
public:
        list<infoPacket> packet_infos;
public:
        IPstream(const struct sockaddr_in* to):IPelement(to)
        {num_packets=0;num_packets_out=0;num_packets_in=0;}
        long int get_num_packets(){return num_packets;}
        void inc(int len,string mode);
        void dec(){num_packets--;}
        long int calc_total_packet(string mode);
        long int get_total_packet(string mode);

        string get_port_str(){return num2str(IPnum->port);}
        string get_num_packets_str(){return num2str(num_packets);}
        string get_total_packet_str(string mode);

};
```

3.2.5 PacketManip

This class is responsible for the manipulation of the UDP packet. The user can implement his own manipulation routines. Two basic method are provided:

```
class PacketManip
{
public:
   char* original_packet;
   bool last_err;
public:
   PacketManip(const char* buf,int len);
   bool RevertByte(char* buf,int b1,int b2);
   bool XORByte(char *buf,int byteindex,int xorvalue);
};
```

The method RevertByte exchange two byte in the UDP packet, while the method XORByte make a classical XOR operation between a byte in the packet and a user provided value. The object could be used during the filtering routine in this way:

```
        if(pmap!=filterlist.end())
        {
          //builda a blank new packet
                char *new_packet=new char[len];
                PacketManip *packetmanip=new PacketManip(buf,len);
                //exchange the byte in position 0 with the last one
                packetmanip->RevertByte(new_packet,0,len-1);
                //xor the first byte with 10 in decimal
                packetmanip->XORByte(new_packet,0,10);
                //send the new packet
                sendto(s,new_packet,len,flags,to,tolen);
        }
```

3.3 The hooked functions

The alghoritm used to store the UDP traffic in the mysendto function is the following

1. create an IPelement object from the current sockaddr_in structure
2. find if the element has to be filtered, if so block the sending actions or modify the payload of the packet
3. search in the hashmap udpstream to check if the receiver has been contacted in the past, it means if it's the first time we send a packet to that address
4. if it's not the first time update the number of outgoing packet in the current stream
5. otherwise generate a new ipstream, check if the receiver is a supernode and insert the new packet
6. call the original sendto api function

The relative code is shown:

```
int  WINAPI mysendto(
  SOCKET s,
  const char* buf,
  int len,
  int flags,
  const struct sockaddr* to,
  int tolen
)
{

        //first build the IPelement object
        IPelement *IPelem=new IPelement((sockaddr_in*)to);

        //search if the element is in the filtered list
        pmap=filterlist.find(IPelem->get_IPnum());

    //OutputDebugStr("Searched string");
    //OutputDebugStr(IPelem->get_IPchar());
        //if the element is in the list do something
        if(pmap!=filterlist.end())
        {
        //        OutputDebugStr("Ipelement was filtered!");
    //    OutputDebugStr(IPelem->get_IPchar());
        }
        else //else log the traffic
        {
                OutputDebugStr("out:Try fo find an udpstream packet!");
                pstream=udpstream.find(IPelem->get_IPnum());

            if(pstream!=udpstream.end())
            {
                  //we count the number of packet sent to this ip
                   OutputDebugStr("Ipstream found in hash map: incremented");
               OutputDebugStr(IPelem->get_IPchar());

                   IPstream udp_curr=pstream->second;
                   udp_curr.inc(len,"out");

                   udpstream.insert(pair<ipaddr_block,IPstream>(pstream->first,udp_curr));
                   OutputDebugStr("out:UDP stream incremented");
                   OutputDebugStr(udp_curr.get_IPchar());

            }
      else
            {

                  //we find if the element is a supernode!
                   IPstream *udp_curr=new IPstream((sockaddr_in*)to);
        pmap=supernodelist.find(IPelem->get_IPnum());
```

```
                        OutputDebugStr("out:Searching_for_supernode");
                        OutputDebugStr(IPelem->get_IPchar());
                    if(pmap!=supernodelist.end())
                        {
                                OutputDebugStr("in:Supernode_found!");
                            udp_curr->isSuperNode=true;
                        }
                  else udp_curr->isSuperNode=false;

                   //added to list
                  udp_curr->inc(len,"out");
                  udpstream.insert(pair<ipaddr_block,IPstream>(udp_curr->get_IPnum(),*udp_curr));
                  OutputDebugStr("out:UDP_stream_added!");
                  OutputDebugStr(udp_curr->get_IPchar());

                  delete udp_curr;

              }
          }

          delete IPelem;
      return sendto(s,buf,len,flags,to,tolen);

}
```

The alghoritm used to store the UDP traffic in the myrecvfrom function is symmetric to the previous one:

1. create an IPelement object from the current sockaddr_in structure
2. find if the element has to be filtered, if so block the sending actions or modify the payload of the packet
3. search in the hashmap udpstream to check if the sender has been contacted in the past, it means if it's the first time we send a packet to that address
4. if it's not the first time update the number of outgoing packet in the current stream
5. otherwise generate a new ipstream, check if the sender is a supernode and insert the new packet
6. call the original recvfrom api function

The corresponding code is the following:

```
int  WINAPI myrecvfrom(
  SOCKET s,
  char* buf,
  int len,
  int flags,
  struct sockaddr* from,
  int* fromlen
)
  {
//we build the object ip
        IPelement *IPelem=new IPelement((sockaddr_in*)from);

        //we find if the element is to be filtered!
        pmap=filterlist.find(IPelem->get_IPnum());

 //    OutputDebugStr("Searched string");
//       OutputDebugStr(IPelem->get_IPchar());
        //we can put something here to make some filter implementation
        if(pmap!=filterlist.end())
        {
        //        OutputDebugStr("Ipelement was filtered!");
    //       OutputDebugStr(IPelem->get_IPchar());
        }
```

```
        else
        {
                OutputDebugStr("in:Try fo find an udpstream packet!");
                pstream=udpstream.find(IPelem->get_IPnum());

                //in this case we must add a new element
            if(pstream!=udpstream.end())
            {
                  //we count the number of packet sent to this ip
                   OutputDebugStr("Ipstream found in hash map: incremented");
               OutputDebugStr(IPelem->get_IPchar());

                   IPstream udp_curr=pstream->second;
                   udp_curr.inc(len,"in");

                   udpstream.insert(pair<ipaddr_block,IPstream>(pstream->first,udp_curr));
                   OutputDebugStr("in:UDP_stream_incremented");
                   OutputDebugStr(udp_curr.get_IPchar());

            }
            else
            {

                  //we find if the element is a supernode!
                  IPstream *udp_curr=new IPstream((sockaddr_in*)from);
    pmap=supernodelist.find(IPelem->get_IPnum());

                        OutputDebugStr("in:Searching for supernode:");
                        OutputDebugStr(IPelem->get_IPchar());
                  if(pmap!=supernodelist.end())
                        {
                                OutputDebugStr("in:Supernode found!");
                            udp_curr->isSuperNode=true;
                        }
          else udp_curr->isSuperNode=false;

                  //added to list
                 udp_curr->inc(len,"in");
                 udpstream.insert(pair<ipaddr_block,IPstream>(udp_curr->get_IPnum(),*udp_curr));
                 OutputDebugStr("UDP_stream_added!");
                 OutputDebugStr(udp_curr->get_IPchar());

          delete udp_curr;

          }
        }

        delete IPelem;
return recvfrom(s,buf,len,flags,from,fromlen);

}
```

Chapter 4

Skype documentation and used tools

The tools I used for the logger are:

- WinAPI32override by Jacqueline Potier
- A small and fast XMl parsing library written by Frank Vanden Berghen
- Visual Studio 7
- some utility for exploring the dll
- NtDebugView for debugging functions
- Ethereal for the network analysis

The papers about the skype peer2peer architecture and the reversing stuff is:

- Silver Nidle Skype
- Skype P2P analysis

Both document are included in the attached zip. The Weka program is a data mining analysis tools, so I decided to export the traffic in this format. The traffic is saved in this format:

```
%Skype UDP stream log stats

@relation UDP_traffic

@attribute UDP_IP_byte1 integer

@attribute UDP_IP_byte2 integer

@attribute UDP_IP_byte3 integer

@attribute UDP_IP_byte4 integer

@attribute UDP_IP_total_packet integer

@attribute UDP_IP_total_packet_in integer

@attribute UDP_IP_total_packet_out integer

@attribute UDP_IP_port integer
```

```
@attribute UDP_IP_is_super_node {Y,N}

@data

 83,83,24,16,120,20,100,39689,Y

 24,26,195,73,54,40,14,53103,Y

 216,16,91,219,10,5,5,161420,Y

 66,183,244,98,1,1,0,24050,Y

 142,151,186,232,1,1,0,37186,Y

 24,211,62,61,1,1,0,58116,Y

 70,82,138,182,4,2,2,19871,Y
 .... others ....
```

Then it's possible to open the arff file and visualize the traffic with Weka Explorer. In this case we have 7 connection to supernodes (see 4.1): And we can

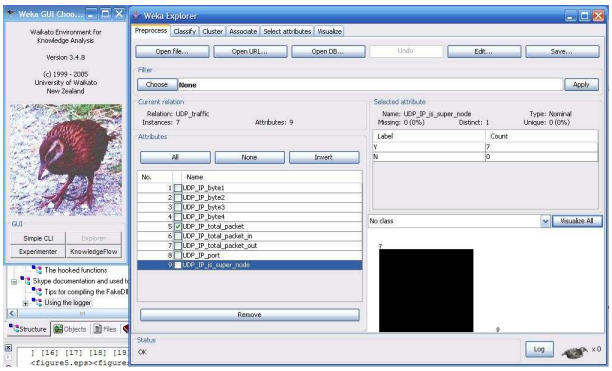


Figure 4.1: Weka Anaylisis

categorize the data (see 4.2):

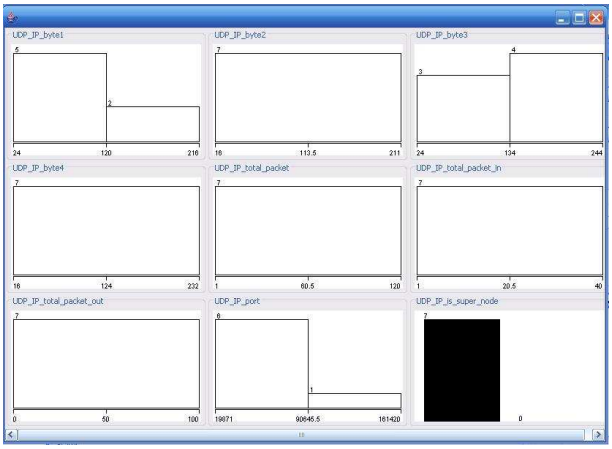


Figure 4.2: Category

4.1 Tips for compiling the FakeDll

Remember that all the path I used to find the filter condifugration or the skype shared.xml are **absolute** so you have to change them according to your directory structure.

4.2 Using the logger

After having compiled the fakedll, start the WinAPIOverride32 interface, and attach to the running skype process. Choose the skype faking dll in the API faking configuration box then load it. The traffic will be saved on the files **only** if you unload properly the dll (see 4.2).

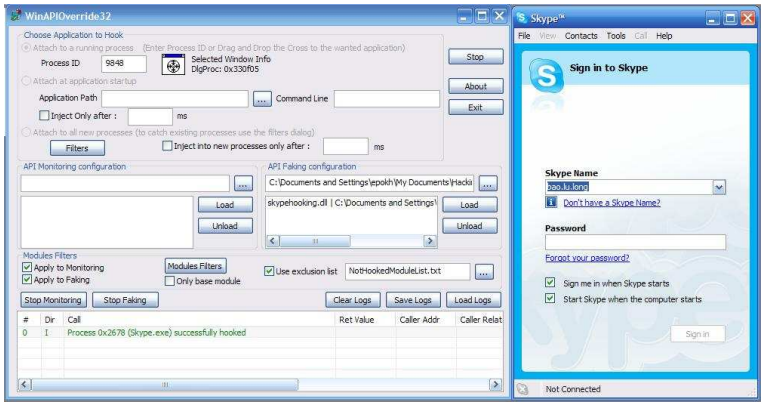


Figure 4.3: Skype Process Hooking

Chapter 5

Special thanks and disclaimer

Thanks a lot to Jacqueline for his support! Thanks to quequero for his leadership!

5.0.1 Disclaimer

The author doesn't take any responsibility on the improper use of the described framework.