

UNIVERSITATEA DIN BUCUREȘTI
FACULTEA DE MATEMATICĂ ȘI INFORMATICĂ

LUCRARE DE LICENȚĂ

Malware Monitor

COORDONATOR ȘTIINȚIFIC

Prof. Dr. Andrei Păun

STUDENT

Ionuț Gabriel Popescu

IUNIE 2014

Cuprins

1. INTRODUCERE	3
1.1. Ce este Malware Monitor?.....	3
1.2. Soluții alternative	4
1.3. Cum funcționează?.....	7
1.4. Utilizarea într-un penetration test	9
1.5. Perspective	10
2. FORMATUL FIȘIERELOR PE	11
2.1. Informații generale.....	11
2.2. Structura generală a fișierelor PE.....	12
2.3. Antetul MS-DOS	14
2.4. Antetul PE.....	20
2.5. Tabelul de secțiuni	31
2.6. Funcțiile importate si funcțiile exportate.....	36
3. INJECTAREA UNUI DLL.....	43
3.1. Introducere.....	43
3.2. Cum funcționează?.....	44
3.3. Funcțiile API folosite.....	45
3.4. Rezumat	53
4. INTERCEPTAREA APELURILOR DE FUNCȚII	55
4.1. Introducere.....	55
4.2. Cum funcționează?.....	56
4.3. Apelurile de funcții	58
4.4. Interceptarea apelurilor	61
5. CONCLUZII	68
6. BIBLIOGRAFIE	69

1. INTRODUCERE

1.1. Ce este Malware Monitor?

Malware Monitor este o aplicație destinată sistemelor de operare Microsoft Windows foarte utilă atât pentru monitorizarea unui malware: virus, troian sau program spion, cât și pentru monitorizarea oricărei alte aplicații pentru o mai bună înțelegere a modului de funcționare a acesteia.

Aplicația oferă posibilitatea monitorizării unui fișier executabil dar și monitorizarea unui proces deja existent. Astfel, dacă un utilizator a executat din greșeală un program posibil malițios, acesta poate folosi Malware Monitor pentru a analiza procesul respectiv: pentru a vedea ce fișiere accesează și modifică, a verifica ce date trimite pe Internet chiar dacă acestea sunt criptate sau să vadă ce date modifică programul respectiv în baza de date Windows numită Windows Registry etc.

Mai mult, Malware Monitor permite utilizatorului să modifice datele returnate de către funcțiile apelate de programele monitorizate. Astfel, dacă un virus încearcă să se conecteze la un website pentru a fura date bancare de pe calculatorul infectat și pentru a le trimite către acel website, Malware Monitor asemenea unui firewall, permite blocarea conexiunii către acel website. De asemenea, dacă virusul vrea să se copieze într-o altă locație, utilizatorul va putea bloca această acțiune.

Ce poate monitoriza:

- Accesul, citirea și scrierea fișierelor
- Conexiunile realizate și transferul de date
- Traficul necriptat, deși în rețea va fi trimis criptat
- Citirea și modificarea datelor din Registry

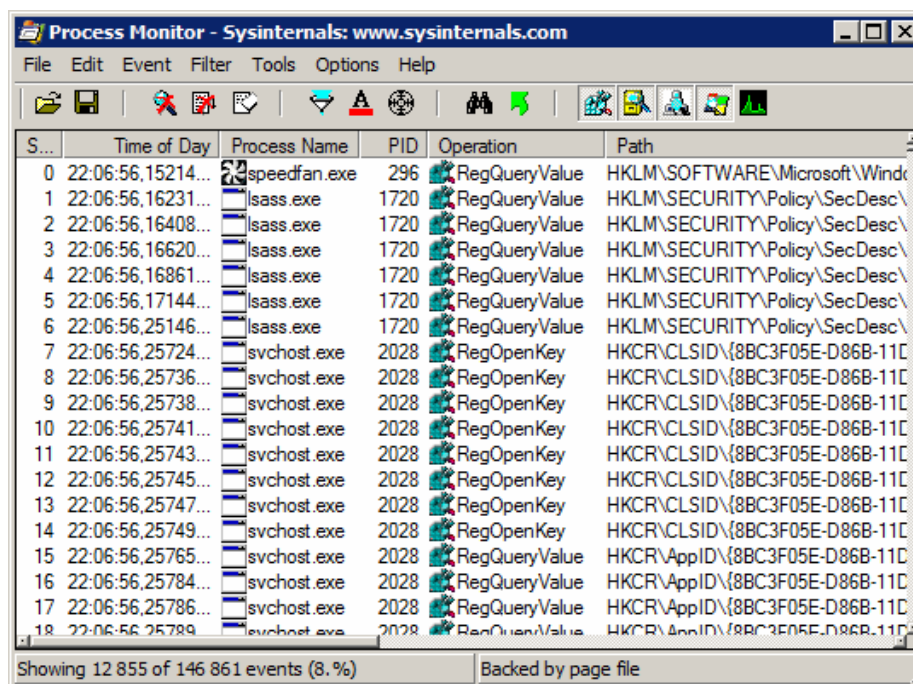
Pe scurt, este un fel de firewall pentru aplicații desktop.

1.2. Soluții alternative

Există deja câteva aplicații care funcționează asemănător însă nu există nicio aplicație care să facă același lucru. Cele mai importante aplicații care trebuie menționate sunt:

- **Process Monitor**¹: o aplicație din suita Microsoft Sysinternals care capturează fiecare apel de sistem realizat de anumite procese. Datele capturate sunt extrem de multe și foarte greu de înțeles, de aceea este destinat în special utilizatorilor avansați. Dezavantajul principal este acest număr mare de apeluri efectuate, majoritatea nefiind utile pentru o analiză a unui executabil.
- **API Monitor**²: există mai multe aplicații care fac acest lucru, cea mai stabilă dintre ele fiind RohitAB API Monitor. Aceste aplicații monitorizează toate apelurile de funcții WinAPI, funcții ale sistemului de operare Windows implementate în mai multe biblioteci de funcții (DLL) ca: user32.dll, advapi32.dll, kernel32.dll și multe altele. Dezavantajul principal este că orice program, oricât de simplu ar părea, apelează foarte multe funcții WinAPI în fundal, iar o analiză folosind un astfel de program este mult prea complicată și de foarte lungă durată.

Process Monitor:



¹ Process Monitor - <http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx>

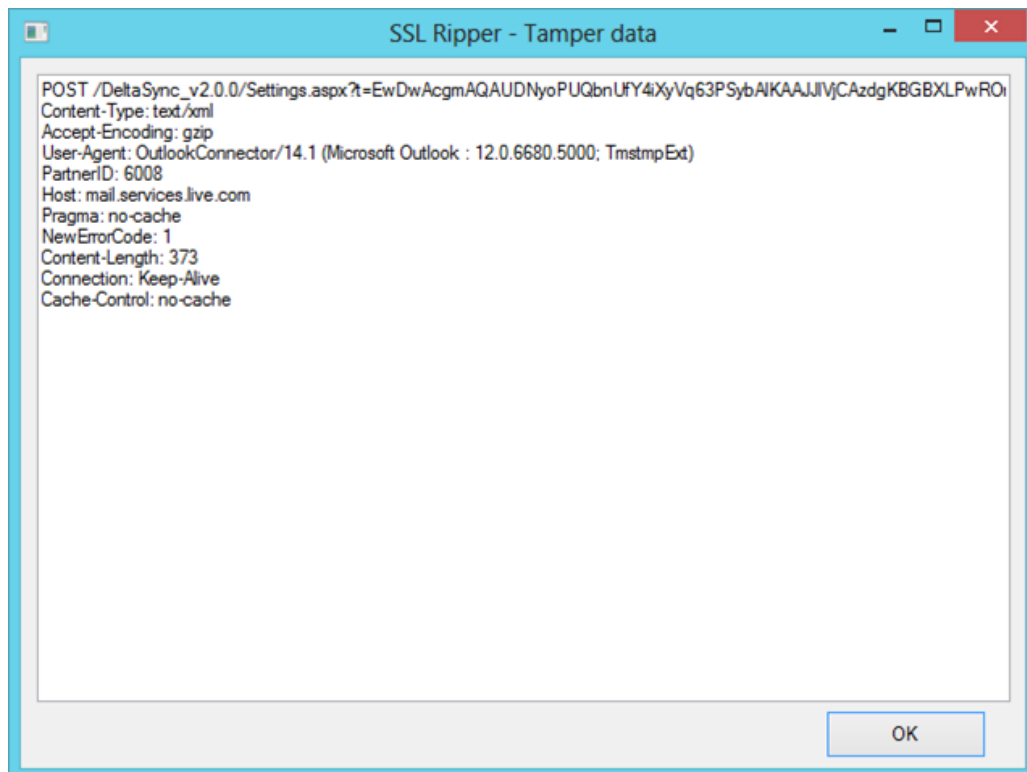
² API Monitor - <http://www.rohitab.com/apimonitor>

API Monitor:

Summary 19 of 303,106 calls 99% dropped 449.20 MB used SecureCRT.exe					
#	Thread	Module	API	Return Value	Duration
130413	2	mf90u.dll	TlsSetValue (6, 0x0000000000725070)	TRUE	0.0000010
130444	2	mf90u.dll	WaitForSingleObject (0x0000000000000140, INFINITE)	WAIT_OBJECT_0	0.0014443
130446	2	KERNELBASE.dll	NtWaitForSingleObject (0x0000000000000140, FALSE, NULL)	STATUS_SUCCESS	0.0014412
130448	1	mf90u.dll	SuspendThread (0x0000000000000144)	0	0.0000082
130449	1	KERNELBASE.dll	NtSuspendThread (0x0000000000000144, 0x000000000020f398)	STATUS_SUCCESS	0.0000051
130460	2	SecureCRT.exe	GetQueuedCompletionStatus (0x0000000000000138, 0x0000000000548fb20, 0x...		
130462	2	KERNELBASE.dll	NtRemoveIoCompletion (0x0000000000000138, 0x0000000000548fb30, 0x00...		
130483	1	MSVCR90.dll	CreateThread (NULL, 0, 0x0000000072b92ffc, 0x00000000004de1a50, CREATE_S...	0x000000000000014c	0.0001134
130484	1	KERNELBASE.dll	NtCreateThreadEx (0x00000000000020ee98, DELETE READ_CONTROL SYNCH...		0.0000385
130486	1	KERNELBASE.dll	RtlQueryInformationActivationContext (1, NULL, NULL, 1, 0x00000000000020e...	STATUS_SUCCESS	0.0000015
130487	1	KERNELBASE.dll	CsrClientCallServer (0x00000000000020f150, NULL, 65537, 24)	STATUS_SUCCESS	0.0000626
130490	1	mf90u.dll	WaitForSingleObject (0x0000000000000140, INFINITE)	WAIT_OBJECT_0	0.0104424
130491	3	ntdll.dll	DllMain (0x00000000771a0000, DLL_THREAD_ATTACH, NULL)	TRUE	0.0000015
130492	3	ntdll.dll	DllMain (0x0000007fefdfa0000, DLL_THREAD_ATTACH, NULL)	TRUE	0.0000015
130493	3	ntdll.dll	DllMain (0x0000007fef61b0000, DLL_THREAD_ATTACH, NULL)	TRUE	0.0000021
130494	3	ntdll.dll	DllMain (0x0000007fefd90000, DLL_THREAD_ATTACH, NULL)	TRUE	0.0000031
130495	3	ntdll.dll	DllMain (0x0000007fefe2a0000, DLL_THREAD_ATTACH, NULL)	TRUE	0.0000221
130498	1	KERNELBASE.dll	NtWaitForSingleObject (0x0000000000000140, FALSE, NULL)	STATUS_SUCCESS	0.0104378
130504	3	ntdll.dll	DllMain (0x00000000669b0000, DLL_THREAD_ATTACH, NULL)	TRUE	0.0000021

După cum se poate observa, ambele aplicații oferă un număr imens de date, deloc utile când se dorește analiza unui executabil.

Malware Monitor, în exemplul de mai jos, afișează datele utile unui utilizator care vrea să înțeleagă ce face o aplicație, în cazul de față, verifică traficul SSL, înainte de a fi criptat și trimis către Internet, de aplicația Microsoft Outlook.



Există de asemenea și aplicații de tipul "sandbox"³, aplicații care permit execuția unui proces într-un spațiu izolat, astfel aplicațiile monitorizate nu au dreptul de a accesa sistemul de fișiere de pe calculator sau să interacționeze cumva cu sistemul de operare al utilizatorului. Însă aceste aplicații doar previn execuția programelor malițioase și nu oferă prea multe detalii despre modul de funcționare al acestor programe. Altfel spus, un utilizator nu își poate da seama dacă un program pe care îl execută este infectat sau dacă nu este nicio problemă în a fi executat.

Un alt avantaj, așa cum este prezentat în detaliu mai jos, îl reprezintă faptul că o versiune limitată a acestei aplicații poate fi folosită de către penetration testerii în proiectele acestora.

Desigur, de pe urma aplicației vor avea cel mai mult de câștigat persoanele care se ocupă cu studierea comportamentului fișierelor malițioase, persoane care de cele mai multe ori lucrează la companii care produc software antivirus. Aplicația va ajuta în principal pentru o primă analiză extrem de rapidă a unui program posibil malițios, fără a fi nevoie ca respectiva persoană să dezasambleze și să ruleze programul într-un debugger. Pur și simplu va deschide programul malițios folosind Malware Monitor și va înțelege cel puțin o parte din funcționalitatea acestuia.

³ Sandbox (computer security) - http://en.wikipedia.org/wiki/Sandbox_%28computer_security%29

1.3. Cum funcționează?

Malware monitor are două componente:

- Malware monitor.exe - Programul cu care interacționează utilizatorul
- InjectedDLL.dll - O bibliotecă de funcții care este injectată în procesele monitorizate

Programul Malware monitor.exe are rolul principal de a interacționa cu utilizatorul: dispune de o interfață grafică unde utilizatorul are posibilitatea de a selecta un proces sau un fișier executabil și injectează componenta InjectedDLL.dll. Pentru a putea accesa alte procese și pentru a putea executa cod în cadrul altor procese, deoarece fiecare proces rulează în propriul spațiu de adrese, cea mai simplă și eficientă metodă este de a crea o bibliotecă de funcții și de a o injecta în procesul respectiv.

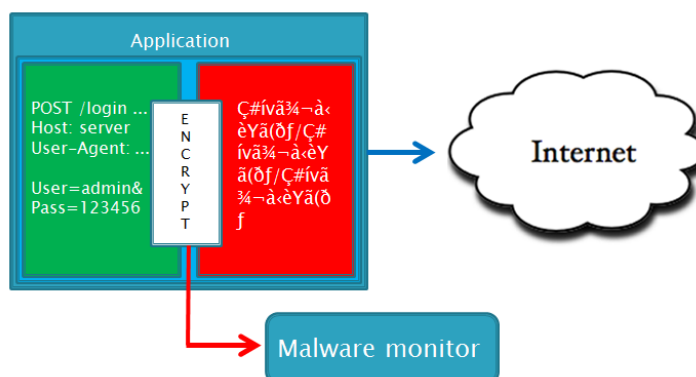
Biblioteca InjectedDLL.dll este componenta care se ocupă de monitorizarea proceselor. O dată injectată și apelată funcția principală din această bibliotecă, sunt puse "hook-uri" pe funcții importante apelate de obicei de către programe pentru a interacționa cu sistemul de operare, funcții WinAPI, adică funcții din DLL-urile sistemului de operare Windows, sau funcții create de autorii altor biblioteci ca OpenSSL.dll.

De exemplu, pentru a scrie un fișier, de cele mai multe ori, un program va ajunge să apeleze funcția WriteFile din biblioteca "kernel32.dll" a sistemului de operare. Chiar dacă un program apelează de fapt funcția "fwrite", aceasta rezultă ulterior într-un apel al funcției WriteFile.

Hook-urile sunt de fapt niste "redirecționări" ale apelurilor de funcții. Mai exact, un hook pus pe o funcție, va permite redirecționarea apelului de funcție către o altă funcție definită în biblioteca InjectedDLL.dll, care ulterior va apela funcția originală și va returna rezultatul, având astfel posibilitatea atât de a putea observa parametrii cu care a fost apelată funcția respectivă, în cazul de mai sus datele care vor fi scrise în fișier, cât și de a modifica valoarea returnată de funcția respectivă. Astfel se pot monitoriza și bloca anumite apeluri de funcții.

Utilizatorul va putea observa funcțiile apelate într-un fișier text, dar va putea bloca și anumite acțiuni pe care dorește programul malițios să le facă.

Exemplu de monitorizare a datelor trimise criptat către Internet, înainte de a fi criptate, pentru a putea înțelege exact ce dorește să facă un program:

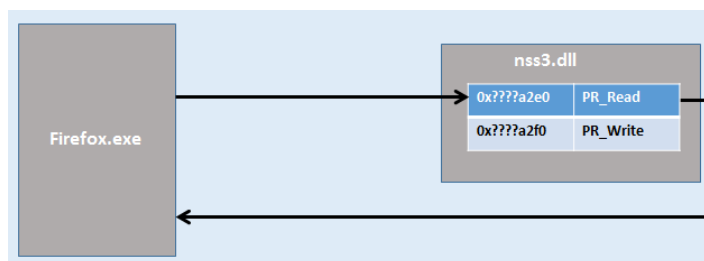


Pentru a putea înțelege pe deplin cum funcționează, e necesară cunoașterea următoarelor concepte:

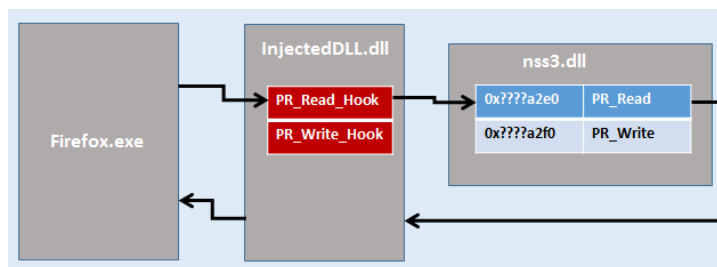
- Formatul fișierelor PE (Portable Executable)
- Injectarea unui DLL (biblioteca de funcții)
- Interceptarea apelurilor de funcții (numită și API Hooking)

De exemplu, pentru a trimite date pe Internet, fie că sunt criptate sau nu, aplicația Mozilla Firefox apelează funcțiile PR_Read si PR_Write.

Așa arată un astfel de apel obișnuit de funcție:



Și așa arată un apel de funcție interceptat:



Astfel, traficul este capturat înainte de a fi criptat.

1.4. Utilizarea într-un penetration test⁴

Malware monitor poate fi folosit și de către penetration testerii (whitehat hackers) profesioniști în cadrul proiectelor de tipul "pentest intern".

Pentest-ul intern presupune deplasarea pentesterilor la sediul clientului sau într-o locație de unde pot accesa rețeaua internă a clientului și conectarea laptopurilor la rețea. Astfel, se încearcă obținerea accesului la sistemele din rețea, de la calculatoare ale utilizatorilor la servere importante, din perspectiva unui atacator intern, un atacator care are deja acces la rețea: un angajat al firmei sau un partener.

De cele mai multe ori, la astfel de proiecte, un atacator obține acces la anumite calculatoare din rețea și folosește accesul obținut pentru a obține acces mai departe la alte servere din cadrul rețelei interne a companiei.

De exemplu, după obținerea accesului la calculatorul unui alt angajat al firmei, poate chiar un angajat din departamentul de IT, atacatorul poate căuta diverse fișiere cu parole pe calculatorul acestuia, poate citi mail-ul sau poate întreprinde orice alte acțiuni ale căror urmări vor fi obținerea accesului la serverele la care are acces angajatul respectiv.

Malware Monitor are integrată funcția de capturare a traficului care în mod normal este transmis criptat în rețea, dar aplicația îl capturează înainte de a fi criptat. Astfel, folosind o versiune limitată a acestei aplicații, un penetration tester poate obține traficul securizat al angajatului la care a obținut acces, către diverse servere din cadrul rețelei.

Mai exact, un penetration tester poate obține traficul ncriptat generat de aplicații ca: Microsoft Outlook (email-urile angajatului), Internet Explorer (parole sau cookie-uri de logare), Putty (date de conectare pe server) etc.

A nu se confunda munca unui penetration tester cu ceea ce face atacatorul! Un penetration tester este angajat de companie tocmai pentru a descoperi astfel de probleme în rețea și de a ajuta la repararea acestora. El nu va face decât să imite un posibil atacator, astfel, dacă el poate obține acces la toate calculatoarele din rețea, indiferent de metodele folosite, înseamnă că și un alt angajat al firmei sau un partener, având cunoștințele necesare, poate obține acces la toate serverele interne, servere care pot conține de la date personale ale clienților la date bancare ale acestora.

⁴ Penetration test - http://en.wikipedia.org/wiki/Penetration_test

1.5. Perspective

Deocamdata aplicația este limitată în câteva privințe:

- Funcționează doar pe sistemele pe 32 de biți
- Nu suportă foarte multe aplicații
- Nu este thread safe⁵

Pe lângă rezolvarea acestor limitări, pe viitor, aplicația va avea o funcționalitate dezvoltată:

- Va funcționa chiar dacă este folosit programul EMET⁶
- Va avea o interfață grafică dezvoltată
- Va avea posibilitatea de a modifica datele capturate
- Va avea un modul de Metasploit⁷

Fără a intra în detalii, EMET (Enhanced Mitigation Experience Toolkit) este un program realizat de către Microsoft, având ca scop protejarea programelor împotriva atacurilor de tip Buffer Overflow, Use after free și altele. Astfel, modificând direct codul anumitor procese, le poate proteja împotriva acestor atacuri, însă poate determina o execuție eronată a aplicației Malware Monitor.

Metasploit este probabil cel mai cunoscut framework din lume, folosit de către penetration testerii sau alte persoane care lucrează în industria "IT security" în proiectele la care participă. Framework-ul este modular, iar modulele sunt împărțite pe categorii. Consider că și Malware Monitor, sau cel puțin o funcționalitate limitată a acestei aplicații, ar merita un astfel de modul, pentru a putea fi integrat și folosit cu ușurință de către penetration testerii din întreaga lume.

⁵ Thread safety - http://en.wikipedia.org/wiki/Thread_safety

⁶ The Enhanced Mitigation Experience Toolkit - <http://support.microsoft.com/kb/2458544>

⁷ Metasploit Project - http://en.wikipedia.org/wiki/Metasploit_Project

2. FORMATUL FIȘIERELOR PE

2.1. Informații generale

Portable Executable⁸ este formatul de fișiere folosit de sistemul de operare Microsoft Windows pentru fișierele executabile și alte tipuri de fișiere care conțin cod executabil, cod masină - care este executat direct de către procesor.

Acest format este folosit în special de către fișierele executabile cu extensia “.exe” și de către bibliotecile de funcții ce sunt încărcate de către aceste executabile având extensia “.dll” dar și de alte tipuri de fișiere cum ar fi controalele ActiveX (extensia “.ocx”) sau driverele/modulele sistemului de operare (extensia “.sys”).

Numele de “Portable” provine de la faptul că formatul este același pentru mai multe arhitecturi: x86, x86-64, IA32 și IA64, este “portabil”. Practic acest format este o versiune modificată a formatului COFF (Common Object File Format) folosit de sistemele UNIX în timp ce sistemele Linux folosesc formatul ELF (Executable and Linkable Format).

Acest format a apărut pentru prima oară odată cu sistemul de operare Windows NT 3.1 și a fost creat pentru a păstra compatibilitatea cu vechiul format, MS-DOS, sistemul anterior sistemului Windows.

⁸ Portable Executable - http://en.wikipedia.org/wiki/Portable_Executable

2.2. Structura generală a fișierelor PE

Din motive de compatibilitate cu formatul folosit de MS-DOS⁹, fișierele PE conțin practic un mic program pentru MS-DOS care afișează mesajul: *“This program cannot be run in DOS mode.”*, acesta fiind singurul lucru pe care îl face.

Formatul folosește mai multe anteturi/headere pentru a defini datele folosite. Așadar fiecare fișier PE va conține atât un header MS-DOS necesar programului MS-DOS, cât și un header PE. Headerele sunt un grup de octeți, număr fix sau variabil, care definesc datele ce vor urma, niște octeți care oferă informații despre structurile ce urmează, structuri pe care le definesc.

Pentru o detaliere mai simplă a acestui format voi afișa definițiile acestor headere ca structuri în C/C++. Astfel se vor putea observa mai ușor dimensiunile fiecărui câmp din structuri. Structurile pot fi regăsite în SDK-uri¹⁰ în fișierul “WinNT.h”.

Ca structură generală un fișier PE este destul de simplu, este alcătuit din programul MS-DOS (denumit adesea “stub”) care e alcătuit dintr-un header MS-DOS și programul MS-DOS, din headerele PE (vom vedea că nu este unul singur) care include și tabelul de secțiuni, urmate de secțiunile propriu-zise, de datele codul programului nostru. Pe înțelesul tuturor, primii octeți dintr-un fișier PE îl reprezintă un mic program MS-DOS, urmat de niște headere care definesc datele ce urmează, urmate de datele propriu-zise ale fișierului PE: secțiuni de date și de cod.

Structura fișierelor PE

Header MS-DOS
Program MS-DOS
Header PE
Header opțional PE
Tabel secțiuni
Secțiunea 1
Secțiunea 2
...
Secțiunea n

⁹ MS-DOS - <http://en.wikipedia.org/wiki/MS-DOS>

¹⁰ Windows Software Development Kit - <http://msdn.microsoft.com/en-us/windows/bg162891.aspx>

Așadar prima parte a oricărui fișier executabil o reprezintă un mic program MS-DOS, adesea în jur de 240 de octeți. Deschis într-un Hex Editor¹¹, acest program MS-DOS compus din header și cod arată cam așa:

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZÿÿ..
00000010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	f0	00	00	008...
00000040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	...°...'í!¸.Lí!Th
00000050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program cannc
00000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
00000070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....\$.....
00000080	63	8a	9f	9f	27	eb	f1	cc	27	eb	f1	cc	27	eb	f1	cc	cŠŸŸ'ëñì'ëñì'ëñì
00000090	2e	93	62	cc	16	eb	f1	cc	27	eb	f0	cc	55	e8	f1	cc	.`bì.ëñì'ëñìUëñì
000000a0	2e	93	63	cc	26	eb	f1	cc	2e	93	64	cc	20	eb	f1	cc	.`cìëñì.`dì ëñì
000000b0	2e	93	72	cc	d1	eb	f1	cc	2e	93	75	cc	c4	eb	f1	cc	.`rìÑëñì.`uìÄëñì
000000c0	2e	93	65	cc	26	eb	f1	cc	2e	93	60	cc	26	eb	f1	cc	.`eìëñì.`ìëñì
000000d0	52	69	63	68	27	eb	f1	cc	00	00	00	00	00	00	00	00	Rich'ëñì.....
000000ef	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

¹¹ Hex editor - http://en.wikipedia.org/wiki/Hex_editor

2.3. Antetul MS-DOS

Headerul care conține informații despre acest mic program este o structură numită *IMAGE_DOS_HEADER*. Structura are dimensiunea de 64 de octeți și ca orice header conține informații despre programul MS-DOS, informații necesare pentru ca programul să poată fi executat cu succes. Practic datele din fișier se mapează peste această structură, astfel, primii 2 octeți dintr-un fișier PE vor fi reprezentați de câmpul “e_magic” din structură, următorii 2 de “e_cblp” și tot așa. Așadar, primii 64 de octeți îi reprezintă un header care conține informațiile definite mai jos:

```
typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD e_magic; // Magic number
    WORD e_cblp; // Bytes on last page of file
    WORD e_cp; // Pages in file
    WORD e_crlc; // Relocations
    WORD e_cparhdr; // Size of header in paragraphs
    WORD e_minalloc; // Minimum extra paragraphs needed
    WORD e_maxalloc; // Maximum extra paragraphs needed
    WORD e_ss; // Initial (relative) SS value
    WORD e_sp; // Initial SP value
    WORD e_csum; // Checksum
    WORD e_ip; // Initial IP value
    WORD e_cs; // Initial (relative) CS value
    WORD e_lfarlc; // File address of relocation table
    WORD e_ovno; // Overlay number
    WORD e_res[4]; // Reserved words
    WORD e_oemid; // OEM identifier (for e_oeminfo)
    WORD e_oeminfo; // OEM information; e_oemid specific
    WORD e_res2[10]; // Reserved words
    LONG e_lfanew; // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Unul dintre lucrurile de bază pe care trebuie să le cunoașteți atunci când programați pentru Windows îl reprezintă tipurile de date¹². Astfel avem:

- *BYTE* – Un număr pe 8 biți (1 byte), definit ca “unsigned char”
- *CHAR* – Un număr pe 8 biți (1 byte), definit ca “char”
- *DWORD* – Un număr pe 4 bytes (32 de biți) fără semn, definit ca “unsigned long”
- *LONG* – Un număr pe 4 bytes (32 de biți) cu semn, definit ca “long”
- *ULONGLONG* – Un număr pe 8 bytes (64 de biți) fără semn, “unsigned long long”
- *WORD* – Un număr pe 2 bytes (16 biți) fără semn, definit ca “unsigned short”

¹² Windows Data Types (Windows) - <http://msdn.microsoft.com/en-us/library/windows/desktop/aa383751%28v=vs.85%29.aspx>

Practic avem nevoie să știm aceste informații deoarece fișierele PE se mapează perfect peste aceste structuri. Astfel, dacă vom citi primii 64 de bytes dintr-un executabil, aceștia vor fi exact acest header, iar dacă vom citi 64 de bytes într-o astfel de structură vom putea avea acces simplu și rapid la fiecare dintre câmpurile acestui header.

Astfel, primii 2 octeți ai fiecărui fișier PE vor reprezenta câmpul `e_magic` (un WORD are 2 octeți), următorii 2 octeți vor fi `e_cblp` și tot așa.

Să citim o astfel de structură și să vedem cum arată. Am scris un mic program care citește un fișier PE, “kernel32.dll”¹³ și afișează informațiile din headerul său PE.

Vom vedea că majoritatea câmpurilor au valoarea 0 deoarece nu sunt folosite. Pe noi ne interesează doar două dintre aceste câmpuri: `e_magic` și `e_lfanew`.

¹³ Microsoft Windows library files - http://en.wikipedia.org/wiki/Microsoft_Windows_library_files

```

#include <stdio.h>
#include <windows.h>

int main()
{
    FILE *FD = NULL;
    IMAGE_DOS_HEADER dos;
    size_t Len = 0;

    // Open file

    FD = fopen("C:\\Windows\\system32\\kernel32.dll", "rb");

    if(!FD)
    {
        puts("Cannot open file!");
        return 1;
    }

    // Read file

    Len = fread(&dos, 1, sizeof(dos), FD);

    if(Len != sizeof(dos))
    {
        puts("Cannot read file!");
        return 1;
    }

    fclose(FD);

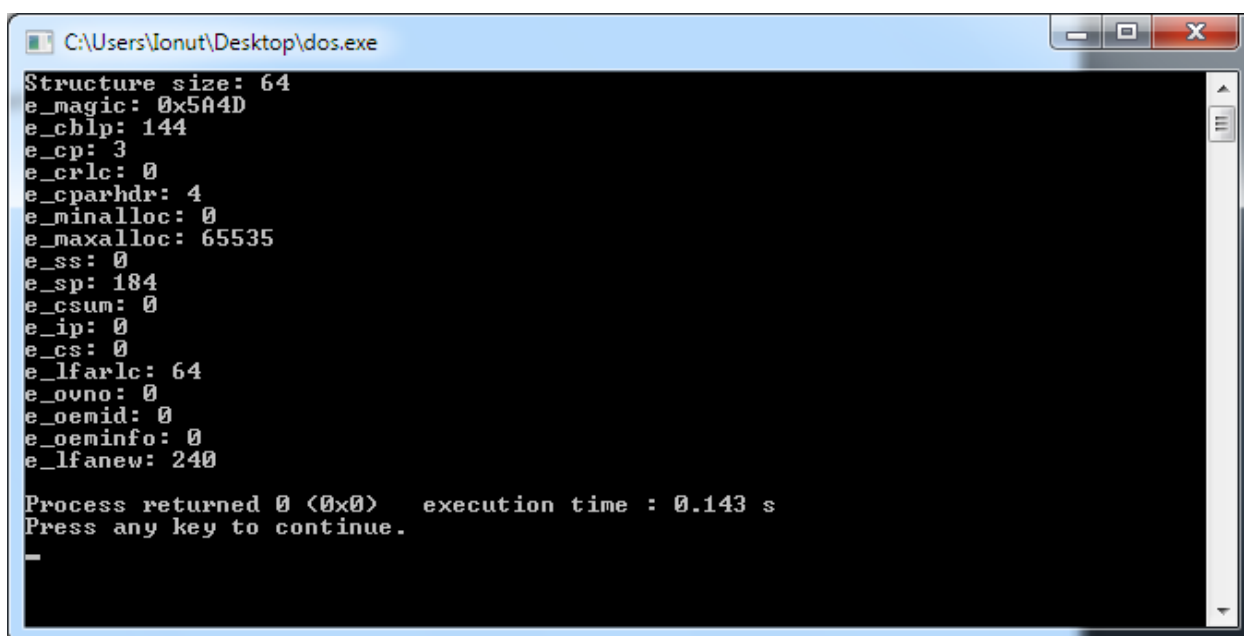
    // Print information

    printf("Structure size: %d\n", sizeof(dos));
    printf("e_magic: 0x%X\n", dos.e_magic);
    printf("e_cblp: %d\n", dos.e_cblp);
    printf("e_cp: %d\n", dos.e_cp);
    printf("e_crlc: %d\n", dos.e_crlc);
    printf("e_cparhdr: %d\n", dos.e_cparhdr);
    printf("e_minalloc: %d\n", dos.e_minalloc);
    printf("e_maxalloc: %d\n", dos.e_maxalloc);
    printf("e_ss: %d\n", dos.e_ss);
    printf("e_sp: %d\n", dos.e_sp);
    printf("e_csum: %d\n", dos.e_csum);
    printf("e_ip: %d\n", dos.e_ip);
    printf("e_cs: %d\n", dos.e_cs);
    printf("e_lfarlc: %d\n", dos.e_lfarlc);
    printf("e_ovno: %d\n", dos.e_ovno);
    printf("e_oemid: %d\n", dos.e_oemid);
    printf("e_oeminfo: %d\n", dos.e_oeminfo);
    printf("e_lfanew: %lu\n", dos.e_lfanew);

    return 0;
}

```


O execuție a acestui program ar produce rezultate asemănătoare:



```
C:\Users\Ionut\Desktop\dos.exe
Structure size: 64
e_magic: 0x5A4D
e_chlp: 144
e_cp: 3
e_crlc: 0
e_cparhdr: 4
e_minalloc: 0
e_maxalloc: 65535
e_ss: 0
e_sp: 184
e_csum: 0
e_ip: 0
e_cs: 0
e_lfarlc: 64
e_ovno: 0
e_oemid: 0
e_oeminfo: 0
e_lfanew: 240

Process returned 0 (0x0)   execution time : 0.143 s
Press any key to continue.
```

Câmpul `e_magic`, primii doi octeți ai fiecărui fișier PE, reprezintă o “semnătură”, un identificator care ne spune că avem un fișier MS-DOS. Dacă ați deschis vreodată un executabil în Notepad ați observat că începe cu literele “MZ”. Aceste litere provin de la numele inginerului Microsoft - Mark Zbykowski.¹⁴

Pentru programatori, aceste litere apar în fișier ca fiind octeții `0x4D` și `0xCA`. De observat că în screenshot-ul de mai sus, numărul este `0x5A4D`, deoarece procesoarele Intel sunt little-endian¹⁵ și în memorie numerele sunt memorate în ordinea inversă a octeților. În `WinNT.h`:

```
#define IMAGE_DOS_SIGNATURE          0x5A4D      // MZ
```

Câmpul `e_lfanew` este cel care ne interesează în mod special, deoarece acesta ne indică unde se termină programul MS-DOS și unde începe headerul PE. Astfel vom putea sări peste acest program și vom ajunge la ceea ce ne interesează.

¹⁴ MZ - <http://wiki.osdev.org/MZ>

¹⁵ Endianness - <http://en.wikipedia.org/wiki/Endianness>

Însă ce face acest “stub” (cum este numit acest mic program) MS-DOS? Cum afișează acel mesaj? Conține cod?

După cum spuneam mai sus, headerul MS-DOS are 64 de octeți, așadar programul MS-DOS va începe imediat după acest header, de la octetul 0x40 (64 în zecimal).

În imaginea de mai jos se poate vedea porțiunea care afișează acel mesaj și mesajul.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZ
00000010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	f0	00	00	008...
00000040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	..°..'.Í!..LÍ!Th
00000050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program cannc
00000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
00000070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....\$.....

Partea selectată va afișa acel mesaj. Primii octeți sunt:

0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd
--

Dacă vom dezasambla acest cod (cu un disassembler pe 16 biți, NASM¹⁶ de exemplu) vom obține:

```

505 C:\Program Files\NASM>ndisasm.exe -b 16 ms-dos.dll
506 00000000 0E                push cs
507 00000001 1F                pop ds
508 00000002 BA0E00            mov dx,0xe
509 00000005 B409            mov ah,0x9|
510 00000007 CD21            int 0x21

```

Primele două linii vor pune în registrul “ds” (data segment) valoarea registrului “cs” (code segment), ceea ce va indica faptul că datele se află în același segment ca și codul.

¹⁶ The Netwide Assembler - <http://www.nasm.us/>

Următoarele două linii vor seta registrul “dx” la valoarea 0xE și registrul “ah” la valoarea 0x9 urmând apoi instrucțiunea “int 0x21”. Această instrucțiune va apela interruptul 0x21, un fel de funcție a sistemului MS-DOS care afișează un șir de caractere pe ecran.¹⁷ Funcția este identificată prin registrul “ah”, iar parametrul, șirul de caractere care urmează să fie afișat, va fi un șir de caractere care se va termina cu caracterul “\$”, e indicat de registrul “dx”.

Dacă ne uităm în hex editor putem observa cu ușurință faptul că șirul de caractere se află la poziția 0xE (față de începutul codului executabil) și că se termină cu caracterul “\$”.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZÿÿ..
00000010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	f0	00	00	008...
0000004e	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	..°..'.í!..Lí!T
00000050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program cannc
00000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
00000070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....\$.....

¹⁷ DOS Interrupts - <http://spike.scu.edu.au/~barry/interrupts.html#ah09>

2.4. Antetul PE

După cum spuneam la prezentarea header-ului MS-DOS, ultimul câmp din această structură, ultimii 4 octeți din cei 64, câmpul `e_lfanew`, reprezintă un număr pe 4 octeți care indică locația “noului” header, indică exact poziția în fișier de unde începe header-ul PE.

Câmpul este un număr LONG, formatul fiind little-endian, astfel octeții sunt în ordine inversă. Desigur, citirea acestor 4 octeți într-o variabilă “long” se va face corect, iar variabila va conține valoarea corectă deoarece formatul din fișier este formatul în care e memorată acea valoare în memorie, pe procesoarele Intel - little-endian.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000000	4d	5a	90	00	03	00	00	00	04	00	00	00	ff	ff	00	00	MZÿÿ..
00000010	b8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000003c	00	00	00	00	00	00	00	00	00	00	00	00	f0	00	00	00\$...
00000040	0e	1f	ba	0e	00	b4	09	cd	21	b8	01	4c	cd	21	54	68	..°..'.í!..Lí!Th
00000050	69	73	20	70	72	6f	67	72	61	6d	20	63	61	6e	6e	6f	is program canno
00000060	74	20	62	65	20	72	75	6e	20	69	6e	20	44	4f	53	20	t be run in DOS
00000070	6d	6f	64	65	2e	0d	0d	0a	24	00	00	00	00	00	00	00	mode....\$.....
00000080	63	8a	9f	9f	27	eb	f1	cc	27	eb	f1	cc	27	eb	f1	cc	cšÿÿ'ëñì'ëñì'ëñì
00000090	2e	93	62	cc	16	eb	f1	cc	27	eb	f0	cc	55	e8	f1	cc	..`bì.ëñì'esìUëñì
000000a0	2e	93	63	cc	26	eb	f1	cc	2e	93	64	cc	20	eb	f1	cc	..`cìëñì.~dì ëñì
000000b0	2e	93	72	cc	d1	eb	f1	cc	2e	93	75	cc	c4	eb	f1	cc	..`rìNëñì.~uìÄëñì
000000c0	2e	93	65	cc	26	eb	f1	cc	2e	93	60	cc	26	eb	f1	cc	..`eìëñì.~`ìëñì
000000d0	52	69	63	68	27	eb	f1	cc	00	00	00	00	00	00	00	00	Rich'ëñì.....
000000e0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000f0	50	45	00	00	4c	01	04	00	15	3b	b8	50	00	00	00	00	PE..I....;P....
00000100	00	00	00	00	e0	00	02	21	0b	01	09	00	00	50	0c	00à..!.....P..

După cum se poate vedea în exemplul de mai sus, header-ul PE se află la locația 0xf0 (240 zecimal) în executabil.

Imediat la aceasta locație putem găsi noul header PE, care poate fi găsit în “WinNT.h” cu numele `IMAGE_NT_HEADERS`.

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Un lucru pe care ar trebui să îl avem în considerare îl reprezintă faptul că aceste structuri sunt diferite pe versiunile de 32 de biți și pe 64 de biți:

```
#ifdef _WIN64
typedef IMAGE_NT_HEADERS64          IMAGE_NT_HEADERS;
typedef PIMAGE_NT_HEADERS64         PIMAGE_NT_HEADERS;
#else
typedef IMAGE_NT_HEADERS32          IMAGE_NT_HEADERS;
typedef PIMAGE_NT_HEADERS32         PIMAGE_NT_HEADERS;
#endif
```

Structura este aparent simplă, conține doar 3 câmpuri:

- **Signature**: un număr de 4 octeți care identifică această structură ca fiind o structură PE, definiția sa poate fi găsită ca “#define IMAGE_NT_SIGNATURE 0x00004550 // PE00”. Se poate identifica ușor în executabil, caracterele text “PE” identifică începutul acestui header în fișier
- **FileHeader**, o structură de tipul IMAGE_FILE_HEADER, simplă, care conține informații referitoare la proprietățile fișierului
- **OptionalHeader**, o structură complexă, de tipul IMAGE_OPTIONAL_HEADER care conține mai multe informații. Deși se găsește în orice fișier executabil sau în orice bibliotecă de funcții, această structură poate fi opțională pentru fișierele obiect, cum ar fi rezultatul compilării unui singur fișier

Câmpul “**FileHeader**”, structura IMAGE_FILE_HEADER este definită astfel:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Informațiile conținute de această structură (20 de octeți) sunt următoarele:

- **Machine:** Un număr care identifică arhitectura procesorului, de exemplu i386, IA64...
Exemple:

#define IMAGE_FILE_MACHINE_I386	0x014c	// Intel 386.
#define IMAGE_FILE_MACHINE_ARM	0x01c0	// ARM Little-Endian
#define IMAGE_FILE_MACHINE_POWERPC	0x01f0	// IBM PowerPC
#define IMAGE_FILE_MACHINE_IA64	0x0200	// Intel 64

În cazul exemplului nostru, imediat după primii 4 octeți care reprezintă semnătura PE (50 45 00 00), urmează 2 octeți care reprezintă valoarea acestui câmp: 4c 01 care este numărul 0x014c, ceea ce înseamnă că executabilul nostru este destinat arhitecturii procesoarelor Intel 386 sau mai noi.

- **NumberOfSections:** Reprezintă numărul de secțiuni din fișier. Vom observa că fișierele conțin date împărțite în secțiuni. În fișierul analizat de noi, imediat după cei 2 octeți care reprezintă câmpul “Machine”, observăm că fișierul nostru conține 4 secțiuni.

- **TimeStamp:** Un număr pe 4 octeți (time_t din C) care reprezintă data la care fișierul a fost creat, măsurat în numărul de secunde trecute de la 1 Ian. 1970.

Putem observa următoarea valoare pentru acest câmp: 15 3b b8 50 adică numărul 0x50b83b15, în zecimal 1354251029, ceea ce înseamnă că fișierul nostru a fost creat la data de “Fri, 30 Nov 2012 04:50:29 GMT”.

- **PointerToSymbolTable:** Acest câmp ar trebui să fie un pointer către un tabel de simboluri COFF însa nu mai este folosit și ar trebui să aibă valoarea 0.

- **NumberOfSymbols:** Numărul de simboluri din tabelul de mai sus care nu este folosit, așadar ar trebui să aibă valoarea 0.

- **SizeOfOptionalHeader:** Conține mărimea header-ului opțional, cel care urmează imediat după această structură

- **Characteristics:** Un set de flag-uri care indică informații despre fișier: dacă este executabil sau bibliotecă de funcții, sau altele.

Exemple:

#define IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	// File is executable
(i.e. no unresolved external references).		
#define IMAGE_FILE_LARGE_ADDRESS_AWARE	0x0020	// App can handle >2gb
addresses		
#define IMAGE_FILE_32BIT_MACHINE	0x0100	// 32 bit word machine.
#define IMAGE_FILE_SYSTEM	0x1000	// System File.
#define IMAGE_FILE_DLL	0x2000	// File is a DLL.

Câmpul **OptionalHeader** (224 de octeți), structura `IMAGE_FILE_HEADER`, este definită astfel:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD    SizeOfCode;
    DWORD    SizeOfInitializedData;
    DWORD    SizeOfUninitializedData;
    DWORD    AddressOfEntryPoint;
    DWORD    BaseOfCode;
    DWORD    BaseOfData;

    //
    // NT additional fields.
    //

    DWORD    ImageBase;
    DWORD    SectionAlignment;
    DWORD    FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD    Win32VersionValue;
    DWORD    SizeOfImage;
    DWORD    SizeOfHeaders;
    DWORD    CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD    SizeOfStackReserve;
    DWORD    SizeOfStackCommit;
    DWORD    SizeOfHeapReserve;
    DWORD    SizeOfHeapCommit;
    DWORD    LoaderFlags;
    DWORD    NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Într-un fișier PE, un exemplu de astfel de structură, cei 224 de octeți din care este alcatuită:

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
00000080	63	8a	9f	9f	27	eb	f1	cc	27	eb	f1	cc	27	eb	f1	cc	cšŸŸ'ēñì'ēñì'ēñì
00000090	2e	93	62	cc	16	eb	f1	cc	27	eb	f0	cc	55	e8	f1	cc	."bì.ēñì'ēsìŸēñì
000000a0	2e	93	63	cc	26	eb	f1	cc	2e	93	64	cc	20	eb	f1	cc	."cìēñì."dì ēñì
000000b0	2e	93	72	cc	d1	eb	f1	cc	2e	93	75	cc	c4	eb	f1	cc	."rìŸēñì."uìĀēñì
000000c0	2e	93	65	cc	26	eb	f1	cc	2e	93	60	cc	26	eb	f1	cc	."ēìēñì."`ìēñì
000000d0	52	69	63	68	27	eb	f1	cc	00	00	00	00	00	00	00	00	Rich'ēñì.....
000000e0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000f0	50	45	00	00	4c	01	04	00	15	3b	b8	50	00	00	00	00	PE..L....;P....
00000100	00	00	00	00	e0	00	02	21	0b	01	09	00	00	50	0c	00à...!.....P..
00000110	00	e0	00	00	00	00	00	00	6f	cd	04	00	00	10	00	00	.à.....oÍ.....
00000120	00	00	0c	00	00	00	de	77	00	10	00	00	00	10	00	00pw.....
00000130	06	00	01	00	06	00	01	00	06	00	01	00	00	00	00	00
00000140	00	40	0d	00	00	10	00	00	c7	ff	0d	00	03	00	40	01	.@.....ÇŸ....@.
00000150	00	00	04	00	00	10	00	00	00	00	10	00	00	10	00	00
00000160	00	00	00	00	10	00	00	00	c0	51	0b	00	b1	a9	00	00ÀQ..±@..
00000170	74	fb	0b	00	f4	01	00	00	00	70	0c	00	28	05	00	00	tû..ô....p..(...
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000190	00	80	0c	00	b0	b0	00	00	b4	59	0c	00	38	00	00	00	.€.."".'Y..8...
000001a0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001b0	00	00	00	00	00	00	00	00	90	28	08	00	40	00	00	00 (..@..
000001c0	00	00	00	00	00	00	00	00	00	10	00	00	fc	0d	00	00ü...
000001d0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001e0	00	00	00	00	00	00	00	00	2e	74	65	78	74	00	00	00text...
000001f0	15	4a	0c	00	00	10	00	00	00	50	0c	00	00	10	00	00	.J.....P.....
00000200	00	00	00	00	00	00	00	00	00	00	00	00	20	00	00	60`

Structura **IMAGE_FILE_HEADER** este alcatuită din următoarele câmpuri:

- **Magic:** Un număr pe 2 octeți care identifică practic tipul de executabil, de cele mai multe ori PE32 (valoarea 0x10b) dar poate fi și PE32+ (valoarea 0x20b), executabile ce permit adresarea pe 64 de biți în timp ce limitează dimensiunea executabilului la 2 GB
- **MajorLinkerVersion, MinorLinkerVersion:** După cum spune numele, aceste câmpuri de câte un octet identifică versiunea link-erului folosită. Cum această structură, IMAGE_FILE_HEADER nu se regăsește în fișierele obiect, înseamnă că structura e creată la linkare, de către linker

Vom observa că datele dintr-un fișier PE sunt împărțite în secțiuni. Astfel avem secțiuni pentru cod (cel mai adesea o secțiune cu numele “.text”), secțiuni pentru date inițializate sau date neinițializate (cel mai adesea numele “.bss”) care conțin variabilele globale dar și variabilele statice, secțiuni pentru date “read-only” ca “.rdata” ce conțin șiruri de caractere sau constante.

- **SizeOfCode:** Indică dimensiunea secțiunii de cod (“.text”) sau dimensiunea tuturor secțiunilor de cod în cazul în care sunt mai multe
- **SizeOfInitializedData:** Indică dimensiunea secțiunii de date inițializate sau dimensiunea tuturor secțiunilor de date inițializate în cazul în care sunt mai multe
- **SizeOfUninitializedData:** Indică dimensiunea secțiunii de date neinițializate sau dimensiunea tuturor secțiunilor de date neinițializate în cazul în care sunt mai multe
- **AddressOfEntryPoint:** Un câmp foarte important, o valoare RVA (Relative Value Address) adică un pointer către o locație relativă la adresa la care este încărcat modulul (.exe, .dll) în memorie (vezi mai jos câmpul ImageBase). E adresa de unde se începe execuția codului pentru un executabil, iar pentru o bibliotecă de funcții este locația funcției de inițializare (DllMain) sau “0” dacă nu există o astfel de funcție
- **BaseOfCode:** La fel, o adresă relativă la ImageBase, locația la care începe secțiunea de cod a fișierului, încărcat în memorie
- **BaseOfData:** Adresa relativă unde datele sunt încărcate în memorie
- **ImageBase:** Un câmp important, adresa (în spațiul de adrese al procesului respectiv) unde fișierul ar “prefera” să fie încărcat în memorie. Trebuie să fie un multiplu de 64KB și adresa implicită este 0x00400000
- **SectionAlignment:** Un lucru important despre fișierele PE este faptul că diferă modul în care arată pe disc și modul în care sunt încărcate în memorie, iar alinierea secțiunilor este această cauza. După cum am spus anterior, un fișier PE este format din secțiuni. Aceste secțiuni, în memorie, trebuie aliniate la dimensiunea specificată de acest câmp, de cele mai multe ori 0x1000 (4096 de octeți – dimensiunea unei pagini de memorie). Astfel, chiar dacă o secțiune ar trebui să înceapă de la adresa 0x2001, acesta nu este un multiplu de 0x1000 și secțiunea va fi încărcată în memorie la adresa 0x3000.

- **FileAlignment:** De asemenea, ca și câmpul anterior, secțiunile trebuie să fie aliniate și în fișierul de pe disc, la un multiplu de valoarea indicată de acest câmp. Implicit are valoarea 512 octeți, dar poate fi orice putere a lui 2 între 512 și 64K.
- **MajorOperatingSystemVersion, MinorOperatingSystemVersion:** Versiunea sistemului de operare destinat execuției fișierului. De exemplu 5.1 e Windows XP, 6.0 e Windows Vista și 6.1 e Windows 7.
- **MajorImageVersion, MinorImageVersion:** Versiunea fișierului
- **MajorSubsystemVersion, MinorSubsystemVersion:** Versiunea subsystemului (Windows Console, Windows...) – Vezi câmpul “Subsystem”
- **Win32VersionValue:** Nu e folosit, ar trebui să fie 0
- **SizeOfImage:** Dimensiunea imaginii încărcate în memorie, inclusiv headerele. Trebuie să fie un multiplu de “SectionAlignment”
- **SizeOfHeaders:** Dimensiunea tuturor headerelor: header DOS, header PE, headerle secțiunilor, rotunjit la un multiplu al câmpului “FileAlignment”
- **Checksum:** O sumă pe 4 octeți, verificată la încărcare pentru DLL-urile care sunt încărcate în timpul procesului de boot, DLL-urile încărcate într-un proces critic și pentru toate drivererele. Algoritmul este încorporat în ImagHelp.dll
- **Subsystem:** Subsistemul fișierului. De exemplu, poate fi:

```
#define IMAGE_SUBSYSTEM_NATIVE          1    // Image doesn't require a
subsystem.
#define IMAGE_SUBSYSTEM_WINDOWS_GUI     2    // Image runs in the Windows
GUI subsystem.
#define IMAGE_SUBSYSTEM_WINDOWS_CUI     3    // Image runs in the Windows
character subsystem.
```

Mai exact, are valoarea 1 dacă este un driver, valoarea 2 dacă este un executabil grafic (folosește ferestre) sau 3 dacă este o aplicație în linie de comandă.

- **DllCharacteristics:** Diverse trăsături cum ar fi: poate fi mutat la o adresă diferită, nu folosește SEH (Structured Exception Handler) sau e necesară forțarea verificării integrității
- **SizeOfStackReserve:** Ce dimensiune să fie rezervată, păstrată pentru stiva folosită de executabil. Nu este alocată, ci este doar specificată ca dimensiune maximă
- **SizeOfStackCommit:** Cât spațiu să fie alocat pe stivă inițial și cât spațiu să fie alocat când mai e necesară o alocare, până când se atinge dimensiunea specificată de câmpul anterior
- **SizeOfHeapReserve:** Cât spațiu să fie rezervat pentru heap
- **SizeOfHeapCommit:** Cât spațiu să fie alocat inițial și cât să fie alocat atunci când este necesar, până se atinge dimensiunea câmpului anterior
- **LoaderFlags:** Câmp rezervat, nefolosit, ar trebui să fie 0
- **NumberOfRvaAndSizes:** Numărul de intrări în tabelul următor, DataDirectory, cel mai adesea 16 (0x10). Înainte de a parcurge acest tabel trebuie verificată această valoare
- **DataDirectory:** Un vector [NumberOfRvaAndSizes] de structuri de tipul "IMAGE_DATA_DIRECTORY".

Câmpul DataDirectory conține de obicei 16 structuri de tipul IMAGE_DATA_DIRECTORY, structură definită astfel:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16
```

Câmpul "VirtualAddress" reprezintă adresa RVA (relativă la ImageBase) unde se află datele respective iar câmpul "Size" reprezintă dimensiunea datelor.

Datele conținute de acest vector, de DataDirectory, sunt predefinite. Astfel, indecșii elementelor de tipul IMAGE_DATA_DIRECTORY din vector sunt următorii:

```
// Directory Entries

#define IMAGE_DIRECTORY_ENTRY_EXPORT          0    // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1    // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2    // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3    // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4    // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5    // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6    // Debug Directory
//      IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7    // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7    // Architecture Specific
Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8    // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9    // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10    // Load Configuration
Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT    11    // Bound Import Directory in
headers
#define IMAGE_DIRECTORY_ENTRY_IAT             12    // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT    13    // Delay Load Import
Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR  14    // COM Runtime descriptor
```

Mai exact, primul element din vector (indicele 0) va conține tabelul de funcții exportate, următorul va conține funcțiile importate, următorul resursele conținute de fișier etc.

Elementele din vector nu sunt obligatorii, dacă un element nu este prezent, atunci va avea valoarea zero.

Member	Offset	Size	Value	Section
Export Directory RVA	00000168	Dword	000B51C0	.text
Export Directory Size	0000016C	Dword	0000A9B1	
Import Directory RVA	00000170	Dword	000BF874	.text
Import Directory Size	00000174	Dword	000001F4	
Resource Directory RVA	00000178	Dword	000C7000	.rsrc
Resource Directory Size	0000017C	Dword	00000528	
Exception Directory RVA	00000180	Dword	00000000	
Exception Directory Size	00000184	Dword	00000000	
Security Directory RVA	00000188	Dword	00000000	
Security Directory Size	0000018C	Dword	00000000	
Relocation Directory RVA	00000190	Dword	000C8000	.reloc
Relocation Directory Size	00000194	Dword	0000B0B0	
Debug Directory RVA	00000198	Dword	000C59B4	.text
Debug Directory Size	0000019C	Dword	00000038	
Architecture Directory RVA	000001A0	Dword	00000000	
Architecture Directory Size	000001A4	Dword	00000000	
Reserved	000001A8	Dword	00000000	
Reserved	000001AC	Dword	00000000	
TLS Directory RVA	000001B0	Dword	00000000	
TLS Directory Size	000001B4	Dword	00000000	
Configuration Directory RVA	000001B8	Dword	00082890	.text
Configuration Directory Size	000001BC	Dword	00000040	

Putem verifica ce conțin fiecare dintre aceste câmpuri cu un program simplu, care afișează datele din FileHeader, din OptionalHeader dar și vectorul DataDirectory.

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

int main()
{
    FILE *FD = NULL;
    IMAGE_DOS_HEADER dos;
    IMAGE_NT_HEADERS ntHeaders;
    size_t Len = 0;

    // Open file

    FD = fopen("C:\\Windows\\system32\\kernel32.dll", "rb");

    if(!FD)
    {
        puts("Cannot open file!");
        return 1;
    }

    // Read file

    Len = fread(&dos, 1, sizeof(dos), FD);

    if(Len != sizeof(dos))
    {
        puts("Cannot read file!");
        return 1;
    }

    // Seek to NT headers position

    if(fseek(FD, dos.e_lfanew, SEEK_SET) != 0)
    {
        printf("Cannot jump to NT headers: %lu", dos.e_lfanew);
        return 1;
    }

    Len = fread(&ntHeaders, 1, sizeof(ntHeaders), FD);

    if(Len != sizeof(ntHeaders))
    {
        puts("Cannot read NT headers!");
        return 1;
    }

    fclose(FD);

    // Print size of structures

    printf("Sizeof IMAGE_NT_HEADERS: %d\n", sizeof(IMAGE_NT_HEADERS));
    printf("Sizeof IMAGE_FILE_HEADER: %d\n", sizeof(IMAGE_FILE_HEADER));
```

```

        // Print FileHeader and Signature

        printf("Signature: 0x%.8lX\n", ntHeaders.Signature);
        printf("FileHeader.Machine: 0x%.4X\n", ntHeaders.FileHeader.Machine);
        printf("FileHeader.NumberOfSections: %d\n",
ntHeaders.FileHeader.NumberOfSections);
        printf("FileHeader.TimeDateStamp: 0x%.8lX\n",
ntHeaders.FileHeader.TimeDateStamp);
        printf("FileHeader.SizeOfOptionalHeader: %d\n",
ntHeaders.FileHeader.SizeOfOptionalHeader);
        printf("FileHeader.Characteristics: 0x%.4X\n\n",
ntHeaders.FileHeader.Characteristics);

        printf("OptionalHeader.Magic: 0x%.4X\n",
ntHeaders.OptionalHeader.Magic);
        printf("OptionalHeader.MajorLinkerVersion: %d\n",
ntHeaders.OptionalHeader.MajorLinkerVersion);
        printf("OptionalHeader.MinorLinkerVersion: %d\n",
ntHeaders.OptionalHeader.MinorLinkerVersion);
        printf("OptionalHeader.SizeOfUninitializedData);
ntHeaders.OptionalHeader.AddressOfEntryPoint: 0x%.8lX\n",
ntHeaders.OptionalHeader.AddressOfEntryPoint);
        printf("OptionalHeader.BaseOfCode: 0x%.8lX\n",
ntHeaders.OptionalHeader.BaseOfCode);
        printf("OptionalHeader.BaseOfData: 0x%.8lX\n",
ntHeaders.OptionalHeader.BaseOfData);
        printf("OptionalHeader.ImageBase: 0x%.8lX\n",
ntHeaders.OptionalHeader.ImageBase);
        printf("OptionalHeader.MajorSubsystemVersion: %d\n",
ntHeaders.OptionalHeader.MajorSubsystemVersion);
        printf("OptionalHeader.MinorSubsystemVersion: %d\n",
ntHeaders.OptionalHeader.MinorSubsystemVersion);
        printf("OptionalHeader.Win32VersionValue: %lu\n",
ntHeaders.OptionalHeader.Win32VersionValue);
        printf("OptionalHeader.SizeOfImage: %lu\n",
ntHeaders.OptionalHeader.SizeOfImage);
        printf("OptionalHeader.SizeOfHeaders: %lu\n",
ntHeaders.OptionalHeader.SizeOfHeaders);
        printf("OptionalHeader.Subsystem: 0x%.4X\n",
ntHeaders.OptionalHeader.Subsystem);
        printf("OptionalHeader.DllCharacteristics: 0x%.4X\n",
ntHeaders.OptionalHeader.DllCharacteristics);
        printf("OptionalHeader.LoaderFlags: 0x%.8lX\n",
ntHeaders.OptionalHeader.LoaderFlags);
        printf("OptionalHeader.NumberOfRvaAndSizes: %lu\n\n",
ntHeaders.OptionalHeader.NumberOfRvaAndSizes);

        // Print data directory

        for(int i = 0; i < ntHeaders.OptionalHeader.NumberOfRvaAndSizes; i++)
            printf("DataDirectory[%d].VirtualAddress: 0x%.8lX [%lu bytes]\n",
i,
                ntHeaders.OptionalHeader.DataDirectory[i].VirtualAddress,
                ntHeaders.OptionalHeader.DataDirectory[i].Size);

        return 0;
}

```

2.5. Tabelul de secțiuni

După aceste headere, urmează tabelul de secțiuni al fișierului PE.

În structura care definește headerele PE, structura `IMAGE_NT_HEADERS`, câmpul “NumberOfSections” din cadrul structurii “FileHeader” de tipul `IMAGE_FILE_HEADER`, conține numărul de secțiuni pe care le conține fișierul.

Acest tabel de secțiuni care urmează imediat după headerle PE conține un vector de structuri de tipul “**IMAGE_SECTION_HEADER**”, structură ce definește o secțiune. Acest vector are exact “NumberOfSections” secțiuni.

Structura “**IMAGE_SECTION_HEADER**” (40 de octeți) e definită astfel:

```
#define IMAGE_SIZEOF_SHORT_NAME 8

typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD    PhysicalAddress;
        DWORD    VirtualSize;
    } Misc;
    DWORD    VirtualAddress;
    DWORD    SizeOfRawData;
    DWORD    PointerToRawData;
    DWORD    PointerToRelocations;
    DWORD    PointerToLinenumbers;
    WORD     NumberOfRelocations;
    WORD     NumberOfLinenumbers;
    DWORD    Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;

#define IMAGE_SIZEOF_SECTION_HEADER 40
```

În această structură, găsim următoarele câmpuri:

- **Name:** Numele secțiunii, un șir de maxim 8 caractere. Atenție, șirul nu se termină obligatoriu cu NULL!
- **VirtualSize:** Dimensiunea secțiunii atunci când este încărcată în memorie
- **VirtualAddress:** Adresa RVA (relativă la ImageBase) la care începe secțiunea
- **SizeOfRawData:** Dimensiunea secțiunii în fișier, trebuie să fie un multiplu al câmpului “FileAlignment” din OptionalHeader
- **PointerToRawData:** Adresa RVA la care începe secțiunea în fișier
- **PointerToRelocations:** Ar trebui să fie 0 pentru fișiere executabile, sau o altă valoare în cazul în care la încărcarea în memorie trebuie să se facă “relocari”. Relocările pot fi necesare pentru DLL-uri. În cazul în care un DLL nu se poate încărca la adresa preferată, există posibilitatea ca anumite adrese să fie modificate pentru a corespunde acestor modificări, acest lucru fiind realizat printr-un tabel de adrese care trebuie relocate în astfel de situații
- **PointerToLinenumbers:** Informații de debug care nu se mai folosesc, ar trebui să aibă valoarea 0
- **NumberOfRelocations:** Dacă sunt necesare relocări, aici se va putea găsi numărul de adrese care trebuie “relocate”
- **NumberOfLinenumbers:** Informații de debug care nu se mai folosesc, ar trebui să aibă valoarea zero
- **Characteristics:** Câmpul este o mască de biți care specifică mai multe detalii despre secțiune: dacă aceasta conține cod, date inițializate sau date neinițializate, permisiunile pentru secțiune când aceasta este încărcată în memorie și multe altele

Câteva exemple de biți care pot fi setați pentru o secțiune:

```
#define IMAGE_SCN_CNT_CODE                0x00000020  // Section contains
code.
#define IMAGE_SCN_CNT_INITIALIZED_DATA    0x00000040  // Section contains
initialized data.
#define IMAGE_SCN_CNT_UNINITIALIZED_DATA  0x00000080  // Section contains
uninitialized data.
#define IMAGE_SCN_MEM_DISCARDABLE         0x02000000  // Section can be
discarded.
#define IMAGE_SCN_MEM_NOT_CACHED          0x04000000  // Section is not
cachable.
#define IMAGE_SCN_MEM_NOT_PAGED           0x08000000  // Section is not
pageable.
#define IMAGE_SCN_MEM_SHARED               0x10000000  // Section is
shareable.
#define IMAGE_SCN_MEM_EXECUTE             0x20000000  // Section is
executable.
#define IMAGE_SCN_MEM_READ                 0x40000000  // Section is
readable.
#define IMAGE_SCN_MEM_WRITE                0x80000000  // Section is
writeable.
```

Exemplu:

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	000C4A15	00001000	000C5000	00001000	00000000	00000000	0000	0000	60000020
.data	00000FF0	000C6000	00001000	000C6000	00000000	00000000	0000	0000	C0000040
.rsrc	00000528	000C7000	00001000	000C7000	00000000	00000000	0000	0000	40000040
.reloc	0000B0B0	000C8000	0000C000	000C8000	00000000	00000000	0000	0000	42000040

Pentru a citi tabelul de secțiuni trebuie doar să sărim peste headerele NT și citim “NumberOfSections” * sizeof(IMAGE_SECTION_HEADER) octeți.

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

int main()
{
    FILE *FD = NULL;
    IMAGE_DOS_HEADER dos;
    IMAGE_NT_HEADERS ntHeaders;
    IMAGE_SECTION_HEADER Section;
    size_t Len = 0;
    unsigned char *pRawSectionTable = NULL;

    // Open file

    FD = fopen("C:\\Windows\\system32\\kernel32.dll", "rb");

    if(!FD)
    {
        puts("Cannot open file!");
        return 1;
    }

    // Read file

    Len = fread(&dos, 1, sizeof(dos), FD);

    if(Len != sizeof(dos))
    {
        puts("Cannot read file!");
        return 1;
    }

    // Seek to NT headers position

    if(fseek(FD, dos.e_lfanew, SEEK_SET) != 0)
    {
        printf("Cannot jump to NT headers: %lu", dos.e_lfanew);
        return 1;
    }

    // Read NT headers

    Len = fread(&ntHeaders, 1, sizeof(ntHeaders), FD);

    // Check if read was succesful

    if(Len != sizeof(ntHeaders))
    {
        puts("Cannot read NT headers!");
        return 1;
    }
}
```

```

        // File pointer is positioned immediatly afer NT Headers
        // So we can read section table

        pRawSectionTable = (unsigned char
*)malloc(ntHeaders.FileHeader.NumberOfSections *
sizeof(IMAGE_SECTION_HEADER));

        if(pRawSectionTable == NULL)
        {
            puts("Cannot allocate memory!");
            return 1;
        }

        Len = fread(pRawSectionTable, 1, ntHeaders.FileHeader.NumberOfSections *
sizeof(IMAGE_SECTION_HEADER), FD);

        if(Len != ntHeaders.FileHeader.NumberOfSections *
sizeof(IMAGE_SECTION_HEADER))
        {
            puts("Cannot read sections table!");
            return 1;
        }

        // Get each section from our buffer

        for(size_t i = 0; i < ntHeaders.FileHeader.NumberOfSections *
sizeof(IMAGE_SECTION_HEADER); i += sizeof(IMAGE_SECTION_HEADER))
        {
            Section = *(IMAGE_SECTION_HEADER *) (pRawSectionTable + i);

            printf("Section #%d\n", i / sizeof(IMAGE_SECTION_HEADER));
            printf("Name: %s\n", Section.Name);
            printf("VirtualSize: %.8lX\n", Section.Misc.VirtualSize);
            printf("VirtualAddress: %.8lX\n", Section.VirtualAddress);
            printf("SizeOfRawData %lu\n", Section.SizeOfRawData);
            printf("PointerToRawData: %.8lX\n", Section.PointerToRawData);
            printf("PointerToLinenumbers: %.8lX\n",
Section.PointerToLinenumbers);
            printf("NumberOfRelocations: %d\n", Section.NumberOfRelocations);
            printf("NumberOfLinenumbers: %d\n", Section.NumberOfLinenumbers);
            printf("Characteristics: %.8lX\n\n", Section.Characteristics);
        }

        fclose(FD);

        return 0;
}

```

2.6. Funcțiile importate si funcțiile exportate

În sistemul de operare Microsoft Windows există două componente de bază:

1. Fișiere executabile (.exe) – Programele de sine stătătoare
2. Bibliotecile de funcții (.dll) – Biblioteci de funcții utilizate de către fișierele executabile

Astfel, se folosesc două concepte importante: importul și exportul funcțiilor. În mod standard, bibliotecile de funcții exportă funcțiile, iar programele executabile le importă pentru a le putea apela. Pentru a putea exporta și importa funcții, programele folosesc următoarele cuvinte cheie acceptate de către compilatoarele pentru Windows:

- `__declspec(dllexport)`¹⁸
- `__declspec(dllimport)`¹⁹

De exemplu, câteva funcții exportate de către biblioteca "**kernel32.dll**":

● Istrcat	0x77e2a739	0x0004a739	1347 (0x543)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcatA	0x77e2a739	0x0004a739	1348 (0x544)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcatW	0x77e46774	0x00066774	1349 (0x545)	kernel32.dll	C:\Windows\system32\kernel32.dll
● Istrcmp	0x77e19231	0x00039231	1350 (0x546)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcmpA	0x77e19231	0x00039231	1351 (0x547)	kernel32.dll	C:\Windows\system32\kernel32.dll
● Istrcmpi	0x77e22be5	0x00042be5	1353 (0x549)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcmpiA	0x77e22be5	0x00042be5	1354 (0x54a)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcmpiW	0x77e2abe7	0x0004abe7	1355 (0x54b)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcmpW	0x77e353d9	0x000553d9	1352 (0x548)	kernel32.dll	C:\Windows\system32\kernel32.dll
● Istrcpy	0x77e2a6c7	0x0004a6c7	1356 (0x54c)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcpyA	0x77e2a6c7	0x0004a6c7	1357 (0x54d)	kernel32.dll	C:\Windows\system32\kernel32.dll
● Istrcpyn	0x77e18fb1	0x00038fb1	1359 (0x54f)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcpynA	0x77e18fb1	0x00038fb1	1360 (0x550)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcpynW	0x77e479e0	0x000679e0	1361 (0x551)	kernel32.dll	C:\Windows\system32\kernel32.dll
● IstrcpyW	0x77e191d7	0x000391d7	1358 (0x54e)	kernel32.dll	C:\Windows\system32\kernel32.dll
● Istrlen	0x77e2a142	0x0004a142	1362 (0x552)	kernel32.dll	C:\Windows\system32\kernel32.dll

Aici încep să apară diferențe între tipurile diferite de fișiere PE: executabile și biblioteci. Fișierele executabile de cele mai multe ori vor avea o secțiune specială pentru importuri de funcții, secțiune care va specifica bibliotecile și numele funcțiilor importate din acele biblioteci, iar bibliotecile vor avea o secțiune specială care va specifica funcțiile exportate: numele și adresele fixe de memorie la care se vor găsi aceste funcții.

¹⁸ Exporting from a DLL - <http://msdn.microsoft.com/en-us/library/a90k134d.aspx>

¹⁹ Importing into an application - <http://msdn.microsoft.com/en-us/library/8fskxacy.aspx>

Pentru a citi lista de funcții exportate de către o bibliotecă de funcții, se folosește următoarea structură care identifică detaliile necesare pentru a putea face acest lucru:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Name;
    DWORD Base;
    DWORD NumberOfFunctions;
    DWORD NumberOfNames;
    DWORD AddressOfFunctions; // RVA from base of image
    DWORD AddressOfNames; // RVA from base of image
    DWORD AddressOfNameOrdinals; // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Structura se regăsește de cele mai multe ori în secțiunea **".edata"** și conține următoarele câmpuri:

- **Characteristics:** Rezervat pentru a fi folosit de către sistemul de operare, trebuie să aibă valoarea zero
- **TimeDateStamp:** Timpul și data la care a fost creat tabelul de funcții exportate
- **MajorVersion:** Versiunea majoră a tabelului de funcții, poate fi specificată de către utilizator
- **MinorVersion:** Versiunea minoră a tabelului de funcții, poate fi specificată de către utilizator
- **Name:** Adresă de memorie către un sir de caractere ASCII care identifică numele DLL-ului
- **Base:** Cum un executabil poate să aibă mai multe tabele de exporturi, acest câmp identifică și trebuie să fie diferit pentru fiecare dintre aceste tabele de exporturi
- **NumberOfFunctions:** Numărul de funcții exportate
- **NumberOfNames:** Numărul de nume de funcții exportate, deoarece funcțiile pot fi exportate și printr-un număr numit ordinal

- **AddressOfFunctions:** Un pointer, o adresă care indică adresa de început a unei liste care conține adresele de memorie la care se vor găsi funcțiile
- **AddressOfNames:** Adresa de memorie a unei liste care conține numele funcțiilor exportate
- **AddressOfNameOrdinals:** Adresa de memorie a unei liste care conține funcțiile exportate prin ordinal, nu prin nume

După cum se poate observa, cele mai importante câmpuri sunt "AddressOfFunctions" și "AddressOfNames", ambele fiind necesare pentru a putea găsi adresa unei funcții în funcție de numele acesteia, în lista de exporturi a unei biblioteci de funcții.

Codul de mai jos, folosit de către aplicația Malware Monitor, exemplifică cum poate fi citit un tabel de funcții exportate și cum se pot memora acestea într-o structură.

```
pcImageBase = vModules[i].modBaseAddr;

// Parse PE headers

IMAGE_DOS_HEADER oDOS;
IMAGE_NT_HEADERS oNT;
IMAGE_DATA_DIRECTORY oExportDirEntry;
IMAGE_EXPORT_DIRECTORY oExportDirectory;

// Parse EAT

DWORD *pdwAddressOfFunctions = NULL;
DWORD *pdwAddressOfNames = NULL;
CHAR *pcFunctionName = NULL;
DWORD dwFunctionAddress = 0;
DWORD dwFunctionPointerLocation = 0;

// Get Export directory

memcpy(&oDOS, pcImageBase, sizeof(oDOS));
memcpy(&oNT, (BYTE *) ((DWORD)pcImageBase + oDOS.e_lfanew), sizeof(oNT));
oExportDirEntry =
oNT.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];
memcpy(&oExportDirectory, (BYTE *) ((DWORD)pcImageBase +
oExportDirEntry.VirtualAddress), sizeof(oExportDirectory));

// Parse names

pdwAddressOfNames = (DWORD *) ((DWORD)pcImageBase +
oExportDirectory.AddressOfNames);
pdwAddressOfFunctions = (DWORD *) ((DWORD)pcImageBase +
oExportDirectory.AddressOfFunctions);

for(DWORD nr = 0; nr < oExportDirectory.NumberOfFunctions; nr++)
{
    EXPORT_ENTRY oExport;

    // Get function details

    pcFunctionName = (CHAR *) ((DWORD)pcImageBase +
(DWORD) (pdwAddressOfNames[nr]));
    dwFunctionAddress = (DWORD)pcImageBase +
(DWORD) (pdwAddressOfFunctions[nr]);
    dwFunctionPointerLocation = (DWORD)pcImageBase +
oExportDirectory.AddressOfFunctions + nr * sizeof(DWORD);

    // Save new function export

    oExport.dwAddress = dwFunctionAddress;
    oExport.dwPointerOfAddress = dwFunctionPointerLocation;
    oExport.sName = pcFunctionName;
    oExport.uOrdinal = (USHORT)nr + 1;

    vExports.push_back(oExport);
}
```

După cum se poate observa, pentru a citi tabelul de exporturi se folosesc doi pointeri: un pointer către numele funcției și un pointer către adresa funcției, deoarece ele sunt memorate în două liste diferite.

De exemplu, în fișierul executabil, numele funcțiilor exportate se regăsesc astfel, într-o listă, separate de caracterul NULL:

000bf5e6	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	
000bf500	73	74	72	63	61	74	41	00	6c	73	74	72	63	61	74	57	strcatA.lstrcatW
000bf510	00	6c	73	74	72	63	6d	70	00	6c	73	74	72	63	6d	70	.lstrcmp.lstrcmp
000bf520	41	00	6c	73	74	72	63	6d	70	57	00	6c	73	74	72	63	A.lstrcmpW.lstrc
000bf530	6d	70	69	00	6c	73	74	72	63	6d	70	69	41	00	6c	73	mpi.lstrcmpiA.l
000bf540	74	72	63	6d	70	69	57	00	6c	73	74	72	63	70	79	00	trcmpiW.lstrcpy.
000bf550	6c	73	74	72	63	70	79	41	00	6c	73	74	72	63	70	79	lstrcpyA.lstrcpy
000bf560	57	00	6c	73	74	72	63	70	79	6e	00	6c	73	74	72	63	W.lstrcpyn.lstrc
000bf570	70	79	6e	41	00	6c	73	74	72	63	70	79	6e	57	00	6c	pynA.lstrcpynW.l
000bf580	73	74	72	6c	65	6e	00	6c	73	74	72	6c	65	6e	41	00	strlen.lstrlenA.

Lucrurile sunt asemănătoare și pentru procesarea funcțiilor importate de către un fișier executabil. Pentru fiecare bibliotecă pe care o folosește fișierul executabil, acesta va conține o structură de forma următoare:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics;           // 0 for terminating null import
    descriptor
        DWORD OriginalFirstThunk;       // RVA to original unbound IAT
    (PIMAGE_THUNK_DATA)
    } DUMMYUNIONNAME;
    DWORD TimeDateStamp;                // 0 if not bound,
    DWORD ForwarderChain;               // -1 if no forwarders
    DWORD Name;
    DWORD FirstThunk;                  // RVA to IAT (if bound this IAT
has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```


Câmpurile din această structură sunt următoarele:

- **OriginalFirstThunk:** Deși este un "union", câmpul nu mai prezintă o valoare care să definească anumite caracteristici, acel câmp "Characteristics" este încă prezent din motive de compatibilitate cu versiunile mai vechi de sisteme de operare. Cât timp executabilul nu este încărcat în memorie de către Windows Loader, funcționalitatea din Windows care se ocupă de execuția programelor și încărcarea bibliotecilor în memorie, acesta e un pointer către un tabel identic cu cel indicat de câmpul "FirstThunk", tabel numit: "Import lookup table".
- **TimeDateStamp:** Valoare folosită de către sistemul de operare pentru încărcarea DLL-urilor la execuție
- **ForwarderChain:** Folosit de către sistemul de operare
- **Name:** Adresă către un șir de caractere ASCII care conține numele bibliotecii din care importă funcții, de exemplu "kernel32.dll"
- **FirstThunk:** Un pointer către un tabel de importuri numit "Import lookup tabel"

Câmpurile utile pentru procesarea funcțiilor importate de către un fișier executabil sau de către o bibliotecă de funcții deoarece și o bibliotecă de funcții poate importa funcții din alte biblioteci, sunt FirstThunk și OriginalFirstThunk.

Ambele câmpuri reprezintă niște pointeri către "Import lookup tabel", un tabel care conține structuri IMAGE_THUNK_DATA astfel:

- pe sistemele pe 32 de biți, structura are 32 de biți
- pe sistemele pe 64 de biți, structura are 64 de biți

Structura pe sistemele de 32 de biți este următoarea:

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString;    // PBYTE
        DWORD Function;          // PDWORD
        DWORD Ordinal;
        DWORD AddressOfData;      // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;
typedef IMAGE_THUNK_DATA32 * PIMAGE_THUNK_DATA32;
```

Structura conține un singur câmp, un pointer către o structura numită `IMAGE_IMPORT_BY_NAME` care conține câmpurile:

```
typedef struct _IMAGE_IMPORT_BY_NAME {  
    WORD    Hint;  
    BYTE    Name[1];  
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Structura are patru octeți și face abstracție de câmpurile pe care le conține. Datele pe care le conține sunt următoarele:

- cel mai semnificativ bit este un bit care indică dacă funcția respectivă este importată prin nume sau prin ordinal
- următorii 16 biți, în cazul în care primul bit are valoarea 1, trebuie să conțină ordinalul funcției importate
- următorii 31 de biți conțin un pointer către adresa la care se găsește numele funcției importate prin nume

Astfel, navigând printre aceste structuri, se poate citi lista de funcții importate de către un fișier Portable Executable.

3. INJECTAREA UNUI DLL

3.1. Introducere

Injectarea unui DLL²⁰, a unei biblioteci de funcții, într-un alt proces, e o metodă de programare foarte folosită pe sistemele Microsoft Windows deoarece permite modificarea sau chiar extinderea funcționalității unor procese chiar dacă nu este disponibil codul sursă pentru aceste.

Metoda este utilă deoarece oferă posibilitatea de a executa cod într-un alt proces, cod scris într-un limbaj ca C/C++ și cod care poate să interacționeze după bunul plac cu procesul în care este injectat DLL²¹-ul respectiv.

Câteva exemple de programe care folosesc această metodă:

- Kaspersky²²: Pentru a intercepta și monitoriza apelurile de funcții și modulele încărcate în memorie de către procese
- EMET: Pentru a proteja posibile programe vulnerabile la atacurile de tipul buffer overflow sau alte atacuri
- Keyscrambler²³: Pentru a proteja utilizatorii împotriva programelor de tipul “keylogger”, programe care capturează tastele apăsate
- SpyEye/Zeus²⁴: Pentru a putea captura datele bancare ale utilizatorilor

Metoda este des folosită împreună cu interceptarea apelurilor de funcții.

²⁰ DLL Injection - http://en.wikipedia.org/wiki/DLL_injection

²¹ Dynamic link library - http://en.wikipedia.org/wiki/Dynamic-link_library

²² Kaspersky - <http://www.kaspersky.com/>

²³ QFX Software, Keyscrambler - <https://www.qfxsoftware.com/>

²⁴ Zeus (trojan horse) - http://en.wikipedia.org/wiki/Zeus_%28Trojan_horse%29

3.2. Cum funcționează?

După cum am stabilit, pentru a putea executa cod în spațiul de adrese al altui proces, cea mai simplă și eficientă metodă de a realiza acest lucru este de a injecta un DLL în procesul respectiv. Astfel, codul scris de către noi în respectivul DLL, deoarece vom injecta un DLL propriu, având codul nostru, va fi executat în spațiul de adrese al unui alt proces și vom avea posibilitatea de a citi și modifica date din procesul respectiv.

Deși există mai multe metode de a injecta un DLL, cea mai sigură metodă este de a urma cațiva pași simpli pe care îi vom descrie ulterior:

- Deschiderea procesului folosind funcția **OpenProcess**
- Găsirea adresei funcției "**LoadLibrary**" folosind funcția **GetProcAddress**
- Alocarea de memorie în spațiul procesului respectiv folosind funcția **VirtualAllocEx**
- Scrierea în memoria alocată a numelui DLL-ului de injectat folosind funcția **WriteProcessMemory**
- Crearea unui nou thread în procesul respectiv folosind funcția **CreateRemoteThread**
- Se închide procesul deschis folosind funcția **CloseHandle**

Pe scurt, modul de funcționare este următorul:

1. Se alocă spațiu în procesul respectiv
2. Se memorează în spațiul alocat șirul "InjectedDLL.dll"
3. Se crează un nou thread în proces. La crearea unui thread e necesară specificarea unei funcții care să fie apelată și executată în nou thread. În cazul de față, în noul thread vom apela funcția "LoadLibrary", funcție care permite încărcarea unui DLL în memorie, specificând ca parametru DLL-ul creat de noi.

Astfel, vom apela din spațiul de adrese al procesului pe care dorim să îl monitorizăm LoadLibrary("InjectedDLL.dll") și DLL-ul nostru va fi încărcat automat și executat.

3.3. Funcțiile API folosite

Toate funcțiile necesare fac parte din biblioteca WinAPI²⁵ kernel32.dll. Pentru o bună înțelegere a acestui proces, vom vedea ce face mai exact fiecare dintre aceste funcții, ce parametri are și ce returnează. MSDN (Microsoft Developer Network)²⁶, portalul pentru dezvoltători de la Microsoft, oferă documentația necesară pentru toate aceste funcții.

OpenProcess:²⁷

```
HANDLE WINAPI OpenProcess(  
    _In_   DWORD dwDesiredAccess,  
    _In_   BOOL bInheritHandle,  
    _In_   DWORD dwProcessId  
);
```

Funcția OpenProcess are următorii parametri:

- **dwDesiredAccess:** o valoare de tipul "bitmask", care specifică drepturile de acces pe care le dorim în procesul pe care îl deschidem: dacă putem crea un nou thread în respectivul proces, dacă putem scrie sau citi zonele sale de memorie etc.
- **bInheritHandle:** dacă e "true", procesele create de acel proces vor moșteni handle-ul returnat de această funcție
- **dwProcessId:** ID-ul procesului pe care dorim să îl deschidem. Deși procesele pot fi indicate prin nume, există posibilitatea de a avea mai multe procese cu același nume, însă fiecare proces are un astfel de identificator unic care se poate obține pentru orice proces

Funcția returnează un "handle" al procesului deschis, mai exact un descriptor al procesului, o valoare prin care putem accesa procesul respectiv așa cum accesăm fișierele deschise cu funcția "fopen" în C/C++.

²⁵ Windows API - http://en.wikipedia.org/wiki/Windows_API

²⁶ Microsoft Developer Network - <http://msdn.microsoft.com/en-us/default.aspx>

²⁷ OpenProcess - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms684320%28v=vs.85%29.aspx>

GetModuleHandle:²⁸

```
HMODULE WINAPI GetModuleHandle(  
    _In_opt_ LPCTSTR lpModuleName  
);
```

În terminologia Windows, în memorie, atât fișierele executabile cât și bibliotecile de funcții se numesc "module". Pentru a putea lucra cu modulele încărcate în memorie, trebuie să le "deschidem" la fel ca pe procese și să obținem un descriptor (handle) al lor. Pentru a obține un handle al unui modul, apelăm funcția GetModuleHandle.

Funcția GetModuleHandle are următorul parametru:

- **lpModuleName:** numele DLL-ului pentru care dorim să obținem un descriptor, sau calea completă a acestuia folosind backslash-uri "\". Dacă e specificată valoarea NULL, se va considera că modul este procesul curent (Malware monitor.exe) care este de asemenea un modul

Făra a intra mai mult în detalii, pentru protecția aplicațiilor împotriva atacurilor de tip "buffer overflow", Microsoft dar și alte sisteme de operare au introdus ASLR (Address Space Layout Randomization), un mecanism care permite încărcarea bibliotecilor de funcții (DLL) dar și a altor structuri folosite de către procese, la adrese de memorie aleatoare, deoarece pâna la Windows XP un DLL era încărcat întotdeauna la aceeași adresă.

Astfel, pentru o bună funcționare a aplicației, va trebui să obținem în mod dinamic adresele funcțiilor pe care dorim să le apelăm dinamic. Un lucru foarte important este că deși ASLR specifică încărcarea DLL-urilor la adrese aleatoare de memorie, DLL-urile vor fi încărcate la exact aceeași adresă în cazul tuturor proceselor. Astfel, dacă kernel32.dll este încărcat în memoria procesului curent la adresa 0x11223344, atunci kernel32.dll va fi încărcat în memoria tuturor proceselor la adresa 0x11223344.

Tot ce trebuie să facem noi este să obținem adresa funcției "LoadLibrary" din procesul curent pentru a o putea în cadrul procesului în care dorim să injectăm DLL-ul.

Funcția returnează un descriptor al procesului curent, descriptor pe care îl vom folosi ulterior pentru a obține adresa funcției "LoadLibrary" exportată de către biblioteca "kernel32.dll".

²⁸ GetModuleHandle - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms683199%28v=vs.85%29.aspx>

GetProcAddress:²⁹

```
FARPROC WINAPI GetProcAddress(  
    _In_   HMODULE hModule,  
    _In_   LPCSTR lpProcName  
);
```

Funcția `GetProcAddress` e folosită pentru a obține adresa unei funcții. Așa cum am discutat anterior, adresa unei funcții exportate de către o bibliotecă de funcții se poate obține și prin procesarea manuală a unui fișier sau proces, dar acest lucru îl face și această funcție și este mult mai simplu și mai rapid.

Funcția `GetProcAddress` are următorii parametri:

- **hModule**: un descriptor al unei biblioteci de funcții în care dorim să găsim adresa unei funcții
- **lpProcName**: un șir de caractere care reprezintă numele funcției a cărei adresă dorim să o descoperim pentru a o putea apela

Funcția întoarce un pointer la funcția a cărei adresă o dorim și pe care o putem apela ulterior.

²⁹ `GetProcAddress` - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms683212%28v=vs.85%29.aspx>

VirtualAllocEx:³⁰

```
LPVOID WINAPI VirtualAllocEx(  
    _In_      HANDLE hProcess,  
    _In_opt_  LPVOID lpAddress,  
    _In_      SIZE_T dwSize,  
    _In_      DWORD flAllocationType,  
    _In_      DWORD flProtect  
);
```

Funcția VirtualAllocEx e folosită pentru a alocă memorie într-un alt proces. Așa cum de exemplu într-un proces se alocă memorie apelând funcția "malloc", așa putem alocă memorie în spațiul de adrese al unui alt proces folosind această funcție.

Funcția VirtualAllocEx are următorii parametri:

- **hProcess**: un descriptor al procesului în care dorim să alocăm memorie, descriptor obținut apelând funcția OpenProcess
- **lpAddress**: un pointer care specifică o zonă anume de memorie unde să fie alocat spațiu. Dacă are valoarea NULL, funcția va găsi o zonă de memorie în care va putea alocă spațiu
- **dwSize**: dimensiunea memoriei pe care dorim să o alocăm în procesul respectiv
- **flAllocationtype**: niște flag-uri care specifică cum să fie alocată memoria, de obicei se folosesc flag-urile MEM_COMMIT și MEM_RESERVE
- **flProtect**: tipul de acces pentru zona respectivă de memorie: citit, scris, executat în zona de memorie alocată

Funcția returnează un pointer, în spațiul de adrese al procesului respectiv, către zona de memorie alocată.

³⁰ VirtualAllocEx - <http://msdn.microsoft.com/en-us/library/windows/desktop/aa366890%28v=vs.85%29.aspx>

WriteProcessMemory:³¹

```
BOOL WINAPI WriteProcessMemory(  
    _In_     HANDLE hProcess,  
    _In_     LPVOID lpBaseAddress,  
    _In_     LPCVOID lpBuffer,  
    _In_     SIZE_T nSize,  
    _Out_    SIZE_T *lpNumberOfBytesWritten  
);
```

Funcția WriteProcessMemory permite scrierea de date în spațiul de adrese al unui alt proces, modificarea memoriei unui alt proces.

Funcția WriteProcessMemory are următorii parametri:

- **hProcess**: un descriptor al procesului deschis anterior cu funcția OpenProcess
- **lpBaseAddress**: adresa de memorie din spațiul de adrese al procesului respectiv unde vrem să scriem date
- **lpBuffer**: un pointer către un buffer care conține datele pe care dorim să le scriem în procesul respectiv
- **nSize**: dimensiunea datelor conținute în buffer
- **lpNumberOfBytesWritten**: un pointer către o variabilă care după apelul funcției va conține numărul de octeți care au fost scriși cu succes în zona respectivă de memorie

Dacă funcția a reușit, va returna true.

³¹ WriteProcessMemory - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms681674%28v=vs.85%29.aspx>

LoadLibrary:³²

```
HMODULE WINAPI LoadLibrary(  
    _In_ LPCTSTR lpFileName  
);
```

Funcția încarcă în memorie o bibliotecă de funcții și execută funcția "Main", mai exact "DllMain" a bibliotecii respective. Ne vom folosi de funcția LoadLibrary, pe care o vom apela din cadrul procesului în care dorim să injectăm DLL-ul, pentru a încărca propriul DLL în spațiul de memorie al procesului respectiv.

Funcția LoadLibrary are următorul parametru:

- **lpFileName:** numele fișierului sau calea completă a DLL-ului pe care dorim să îl încercăm în memorie

Funcția va returna un descriptor al modulului proaspăt încărcat în memorie.

³² LoadLibrary - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms684175%28v=vs.85%29.aspx>

CreateRemoteThread:³³

```
HANDLE WINAPI CreateRemoteThread(  
    _In_     HANDLE hProcess,  
    _In_     LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    _In_     SIZE_T dwStackSize,  
    _In_     LPTHREAD_START_ROUTINE lpStartAddress,  
    _In_     LPVOID lpParameter,  
    _In_     DWORD dwCreationFlags,  
    _Out_    LPDWORD lpThreadId  
);
```

O funcție necesară pentru a duce la îndeplinire proiectul este CreateRemoteThread. Funcția, după cum indică și numele acesteia, ne permite să creăm un nou thread într-un alt proces!

Funcția CreateRemoteThread are următorii parametri:

- **hProcess**: descriptor al procesului deschis cu OpenProcess
- **lpThreadAttributes**: attribute pentru thread, NULL și se vor folosi attributele implicite
- **dwStackSize**: dimensiunea stack-ului, vom seta valoarea 0 pentru dimensiunea implicită
- **lpStartAddress**: cel mai important parametru, adresa funcției care va fi apelată când va fi creat noul thread
- **lpParameter**: foarte important, parametrul cu care va fi apelată funcția specificată la parametrul anterior
- **dwCreationFlags**: flag care specifică dacă threadul să fie executat imediat sau nu
- **lpThreadId**: după apelul funcției va conține identificatorul unic al thread-ului

Funcția va returna un descriptor pentru noul thread. După cum se poate observa, la apelul acestei funcții putem specifica atât adresa unei funcții cât și valoarea unui parametru. În apelul nostru vom specifica adresa funcției LoadLibrary iar ca valoare a parametrului vom seta pointer-ul la care am scris numele DLL-ului folosind funcția WriteProcessMemory.

³³ CreateRemoteThread - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms682437%28v=vs.85%29.aspx>

CloseHandle:³⁴

```
BOOL WINAPI CloseHandle(  
    _In_ HANDLE hObject  
);
```

După ce am deschis un proces trebuie să îl și închidem. Pentru acest lucru folosim funcția CloseHandle.

Funcția CloseHandle are următorul parametru:

- **hObject**: descriptorul procesului pe care dorim să îl închidem

Funcția va returna true dacă descriptorul va fi închis cu succes.

³⁴ CloseHandle - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms724211%28v=vs.85%29.aspx>

3.4. Rezumat

Putem înțelege astfel mai detaliat cum se injectează un DLL. Tot ceea ce vrem să facem este să apelăm din cadrul procesului în care vrem să injectăm un DLL apelul:

LoadLibrary("InjectedDLL.dll");

Pentru a realiza acest lucru trebuie să rezolvăm trei probleme:

1. Să obținem adresa funcției LoadLibrary
2. Să scriem șirul de caractere "InjectedDLL.dll" în procesul respectiv, deoarece specificat între ghilimele, șirul de caractere va face parte doar din spațiul de adrese al procesului curent, proces din care injectăm DLL-ul
3. Să apelăm funcția cu parametrul necesar

Vom parcurge astfel pașii discutați anterior și vom rezolva problemele:

1. GetProcAddress - obținem adresa funcției LoadLibrary
2. WriteProcessMemory - scriem șirul în procesul în care injectăm DLL-ul
3. CreateRemoteThread - apelăm funcția

Pe lângă această metodă există multe alte metode pentru a putea obține același rezultat:

1. Modificarea unei valori în Windows Registry, ceea ce va cauza ca orice executabil încărcat în memorie de către sistemul de operare să încarce și un anumit DLL
2. Folosirea funcției WinAPI SetWindowHookEx³⁵
3. Plasarea unui DLL cu aceleași exporturi ca și cel folosit de către program în același director cu programul. Astfel, în loc să încarce DLL-ul corect, programul va încarca DLL-ul plasat de utilizator în directorul său

Ideea principală a acestei metode de accesare a unui proces este că nu vom avea nicio limitare în codul pe care îl vom executa în cadrul altui proces, iar acest cod poate fi scris rapid și eficient într-un limbaj ca C/C++, fără a fi nevoie de Assembler sau limbaj mașină.

³⁵ SetWindowHookEx - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms644990%28v=vs.85%29.aspx>

Codul care realizează injectarea unui DLL:

```
// Function used to inject a DLL into a specific process

BOOL InjectDLL(string p_sDLLName, DWORD p_dwID)
{
    HANDLE hProcess, hRemoteThread;
    LPVOID pvString, pvLoadLibrary;
    BOOL bResult;

    // Open process

    hProcess = OpenProcess(PROCESS_ALL_ACCESS, false, p_dwID);

    // Get LoadLibrary address

    pvLoadLibrary = (LPVOID)GetProcAddress(GetModuleHandle("kernel32.dll"),
"LoadLibraryA");

    // Allocate space in remote process for DLL name

    pvString = (LPVOID)VirtualAllocEx(hProcess, NULL, p_sDLLName.length(),
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

    // Write DLL name in allocated space

    SIZE_T written = 0;

    bResult = WriteProcessMemory(hProcess, (LPVOID)pvString,
p_sDLLName.c_str(), p_sDLLName.length(), &written);

    // Create Remote thread to call "LoadLibrary(dll)"

    hRemoteThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)pvLoadLibrary, (LPVOID)pvString, 0, NULL);

    CloseHandle(hProcess);

    return true;
}
```

4. INTERCEPTAREA APELURILOR DE FUNCȚII

4.1. Introducere

Interceptarea apelurilor de funcții, cunoscută de asemenea sub numele de “API Hooking”³⁶, este un mecanism puternic de programare, folosit adesea de către aplicațiile care rulează pe sistemul de operare Microsoft Windows și care le permite acestora să aibă niște funcționalități spectaculoase și foarte utile pentru utilizatorii acestui sistem de operare.

Cea mai cunoscută categorie de astfel de aplicații îl reprezintă programele antivirus, programe care folosesc această metodă de programare atât pentru monitorizarea apelurilor de funcții din modul utilizator, cât și pentru interceptarea apelurilor de funcții de sistem, pentru un control detaliat al acțiunilor proceselor, cu scopul de a bloca sau anunța utilizatorul când un program dorește să execute o acțiune malițioasă.

Însă nu numai programele antivirus fac acest lucru. Există de asemenea foarte multe programe de tip “Banking trojan”, programe create cu rolul de a fura de la utilizatori datele bancare, care se folosesc de această metodă pentru a putea fura datele de card, date care cel mai adesea sunt trimise criptat către serverele procesatorilor de plăți. Un bun exemplu pentru această categorie îl reprezintă programele Zeus și SpyEye, programe care monitorizează apelurile de funcții din browsere, funcții care sunt folosite pentru a cripta sau trimite datele către servere, astfel având posibilitatea de a captura datele bancare.

Malware Monitor funcționează în mod asemănător, interceptează apelurile de funcții, însă nu cu același scop, ci cu scopul de a oferi utilizatorului informații detaliate despre modul de funcționare al unui fișier executabil.

³⁶ Hooking - <http://en.wikipedia.org/wiki/Hooking>

4.2. Cum funcționează?

Există mai multe metode de implementare a unei astfel de funcționalități, cele mai cunoscute dintre ele fiind următoarele:

- IAT (Import Address Table) patching
- EAT (Export Address Table) patching
- Inline hooking (code overwriting)

IAT (Import Address Table) este tabelul de funcții importate conținut de către fișierele executabile dar și de către bibliotecile de funcții care importă alte funcții. EAT (Export Address Table) reprezintă tabelul de funcții exportate de către bibliotecile de funcții.

Metodele de interceptare a apelurilor de funcții modifică tabelul de funcții importate sau tabelul de funcții exportate, în memorie, pentru procesele pentru care se dorește interceptarea funcțiilor, astfel încât adresele funcțiilor să nu mai indice către adresele funcțiilor originale, din bibliotecile de funcții care se doresc a fi monitorizate, ci adresa acelor funcții e suprascrisă cu o funcție cu un antet identic, cu același număr și tip de parametri dar și același tip de date returnat, sau cel puțin dimensiunea parametrilor și a valorii returnate să aibă aceeași valoare cu a funcției originale, funcție implementată într-un DLL implementat anterior. Astfel, apelul de funcție original va fi redirecționat către funcția din biblioteca de funcții injectată în proces, funcția va avea acces la parametrii cu care a fost apelată funcția originală și va putea apela funcția originală sau va putea bloca apelul returnând un cod de eroare.

Pe lângă avantajul de a putea fi rapid implementate, prin citirea și suprascrierea tabelului de funcții importate sau a tabelului de funcții exportate, aceste metode prezintă un dezavantaj major: nu oferă posibilitatea de a monitoriza apelurile dinamice ale funcțiilor.

Așa cum am discutat în capitolul anterior, există posibilitatea de a apela funcții în mod dinamic, la execuție. Această facilitate este oferită de către sistemul de operare Microsoft Windows prin intermediul bibliotecii de funcții “kernel32.dll” și a celor două funcții pe care le-am discutat anterior: LoadLibrary și GetProcAddress.

Din moment ce sunt apelate în mod dinamic, la execuție, fișierele executabile sau bibliotecile de funcții care apelează în mod dinamic funcții vor importa și vor avea în tabelul de funcții importate doar aceste funcții și niciuna dintre funcțiile pe care le vor apela. Astfel, cum tabelul de importuri nu va conține funcțiile respective, acestea nu vor putea fi monitorizate.

Pentru tabelul de funcții exportate apar de asemenea probleme: funcționează în mod corect în cazul în care o funcție este apelată prin intermediul acestui tabel, însă nu va funcționa în cazul în care a fost obținută deja adresa funcțiilor prin intermediul acestui tabel și adresele respective vor fi folosite, adresele către funcțiile originale. Spre exemplu, dacă la execuție, un program salvează într-o variabilă adresa unei funcții, degeaba e modificat ulterior tabelul de funcții exportate deoarece programul respectiv se va folosi de pointerul la funcție pentru a o apela.

Dacă în cazul tabelului de exporturi nu se poate face ulterior nimic, există posibilitatea de a extinde metoda IAT patching prin interceptarea și modificarea apelurilor de funcții ale funcțiilor LoadLibrary și GetProcAddress. Metoda extinsă poartă numele de “Extended IAT patching” și este mult mai eficientă dar mai greu de implementat. Astfel, la un apel al funcției LoadLibrary se pot pune “hook”-urile pe funcțiile care se doresc interceptate iar “hook”-ul pe funcția GetProcAddress poate returna direct un pointer către funcția din biblioteca de funcții care a fost injectată în proces.

Probabil cea mai eficientă metodă de API Hooking o reprezintă metoda „Inline patching”. Metoda este simplă la bază, însă implementarea acesteia poate să fie foarte dificilă. Ideea de bază este modificarea codului funcțiilor care se doresc a fi modificate, prin modificarea primelor instrucțiuni ale funcțiilor, instrucțiuni ce urmează a fi executate de către procesor, și prin plasarea unei instrucțiuni de salt, instrucțiunea „jmp” care permite saltul la o altă funcție.

Astfel, când un program va apela o funcție care este interceptată, programul va apela funcția originală, însă la realizarea apelului, procesorul va executa mai întâi instrucțiunea de salt la o altă adresă, de exemplu “jmp 0x11223344”, salt care va redirecționa procesorul către codul unei funcții din interiorul unei biblioteci de funcții injectate anterior în proces.

4.3. Apelurile de funcții

Dacă pentru un programator C/C++, un apel de funcție pare foarte simplu, pentru pasionații detaliilor lucrurile stau puțin altfel. Pentru un programator, un apel de funcție se rezumă la:

tip val_ret = **nume_funcție** (...parametri...) ;

Însă ceea ce se întâmplă în fundal este foarte complicat. Odată compilat, acest cod e transpus în mai multe instrucțiuni, instrucțiuni executate de către procesor pentru realizarea corectă a apelului de funcție.

În primul rând, există mai multe metode de a apela o funcție, metode care poartă denumirea de “calling conventions”³⁷. Cele mai cunoscute și folosite sunt:

- **cdecl**
- **stdcall**

Primul lucru care trebuie înțeles în legătură cu apelurile de funcții este existența unei stive. Fiecare proces are o stivă proprie, stivă folosită în cazul apelului de funcții pentru memorarea argumentelor. O altă caracteristică importantă a stivei este faptul că tot acolo sunt memorate și variabilele locale, variabile folosite în interiorul unei funcții când aceasta este apelată.

Fără a intra în detalii trebuie menționat faptul că procesorul folosește o serie de regiștrii: EAX, EBX, EBP, ESP³⁸ și înțelege anumite instrucțiuni: mov, jmp, call³⁹ și multe altele. Compilatorul este responsabil de transformarea codului C/C++ în astfel de instrucțiuni pe care procesorul le înțelege, le execută și programul funcționează așa cum se dorește.

Pentru a lucra cu stiva, procesorul folosește doi regiștrii: EBP – Extended stack Base Pointer, un registru, asemenea unei variabile C/C++, care memorează un pointer către zona inferioară a stivei, baza acesteia și ESP – Extended Stack Pointer, un registru care memorează adresa zonei superioare a stivei.

³⁷ X86 calling conventions - http://en.wikipedia.org/wiki/X86_calling_conventions

³⁸ Processor register - http://en.wikipedia.org/wiki/Processor_register

³⁹ X86 instruction listing - http://en.wikipedia.org/wiki/X86_instruction_listings

După cum se știe, o stivă “crește” atunci când sunt adăugate date. În cazul procesoarelor Intel x86, mecanismul este diferit, adică stiva “scade”, pleacă de la o adresă mai mare și când sunt adăugate date, valoarea ESP-ului, care memorează efectiv vârful stivei, scade. De exemplu, dacă ESP va conține valoarea 0x10000004, după adăugarea unor date pe stivă, valoarea acestui registru se va modifica automat la valoarea 0x10000000, scade cu 4 octeți, pe sistemele pe 32 de biți. Pentru adăugarea și eliminarea datelor de pe stivă se folosesc instrucțiunile: PUSH și POP.⁴⁰

Dar cum se realizează apelul unei funcții mai exact? Acest lucru depinde de acel “calling convention” pe care îl folosește funcția. Metoda “cdecl” este metoda implicită folosită pentru apelul funcțiilor dintr-un program scris în C/C++, iar “stdcall” este metoda folosită de către toate funcțiile sistemului de operare Windows, funcțiile WinAPI.

Spre exemplu, următorul cod C:⁴¹

```
int callee(int, int, int);  
  
int caller(void)  
{  
    int ret;  
  
    ret = callee(1, 2, 3);  
    ret += 5;  
    return ret;  
}
```

Va fi transformat de către compilator în:

```
caller:  
    push    ebp  
    mov     ebp, esp  
    push    3  
    push    2  
    push    1  
    call    callee  
    add     esp, 12  
    add     eax, 5  
    pop     ebp  
    ret
```

⁴⁰ X86 stack - http://en.wikibooks.org/wiki/X86_Disassembly/The_Stack

⁴¹ Source code: http://en.wikipedia.org/wiki/X86_calling_conventions

Codul în limbaj de asamblare nu este complet și în funcție de compilator, poate să fie mai mult sau mai puțin diferit, însă conține toate elementele necesare pentru a înțelege cum funcționează în fundal un apel de funcție.

Se pot observa următoarele lucruri:

1. Funcția începe cu două instrucțiuni: `push ebp` și `mov ebp, esp`. Aceste două instrucțiuni au rolul de a crea ceea ce se numește “stack frame”⁴². Cu alte cuvinte, pentru fiecare apel de funcție vom avea o zonă de stivă proprie apelului, delimitată de către regiștrii EBP și ESP
2. Funcția se termină cu instrucțiunile: `pop ebp` și `ret`. Prima instrucțiune are rolul de a face trecerea stivei la stiva funcției anterioare apelului funcției curente iar instrucțiunea “`ret`” e folosită pentru a returna o valoare și a urma execuția în locul următor apelului funcției curente. Valoarea returnată va fi disponibilă în registrul EAX
3. Există trei apeluri ale instrucțiunii `push`. După cum se poate observa în codul C, funcția “`callee`” este apelată cu 3 parametri: 1, 2 și 3. După cum se poate vedea în codul în limbaj de asamblare, acești parametri sunt puși pe stivă, însă în ordine inversă, de la ultimul spre primul. Această funcționalitate este folosită de către apelul “`cdecl`” al unei funcții, dar și de un apel “`stdcall`”
4. Urmează apoi instrucțiunea “`call`”, instrucțiune care apelează funcția dorită cu parametrii proaspăt adăugați pe stivă
5. Ceea ce face diferența între `cdecl` și `stdcall` este instrucțiunea următoare apelului de funcție: `add esp, 12`. Această instrucțiune va adăuga valorii registrului ESP valoarea 12, adică dimensiunea celor trei parametri cu care a fost apelată funcția “`callee`”. Cu alte cuvinte va elibera stiva, o va curăța de valorile parametrilor cu care a realizat apelul
6. Instrucțiunea `add eax, 5` este echivalentul codului C: `ret += 5`, valoare care urmează a fi returnată

⁴² Call stack - http://en.wikipedia.org/wiki/Call_stack

4.4. Interceptarea apelurilor

Am observat cum funcționează un apel de funcție, vom vedea acum cum se pot intercepta apelurile de funcții prin metoda “Inline patching”.

Un apel de funcție se realizează astfel:

```
push 3
push 2
push 1
call callee
```

Iar corpul unei funcții începe astfel:

```
push ebp
mov ebp, esp
```

Pentru a putea intercepta un apel de funcție, vom înlocui primele instrucțiuni din corpul funcțiilor pe care le vom intercepta cu o instrucțiune “jmp”, instrucțiune care va redirecționa execuția codului către o funcție din biblioteca ce conține codul nostru. După cum s-a înțeles, înainte de un apel de funcție, parametrii cu care este apelată respectiva funcție sunt puși pe stivă. Dacă vom redirecționa apelul funcției către o funcție scrisă de noi, atunci vom avea parametri cu care a fost apelată funcția respectivă pe stivă.

Înainte de a realiza acea înlocuire trebuie să ne gândim la faptul că vom fi probabil nevoiți să apelăm funcția originală, iar dacă vom înlocui primii octeți din corpul funcției nu o vom mai putea apela în mod corect. Astfel, înainte de înlocuirea primilor octeți vom face o copie a octeților pe care îi vom înlocui.

O interceptare de funcție va arăta astfel:

1. Se face o copie a primilor octeți din corpul funcției
2. Se înlocuiesc primii octeți cu o instrucțiune jmp către codul nostru
3. Când se execută codul nostru, restaurăm octeții pe care i-am înlocuit
4. Facem ce dorim cu parametri cu care a fost apelată funcția
5. Apelăm funcția originală, sau nu, dacă utilizatorul nu dorește acest lucru
6. Punem din nou instrucțiunea jmp

Primul pas pe care trebuie să îl facem este să creăm instrucțiunea “jmp”. Instrucțiunea este una simplă, are cinci octeți după cum urmează:

- 0xE9 – Octetul care identifică instrucțiunea
- 0x00 0x00 0x00 0x00 – Patru octeți care identifică adresa de memorie la care se sare

Un lucru foarte important despre această instrucțiune îl reprezintă faptul că adresa la care urmează să se sară este una relativă la poziția curentă. Astfel, dacă funcția pe care o interceptăm se afla la adresa 0x10000000 și funcția din DLL-ul injectat de noi în memorie se află la adresa 0x40000000, atunci va trebui să calculăm distanța dintre cele două funcții pentru a realiza un salt corect, deci vom sări peste 0x30000000 octeți.

Instrucțiunea va arăta astfel: jmp 0x30000000 și codul mașină, valorile prin care putem indica procesorului această instrucțiune în memorie sunt: 0xE9 0x00 0x00 0x00 0x30. Se poate observa că valoarea adresei este în little-endian, adică în ordinea inversă a octeților.

Un alt lucru pe care va trebui să îl facem pentru a putea modifica instrucțiunile funcției este să modificăm permisiunile zonei de memorie în care este memorat corpul funcției. Mai exact, funcția este memorată într-o zonă de memorie cu privilegii doar de citire și de execuție, deoarece în mod normal nu e necesar privilegiul de scriere pentru zona respectivă de memorie. Pentru a marca drept zonă de memorie “writeable” zona de memorie în care se află funcția, vom folosi funcția sistemului de operare care ne permite să schimbăm permisiunile unei pagini de memorie: VirtualProtect.

VirtualProtect:⁴³

```
BOOL WINAPI VirtualProtect(  
    _In_ LPVOID lpAddress,  
    _In_ SIZE_T dwSize,  
    _In_ DWORD flNewProtect,  
    _Out_ PDWORD lpflOldProtect  
);
```

Funcția VirtualProtect are următorii parametri:

- **lpAddress**: Adresa de memorie pentru care dorim să schimbăm permisiunile
- **dwSize**: Dimensiunea pentru care dorim să schimbăm permisiunile
- **flNewProtect**: Noile permisiuni pe care le dorim (PAGE_EXECUTE_READWRITE)
- **lpflOldProtect**: Pointer către o variabilă care după apelul funcției va conține privilegiile pe care le avea anterior zona respectivă de memorie

Urmând pașii discutați anterior, vom putea intercepta orice funcție și îi vom putea modifica comportamentul în funcție de propriile necesități.

⁴³ VirtualProtect - <http://msdn.microsoft.com/en-us/library/windows/desktop/aa366898%28v=vs.85%29.aspx>

Exemplu de cod folosit pentru a intercepta funcții:

```
// Functions from ncrypt.dll

vector<EXPORT_ENTRY> ncryptFunctions = GetDLLExports("ncrypt.dll");

for(size_t i = 0; i < ncryptFunctions.size(); i++)
{
    // SslEncryptPacket function

    if(ncryptFunctions[i].sName.compare("SslEncryptPacket") == 0)
    {
        // Original function pointer

        g_SslEncryptPacket_PFN_Original_Function = (unsigned char
*)ncryptFunctions[i].dwAddress;

        // Create jump

        jump = (DWORD)SslEncryptPacket_Hook -
(DWORD)g_SslEncryptPacket_PFN_Original_Function - 5;

        g_SslEncryptPacket_JMP_Bytes[0] = (char)0xE9;
        memcpy(&g_SslEncryptPacket_JMP_Bytes[1], &jump, 4);

        // Set page permissions

        VirtualProtect(g_SslEncryptPacket_PFN_Original_Function,
4096, PAGE_EXECUTE_READWRITE, &oldP);

        // Copy original bytes

        memcpy(g_SslEncryptPacket_Original_Bytes,
g_SslEncryptPacket_PFN_Original_Function, REPLACED_BYTES);

        // Set first hook

        SSLENCRYPTPACKET_HOOK_SET_JMP

        // Set pointers to functions

        g_pSslEncryptPacket =
(pfncSslEncryptPacket)g_SslEncryptPacket_PFN_Original_Function;
    }
}
```


Se folosesc ulterior două funcții:

1. `WinHttpRequest_Hook` – o funcție cu prototipul identic cu funcția pe care o interceptăm
2. `WinHttpRequest_Reinsert_Hook` – o funcție folosită pentru a reintroduce instrucțiunea `jmp` și de a redirecționa codul către instrucțiunea imediat următoare apelului de funcție original

Pentru a ajunge la rezultatul dorit, ne folosim de un lucru important legat de apelul funcțiilor din limbajul de asamblare: la executarea instrucțiunii `CALL`, procesorul pune pe stivă adresa imediat următoare instrucțiunii `CALL`, adresa la care trebuie să revină codul după apelul funcției apelate.

În funcția `WinHttpRequest_Hook` salvăm într-o variabilă o copie locală a acestei valori pe care o citim de pe stivă și o restaurăm ulterior în funcția `WinHttpRequest_Reinsert_Hook`. Pentru realizarea acestui lucru e necesară folosirea câtorva instrucțiuni de limbaj de asamblare în codul C/C++ în care am scris biblioteca de funcții care interceptează apelurile.

Un exemplu complet, codul funcției către care redirecționăm apelul unei funcții este următorul.

```
// Function that reinserts hook

extern "C" void WinHttpRequest_Reinsert_Hook()
{
    // push ebp
    // mov ebp, esp

    int* this_old_eip;
    __asm push eax;    // Save the result returned by previous function

    // Reinsert hook

    WINHTTPSENDREQUEST_HOOK_SET_JMP

    // Make the function return back to the good location (to legitimate code)

    __asm {
        mov this_old_eip, ebp; // this is the location where old EIP must be
placed for this function

        push eax;
        push ebx;
        push ecx;

        mov ebx, this_old_eip;
        mov ecx, g_WinHttpRequest_Old_EIP_Global;

        // EBP must be lifted 4 bytes on stack, g_PR_Read_Old_EIP_Global must
be inserted in the initial place of EBP

        sub ebp, 4;
        mov eax, [ebp+4];
        mov [ebp], eax;
        mov [ebx], ecx;

        pop ecx;
        pop ebx;
        pop eax;
    }

    __asm pop eax; // Restore the result returned by previous function

    // mov esp, ebp
    // pop ebp
}
```

```

// Our hook function

extern "C" BOOL WinHttpRequest_Hook(HINTERNET hRequest, LPCWSTR
pwszHeaders, DWORD dwHeadersLength, LPVOID lpOptional, DWORD dwOptionalLength,
DWORD dwTotalLength, DWORD_PTR dwContext)
{
    // push ebp
    // mov ebp, esp

    int *old_eip;
    BOOL res;

    // Get old EIP

    __asm
    {
        mov old_eip, ebp
    }

    old_eip++;
    g_WinHttpRequest_Old_EIP_Global = (int)*old_eip; // +4
    // Return address

    // Replace old EIP with our function that will reinsert the hook

    *old_eip = (int)WinHttpRequest_Reinsert_Hook;

    // Restore original bytes

    __asm pushad

    WINHTTPSENDREQUEST_HOOK_SET_ORIGINAL

    // Call original function

    res = g_pWinHttpRequest(hRequest, pwszHeaders, dwHeadersLength,
lpOptional, dwOptionalLength, dwTotalLength, dwContext);

    // Do things

    if(lpOptional != NULL && dwOptionalLength > 0)
    {
        // PCAPWinHttp.AddToFile((unsigned char *)lpOptional,
dwOptionalLength, true);

        FILE *pFile = fopen ("D:\\WinHttpRequest.txt", "a");
        fwrite (lpOptional , sizeof(char), dwOptionalLength, pFile);
        fclose (pFile);
    }

    __asm popad

    // Return result

    return res;

    // mov esp, ebp
    // pop ebp
}

```

5. CONCLUZII

Folosind Malware Monitor, utilizatorul are posibilitatea de a înțelege cum funcționează o anumită aplicație în fundal:

- Ce fișiere accesează, modifică sau șterge
- Ce date trimite către Internet, fie că sunt criptate sau nu
- La ce servere se conectează și multe altele

După cum s-a putut observa, deși pentru folosirea unei aplicații care folosește hook-uri sunt necesare de obicei cunoștințe avansate ale sistemului de operare Windows, ale formatului fișierelor Portable Executable sau a funcțiilor Windows API, Malware Monitor este însă o alternativă pentru persoanele care nu sunt familiare cu astfel de noțiuni.

Deși uneori pot să apară probleme legate de detecția acestor programe de către programele antivirus, de cele mai multe ori aceste produse software nu detectează activitățile proceselor în memorie ci detectează doar fișierele de pe disc.

Trebuie menționat faptul că sunt foarte multe programe care folosesc astfel de tehnici pentru a-și atinge scopul, cele mai importante exemple fiind programele antivirus și firewall, dar și virușii care fură date bancare. Antiviruşii monitorizează apelurile de funcții ale tuturor programelor care rulează pe calculator în speranța de a bloca eventuale activități malițioase.

Desigur, Malware Monitor nu trebuie confundat cu o aplicație sandbox. Cu alte cuvinte, nu executați pe propriul calculator un program malițios în speranța că Malware Monitor îi va bloca toate acțiunile malițioase, folosiți aplicația într-o mașină virtuală pentru teste.

Astfel spus, deși Malware Monitor este destinat persoanelor care posedă anumite cunoștințe ale sistemului de operare Windows, aplicația poate fi folosită cu ușurință de către oricine datorită modului simplu de folosire.

6. BIBLIOGRAFIE

- [1] Mark Russinovich, David Solomon, Alex Ionescu: Windows Internals, 5th edition, Microsoft Press, 2009, 1264 pagini
- [2] Jeffrey Richter, Christophe Nasarre: Windows via C/C++, 5th edition, Microsoft Press, 2007, 848 pagini
- [3] Mark Russinovich, Aaron Margosis: Windows Sysinternals Administrator's Reference, Microsoft Press, 2011, 496 pagini
- [4] Michael Howard, David LeBlanc: Writing secure code for Windows Vista, Microsoft Press, 2007, 224 pagini
- [5] MSDN – Microsoft Software Developers Network - <http://msdn.microsoft.com/en-us/default.aspx>
- [6] Microsoft PE and COFF specification - <http://msdn.microsoft.com/en-us/gg463119.aspx>