Introduzione all'analisi dei malware

(cc) BY-NC-ND

Introduzione

L'analisi dei malware è una attività svolta da appassionati e ricercatori, ma risulta utile anche per altre figure nel mondo dell'I.T., quali gli amministratori di sistema.
Attraverso l'analisi è possibile sviluppare delle contromisure effettive, strategie di risk management, un incident response dei sistemi compromessi, sviluppare a priori un incident response plan, creare delle “impronte comportamentali” da utilizzare in sistemi di HIPS o ottenere informazioni utili per il riconoscimento automatico da parte di software antivirus.

(cc) BY-NC-ND

Strumenti e Tecniche

Creazione di un ambiente sicuro per l'analisi analisi

È innanzitutto necessario creare un ambiente sicuro per poter analizzare il codice evitando rischi. Si possono utilizzare dei pc organizzati in una rete isolata esclusivamente per questo scopo, creando immagini dei dischi per ripristinare lo stato iniziale.
O una virtual machine, molti software di virtualizzazione permettono di creare delle *"snapshot"* tramite cui ripristinare i diversi stati.

BY-NC-ND

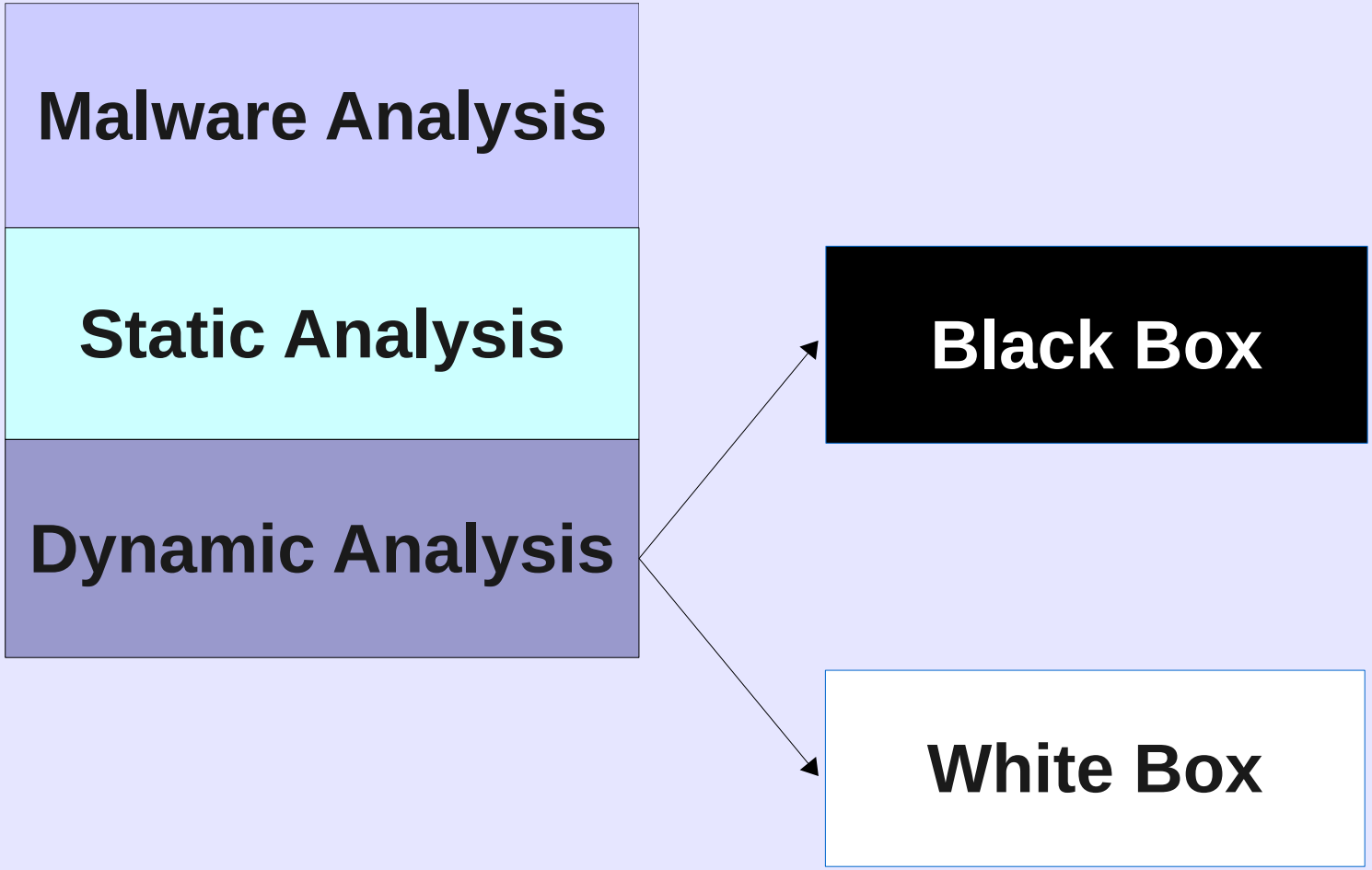
L'analisi statica può essere effettuata con le dovute precauzioni anche al di fuori dell'ambiente creato per l'analisi, per esempio, per maggiore sicurezza, su un SO differente da quello per cui è stato sviluppato il malware.
Per poter verificare come il malware interagisce può essere necessario allestire una rete locale isolata, in cui siano presenti server associati a determinati servizi (dns, server windows, server *nix...), utilizzando netcat per interagire o creando dei software server adatti.
L'approccio da adottare risulta spesso più semplice, basta una unica macchina virtuale su cui sia stata impostata l'accesso a nessuna rete o una rete che possa comunicare solo con la macchina host.
L'utilizzo delle virtual machine, non garantisce però una sicurezza perfetta. Alcuni malware sono in grado di riconoscere la presenza della virtual machine, per esempio verificando la presenza di driver specifici della virtual machine (le “guest additions” di virtualbox), la presenza di chiavi nel registro (vmware),

(cc) BY-NC-ND

utilizzando i meccanismi di comunicazione guest-host / host-guest, istruzioni privilegiate, o valori di ritorno di alcune istruzioni (CPUID, in Virtual Pc ritorna “Connectix CPU”).

Oltretutto, come suggerito nel paper “Attacks on Virtual Machine Emulators” (Peter Ferrie, Symantec), ci possono essere casi in cui il malware possa riuscire ad “evadere” l'ambiente della virtual machine: nel caso di una rete “host-only”, eventuali bug presenti nei driver di rete dell'host, o in servizi presenti sull'host possono essere sfruttati, attraverso l'invio di pacchetti appositamente creati, o sfruttando bug (per esempio degli input non controllati) presenti in alcuni componenti del software di virtualizzazione (nel 2005, un heap buffer corruption riguardante il componente *VMnat* di Vmware).
L'analisi di un malware può essere svolta utilizzando due metodi: analisi statica e dinamica (esistono anche metodi di analisi automatica, che non rientrano in queste categorie). Entrambe i metodi hanno i loro pregi e difetti, e vengono utilizzati in maniera complementare.

(cc) BY-NC-ND


Malware Analysis
Static Analysis
Dynamic Analysis
Black Box
White Box

(cc) BY-NC-ND

Static Analysis

Consiste nell'analizzare i file senza eseguirli, con un po' di attenzione questo tipo di analisi può essere effettuata anche nell'ambiente host (meglio se diverso dall'ambiente per cui è stato sviluppato il malware).
Script contenenti exploit possono essere analizzati tramite static analysis.
Gli strumenti utilizzati per questo tipo di analisi sono quindi, PE editor, disassemblatori, e decompilatori, e unpacker che non richiedano l'esecuzione del malware (per esempio upx).
È utile in questa fase prendere nota degli hash md5 dei vari file di cui è composto il malware.

(cc) BY-NC-ND

Alcuni strumenti utili durante questa analisi sono:
•*IDA Pro* (http://www.hex-rays.com/idapro/), un ottimo disassembler commerciale, al momento è possibile anche scaricare la versione 4.3 free.
•*CFF Explorer* (http://www.ntcore.com/exsuite.php), un ottimo PE editor, utilizza anche una lista di signatures per poter riconoscere con quale compilatore un file pe sia stato compilato, o se sia stato utilizzato un protettore/packer comune.
•Malzilla (http://malzilla.sourceforge.net) utile per visualizzare codice/script di exploit contenuti nelle pagine web, è molto utile per analizzare pagine che utilizzano javascript offuscati.
•PEid (http://www.peid.has.it)
•md5sum, sha1sum

(cc) BY-NC-ND

Dynamic Analysis

Spesso, una analisi statica non è sufficiente. I malware possono utilizzare tecniche che rendono complessa l'analisi (layer di compressione/cifratura, self modifying code...), in questi casi sarà quasi sempre necessario controllare il flusso dell'esecuzione tramite un debugger, per poter analizzare prima i vari layer presenti, e solo infine analizzare il “comportamento” dell'eseguibile.
Per questo viene utilizzata la dyniamic analysis, la quale richiede l'esecuzione del codice e può essere suddivisa in due metodi “black box” e “white box”.

(cc) BY-NC-ND

Black box

Si effettua senza avere necessità di comprendere i “meccanismi interni” del malware, ne vengono studiati gli effetti sul sistema. Durante l'esecuzione utilizzando strumenti quali packet sniffers, file e registry monitors e dopo l'esecuzione, cercando nuovi file/chiavi di registro presenti.

- Sysanalyzer (http://labs.idefense.com/software/malcode.php)
- Process Explorer (http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx)
- Process Monitor (http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx)
- TCPView (http://technet.microsoft.com/en-us/sysinternals/bb897437.aspx)
- Wireshark (http://www.wireshark.org/)
- oSpy (http://code.google.com/p/ospy/)

(cc) BY-NC-ND

White box

È più complesso, e richiede maggiori conoscenze rispetto all'analisi *black box*, ma consente di ottenere informazioni più approfondite sul funzionamento, sulla struttura e sul codice del malware.
Di solito non si ha a disposizione il sorgente del malware, quindi bisognerà effettuare una analisi file binario (il file eseguibile), dove l'analisi statica tramite il disassembler non ci abbia fornito abbastanza informazioni, è necessario l'uso di un debugger, per potere controllare l'esecuzione del codice a basso livello (assembly).

- Windbg (http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx)
- Ollydbg (http://www.ollydbg.de/)
- IDA Pro (http://www.hex-rays.com/idapro/)
- Rasta Ring 0 debugger (http://rr0d.droids-corp.org/)
- Syser Debugger (http://www.sysersoft.com/)
- PaiMei (http://pedram.openrce.org/code/paimei/)

(cc) BY-NC-ND

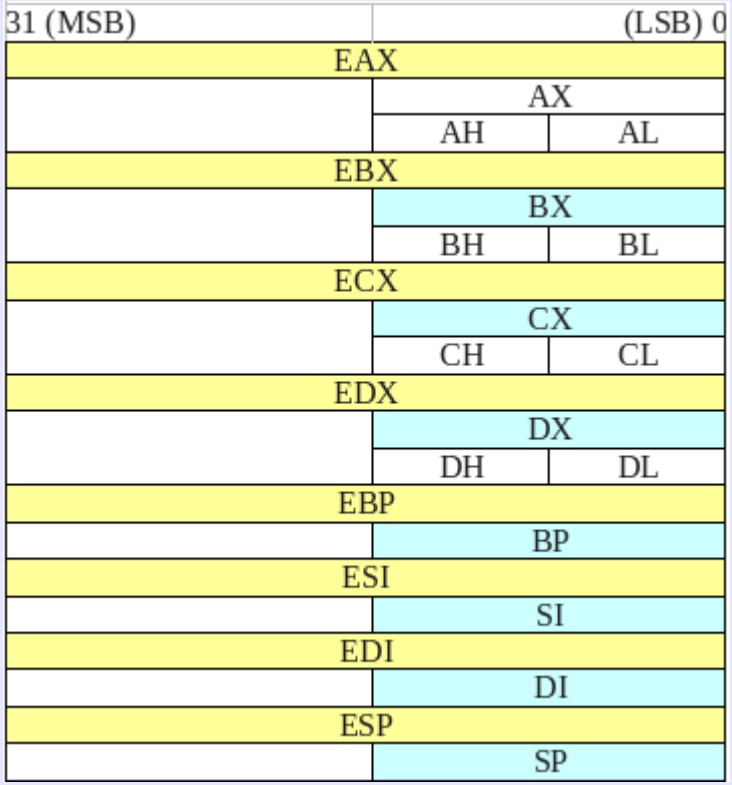
Introduzione al linguaggio assembly

L'assembly è un linguaggio a basso livello, molto vicino al linguaggio macchina. Nel linguaggio macchina, delle sequenze di bit specifiche sono associate a delle istruzioni basilari.
Il linguaggio assembly associa a delle istruzioni mnemoniche questi codici operativi (*opcodes*).
Per ogni architettura differente esiste quindi un differente linguaggio assembly, noi ci interesseremo della architettura intel x86 a 32 bit.
L'argomento è molto vasto, vedremo solo le caratteristiche più importanti, e soprattutto come analizzare un semplice programma scritto nel linguaggio di alto livello C.

(cc) BY-NC-ND

I Registri

I registri sono dei “contenitori”, come le variabili di un linguaggio di alto livello.



(cc) BY-NC-ND

I primi otto registri chiamati anche “*general-purpose registers*”, possono essere utilizzati in maniera generica, ma hanno anche degli usi specifici.

EAX: accumulatore
EBX: puntatore a dati
ECX: contatore
EDX: puntatore I/O
ESI: puntatore di origine per le operazioni sulle stringhe
EDI: puntatore di destinazione per le operazioni sulle stringhe
ESP: stack pointer
EBP: puntatore a dati nello stack

EIP è un registro particolare e contiene l'inidirizzo della prossima istruzione che deve essere eseguita.

(cc) BY-NC-ND

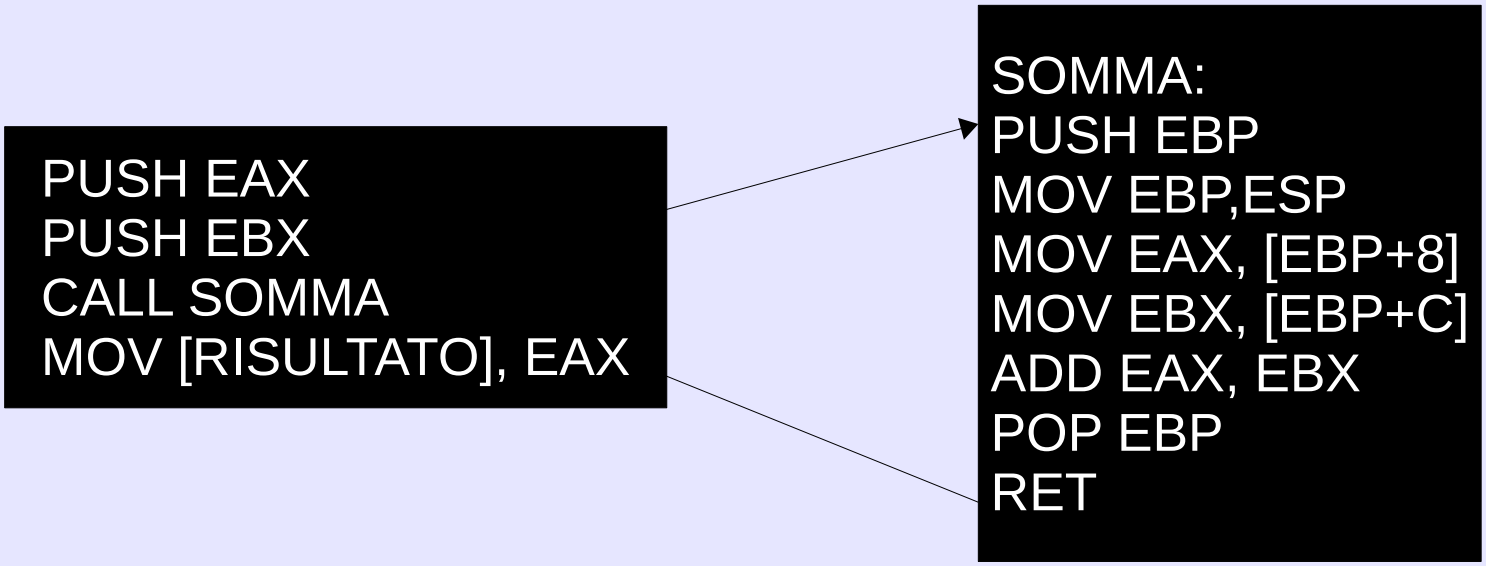
EFLAGS è un registro che contiene dei flag, quelli che ci interessano sono gli status flag e il flag df.
Gli status flags indicano il risultato di operazioni aritmetiche. Quelli che vengono usati maggiormente sono:
CF, carry flag: indica se una istruzione ha dovuto effettuare il riporto di un bit.
ZF, zero flag: indica se il risultato è zero.
SF, sign flag: indica se il risultato è positivo o negativo.

C'è anche un flag DF di direzione, controlla la direzione di lettura/scrittura per le istruzioni sulle stringhe.

BY-NC-ND

Lo Stack

È una struttura di tipo LIFO (Last In First Out) presente in memoria, lo stack pointer punta a quest'area di memoria, che solitamente viene manipolata attraverso le istruzioni di push e pop. Oltre a contenere dati contiene anche gli *indirizzi di ritorno delle call*. Infatti quando viene eseguita una istruzione CALL, per chiamare una sottoprocedura, viene messo nello stack l'indirizzo della istruzione successiva rispetto alla call. In questo modo, terminata la procedura, attraverso l'istruzione RET, il programma riprende l'esecuzione dalla istruzione immediatamente successiva.



BY-NC-ND

Le istruzioni

Nella sintassi intel il primo operando rappresenta la destinazione, il secondo l'origine dei dati.
MOV EAX, EBX, copia il contenuto di ebx in eax.
MOV EAX, [EBX] copia la dword puntata dall'indirizzo contenuto inebx, in eax.
JMP $1234, è un salto incondizionato all'indirizzo 1234.
CMP EAX, EBX confronta il contenuto di EAX e EBX, e modifica gli status flags di conseguenza.
Ci sono anche i *salti condizionati* il salto viene eseguito o meno a seconda del flag corrispondente.
JNZ/JNE salta solo se il risultato dell'ultima operazione è differente (se eax ed ebx nell'istruzione cmp sono differenti)
JLE, salta se minore o uguale...

(cc) BY-NC-ND

Da linguaggio di alto livello all'assembly

Vediamo ora un piccolo esercizio, per cominciare a prendere dimestichezza con l'assembler.

```
int somma(int a, int b){
	return (a+b);
}

int main(void){
	int i,j;
	for (i=0, j=0;i<5;i++){
		j = somma(i,j);
		printf("hello, %d\n",j);
	}
	return 0;
}
```

(cc) BY-NC-ND

Il risultato della compilazione è il seguente:

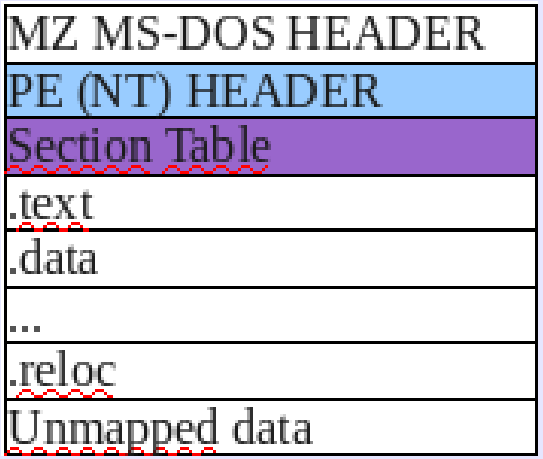
```
.text:00401740 main            proc near
.text:00401740
.text:00401740 var_8           = dword ptr -8
.text:00401740 var_4           = dword ptr -4          ;variabili locali
.text:00401740
.text:00401740                 push    ebp
.text:00401741                 mov     ebp, esp
.text:00401743                 sub     esp, 8
.text:00401746                 mov     [ebp+var_4], 0      ;inizializza i
.text:0040174D                 mov     [ebp+var_8], 0      ;inizializza j
.text:00401754                 jmp     short loc_40175F
.text:00401756 ;
```

(listato_1)

Le strutture di alto livello sono facilmente distinguibili all'interno del codice assembly.

(cc) BY-NC-ND

Il formato PE



È il formato utilizzato per i file eseguibili di windows. Le parti più interessanti sono sicuramente il pe header e la section table.
Le sezioni possono contenere codice o dati.
Del pe header in particolare ci interessano l'entry point, che descrive il punto da cui comincia il codice, l'image base, che specifica l'indirizzo dove verrà mappato in memoria il file.
Le entry della struttura IMAGE_DATA_DIRECTORY contenute nell'optional header, sono altrettanto interessanti, tra queste entry è presente IMAGE_DIRECTORY_ENTRY_IMPORT, o Import Table.

BY-NC-ND

La Import Address Table

La IAT è una tabella di “puntatori a funzioni esterne” che l'applicazione utilizza. La locazione in memoria delle dll, però, è dinamica. Quindi il linker non può conoscere a priori dove una certa funzione è in memoria. La tabella viene quindi creata in maniera dinamica, attraverso il nome della funzione e della dll. Il loader di windows, quindi sovrascriverà la IAT, con le locazioni ottenute. Ogni volta che una funzione viene richiamata, viene utilizzato il puntatore della iat, il metodo utilizzato può cambiare a seconda del compilatore e delle ottimizzazioni. È facile trovare per esempio uno schema simile a questo:

```
CALL j_funzione
...
j_funzione:
JMP [iat->funzione]
```

(A meno che l'indirizzo della procedura non venga ottenuto a runtime, per esempio attraverso GetProcAddress)

(cc) BY-NC-ND

PE Packers

I pe packers (anche i crypters/protectors) sono dei software che rendono più complessa l'analisi e il riconoscimento tramite signatures da parte degli antivirus.
Il contenuto del file exe originale viene compresso/cifrato. All'interno del nuovo file eseguibile saranno presenti i dati originali compressi e uno stub che provvede alla decompressione.
Lo stub utilizza una propria IAT (spesso con poche funzioni come GetProcAddress, LoadLibraryA e VirtualAlloc), una volta eseguita la decompressione in memoria, la IAT viene popolata con le entry dell'eseguibile originale.
L'analisi statica in questo caso è quindi inutile, sarà necessario prima rimuovere il layer (o più di uno) di compressione/protezione per rendere più semplice l'analisi in seguito.

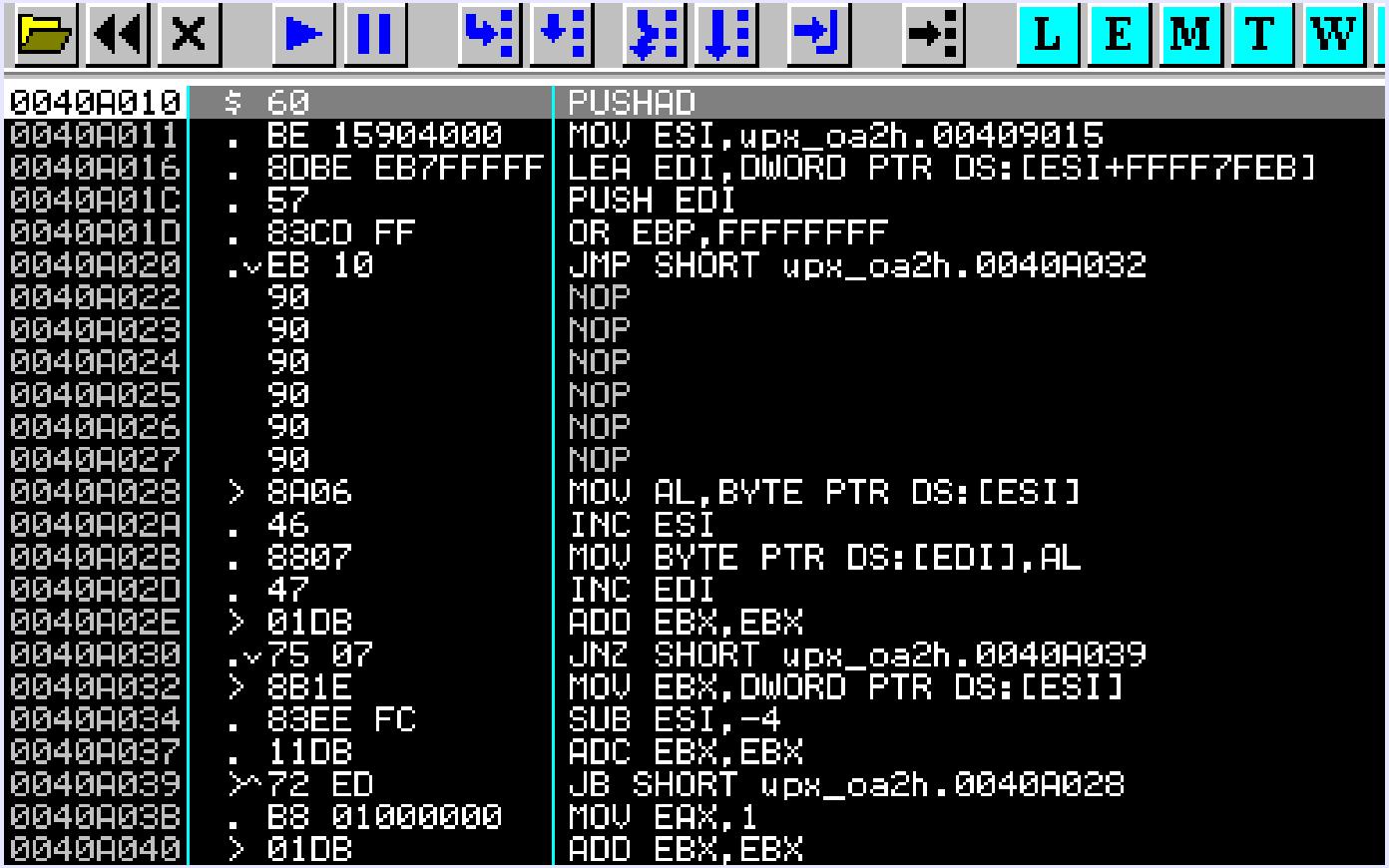
BY-NC-ND

Una volta analizzato il layout del pe, se sembra sia stato utilizzato un packer possiamo utilizzare un tool come PEid o CFF Explorer, per poter identificare il packer usato. (l'identificazione è basata su signatures, quindi è abbastanza semplice modificare il pe in modo da non rendere possibile l'identificazione del packer).
Qualunque sia il packer il codice/dati originali devono essere in memoria per poter essere utilizzati, per poter effettuare l'unpacking dobbiamo conoscere il momento esatto in cui i dati sono presenti in memoria. È anche possibile che i dati siano presenti in memoria solo per poco tempo.
Per eseguire l'unpacking è necessario quindi raggiungere l'entry point originale (OEP) e dumpare il processo, modificare il pe header in modo da ripristinare l'entry point, e ripristinare una IAT valida.
Ci sono dei tool appositi per ricostruire la Import Table in maniera automatica, Revirgin e ImpRec.

(cc) BY-NC-ND

UPX Unpacking

Notiamo immediatamente che il file PE è compresso, ci sono poche import, i dati presenti nella resource directory non sono consistenti. Una veloce occhiata all'entry point rivela lo stub di upx:



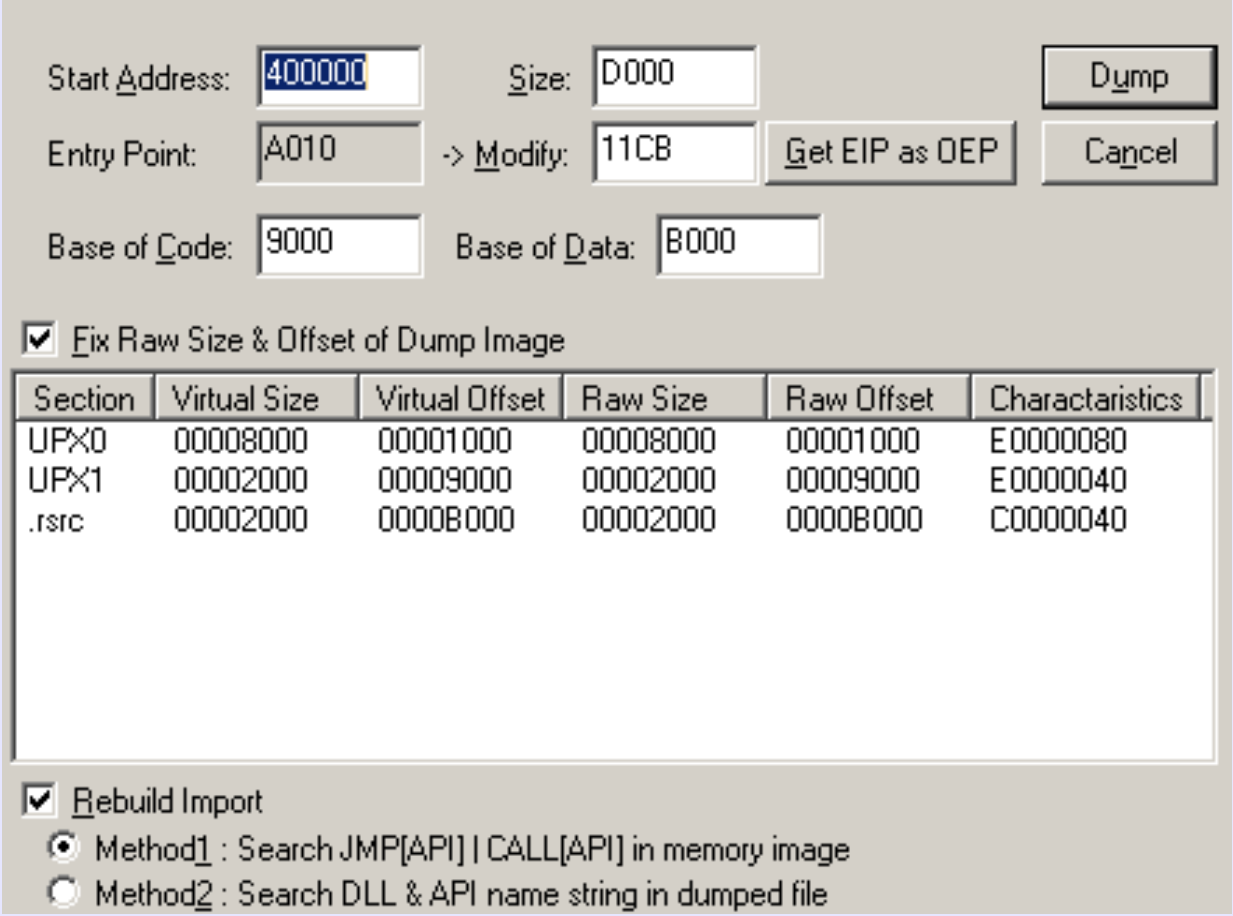
(cc) BY-NC-ND

Il pushad all'inizio salva i valori presenti nei registri all'inizio dell'esecuzione, upx salva nello stack questi valori e li ripristina immediatamente prima di saltare all'oep. Per raggiungere l'entry point, quindi la maniera più semplice è cercare per le due istruzioni POPAD / JMP, e mettere un breakpoint sul jmp, o eseguire l'istruzione pushad e mettere un memory breakpoint sull'accesso a quella zona di memoria, Lasciare eseguire lo stub di upx e arrivati al breakpoint, steppare fino all'oep.

```
0040A18C   . 58             POP EAX
0040A18D   . 61             POPAD
0040A18E   . 8D4424 80      LEA EAX,DWORD PTR SS:[ESP-80]
0040A192   > 6A 00          PUSH 0
0040A194   . 39C4           CMP ESP,EAX
0040A196   .^75 FA          JNZ SHORT upx_oa2h.0040A192
0040A198   . 83EC 80        SUB ESP,-80
0040A19B   .-E9 2B70FFFF    JMP upx_oa2h.004011CB
```

(cc) BY-NC-ND

Arrivati all'oep possiamo utilizzare il plugin OllyDmp, e utilizzare l'opzione integrata nel plugin per ricostruire la IAT (o utilizzando ImpRec).



(cc) BY-NC-ND

Antidebug

Ci sono diversi metodi tramite cui il malware può riconoscere la presenza di un debugger:

- IsDebuggerPresent
- RDTSC, QueryPerformanceCounter, GetTickCount
- PEB->IsDebugged
- NtGlobalFlags
- HeapFlags

...

Una reference utile sull'argomento è “Windows Anti-Debug Reference” (Nicolas Falliere)
http://www.securityfocus.com/infocus/1893

(cc) BY-NC-ND

(cc) BY-NC-ND