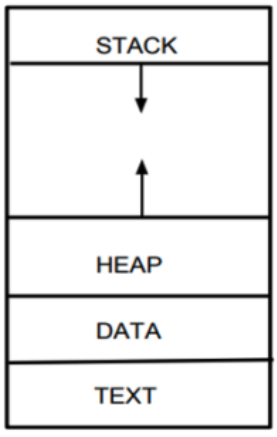
Approfondimenti - Buffer Overflow

Index

Introduction

L'area di memoria di un processo caricato in memoria è diviso nelle sezioni seguenti



Stack

Lo stack nello specifico è costituito da **frames**. In ciascun frame sono contenuti i parametri passi alla funzioni, variabili locali, indirizzo di ritorno e instruction pointer Sono inoltre presenti due puntatori:

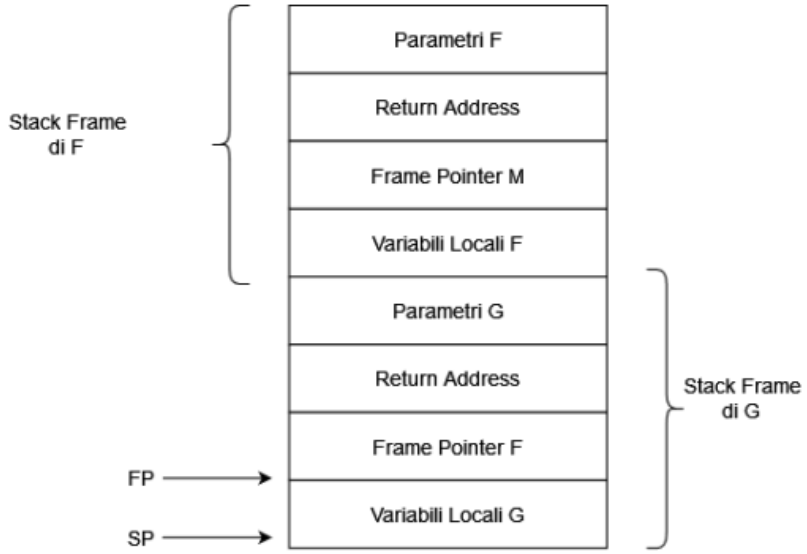
- stack pointer → punta alla cima dello stack (indirizzo più basso)
- frame pointer → punta alla base del frame corrente

Chiamata di funzione

Quando all'interno di una chiamata di funzione sono presenti dei parametri, questi sono aggiunti allo stack. Inoltre quando avviene una chiamata vengono memorizzati sullo stack anche:

- indirizzo di ritorno
- puntatore allo stack frame
- spazio ulteriore per le variabili locali della funzione chiamata

Nel caso in cui vengono chiamate due funzioni G e F si ha



Il problema

Cosa succede adesso?

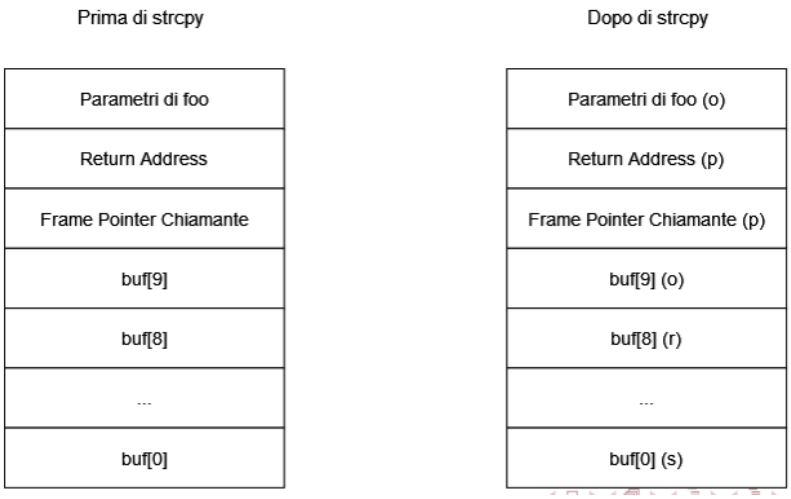
```
void foo(char *s) {
        char buf[10];
        strcpy(buf, s);
        printf("buf is %s\n", s);
}
```

```
foo("stringatroppolungaperbuf");
```

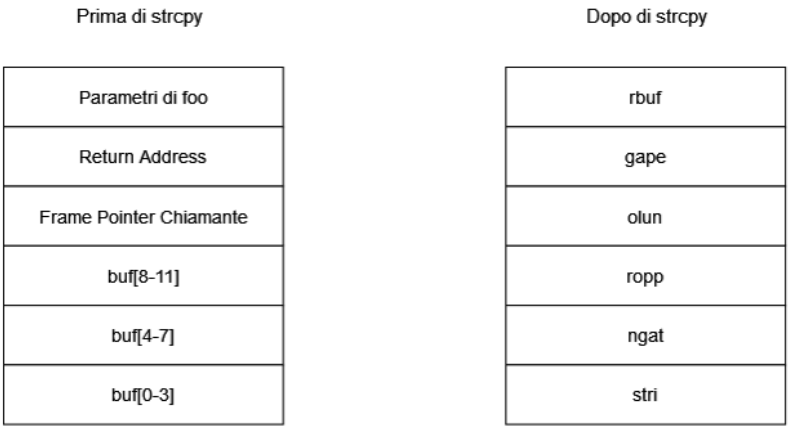
In questo caso stiamo inserendo troppi dati rispetto alla dimensione del buffer però il computer non sa la dimensione del buffer (per lui sono solo indirizzi di memoria). Dunque continua a copiare `"stringatroppolungaperbuf"` a partire dal primo indirizzo di memoria di `buf[]`, fino a che non ha occupato tutti gli indirizzi di memoria del buffer, e poi continua a sovrascrivere qualsiasi cosa trovi, finché non completa l'operazione richiesta

Evoluzione dello stack

Tra parentesi si trovano i valori che prendono i diversi indirizzi dello stack (semplificato)



In realtà un indirizzo conterrà più lettere, in base alla dimensione delle parole in memoria



Stack Smashing - conseguenze

Di solito, fare overflow di un buffer nel modo visto sopra, porta alla terminazione del programma, tuttavia, se i dati che sono usati nell'overflow del buffer sono preparati in

modo accurato, è possibile eseguire codice arbitrario

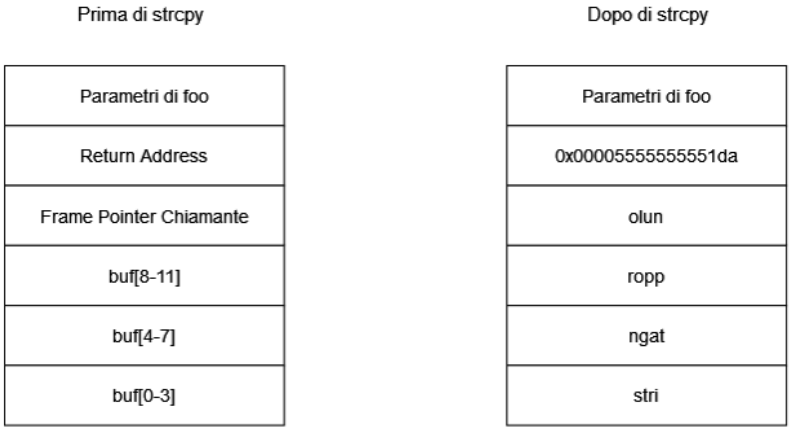
Il problema

Cosa succede se cambiamo la stringa?

```
void foo(char *s) {
        char buf[10];
        strcpy(buf, s);
        printf("buf is %s\n", s);
}

foo("stringatroppolun\xda\x51\x55\x55\x55\x55\x00\x00");
```

Facendo in questo sovrascriviamo l'indirizzo di ritorno a piacere



Esecuzione di codice arbitrario

Modificare l'indirizzo di ritorno arbitrariamente è utile, ma come eseguiamo del codice arbitrario?
Per farlo si hanno diverse tecniche:

- Shellcode
- return-to-libc
- Stack frame pointer replacement
- return-oriented programming (ROP)

Shellcode

Con questa modalità un piccolo (deve rientrare nelle dimensioni del buffer) pezzo di codice viene eseguito quando si sfrutta una vulnerabilità (es. buffer overflow) per attaccare un sistema

Viene chiamato **shellcode**, perché solitamente avvia una command shell, dalla quale l'attaccante può prendere il controllo della macchina

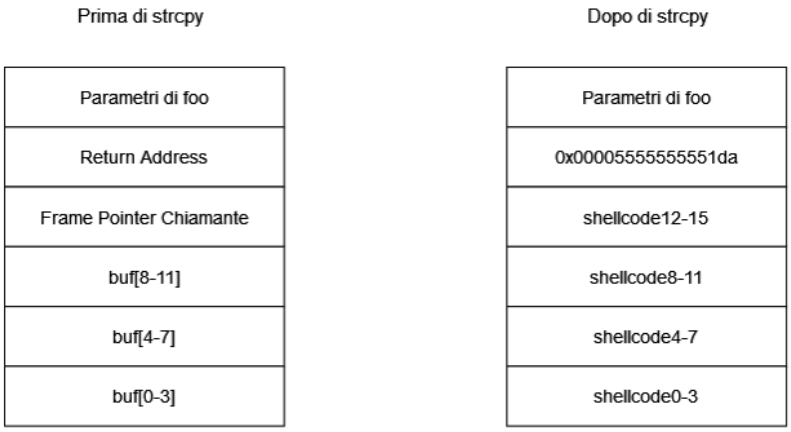
L'idea consiste nell'inserire del codice eseguibile arbitrario nel buffer, e cambiare il return address con l'indirizzo del buffer

Assumendo che l'indirizzo di `buf` sia `0x00005555555551da` possiamo eseguire così l'attacco

```
void foo(char *s) {
        char buf[10];
        strcpy(buf, s);
        printf("buf is %s\n", s);
}

foo("<shellcode>\xda\x51\x55\x55\x55\x55\x00\x00");
```

In questo modo, una volta completata la chiamata a `foo()`, il processore salterà all'indirizzo `0x00005555555551da` ed eseguirà il codice che trova (shellcode)



return-to-libc

Poiché non è sempre possibile inserire shellcode arbitrario (es. buffer piccolo, meccanismi di difesa), si può utilizzare come indirizzo di ritorno l'indirizzo di una funzione di sistema utile per un attacco (es. system). Infatti le librerie dinamiche e di sistema sono sempre presenti in RAM e sono raggiungibili dal nostro processo
Viene chiamato **return-to-libc** in quanto solitamente modifica l'indirizzo di ritorno con l'indirizzo di una funzione standard della libreria C

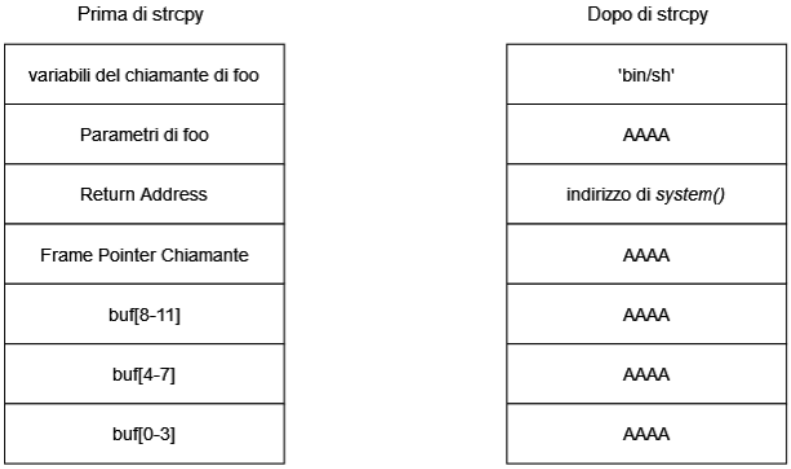
Assumendo di conoscere l'indirizzo di `system()`, possiamo effettuare così l'attacco

```
void foo(char *s) {
        char buf[10];
        strcpy(buf, s);
```

```
        printf("buf is %s\n", s);
}

foo("AAAAAAAAAAAAAAAA<indirizzo di system>AAAA'bin/sh'");
```

In questo modo, una volta completata la chiamata a `foo()`, il processore salterà all'indirizzo di `system()` e ne eseguirà il codice usando come parametro `bin/sh`



Contromisure

Esistono due tipi di contromisure per il buffer overflow:

- difese a tempo di compilazione
- difese a tempo di esecuzione

Tempo di compilazione

A tempo di compilazione è risolvibile usando **linguaggi di programmazione e di funzioni sicure**, infatti in C l'overflow è possibile in quanto ci sono anche funzioni che spostano dati senza limiti di dimensione, oppure con l'uso dello **Stack Stashing Protection**, tramite il quale il compilatore inserisce del codice per generare un valore casuale (*canary*) a runtime. Il valore *canary* viene inserito tra il frame pointer e l'indirizzo di ritorno, se il valore canary viene modificato prima che la funzione ritorni, viene interrotta l'esecuzione (vuol dire che è stato sovrascritto da un possibile attacco)

Tempo di esecuzione

A tempo di esecuzione si hanno due possibili soluzioni:

- **Executable Space Protection**
- **Address Space Layout Randomization**

Tramite lo *Executable Space Protection* il SO marca le pagine/segmenti dello stack e heap come non eseguibile (infatti stack e heap non contengono codice eseguibile), quindi se un attaccante cerca di eseguire del codice nello stack (shellcode), il sistema terminerà il processo con un errore (return-to-libc funziona comunque)
Tramite l'*Address Space Layout Randomization* vengono randomizzati, ad ogni esecuzione, gli indirizzi dove sono caricati i diversi segmenti del programma (stack, heap, …), quindi se l'attaccante non sa dove inizia lo stack, è molto più difficile indovinare l'indirizzo del buffer contenente lo shellcode ed anche l'indirizzo delle librerie standard per attacco return-to-libc