

# Buffer overflow: attacco e difesa

a cura di **Simone Caligola**(A40/000196)



Università degli studi di Catania  
Facoltà di scienze MM.FF.NN  
C.d.s in Informatica Applicata  
Sicurezza dei sistemi informatici 1  
A.A 2008/2009

# Indice

- 1. Sicurezza informatica [4-5]
  - 1.1 Definizione[4]
  - 1.2 Vincoli di difesa e sicurezza[4-5]
  - 1.3 Sicurezza dei programmi[5]
  
- 2. Buffer overflow[6-13]
  - 2.1 Definizione di buffer[6]
  - 2.2 Organizzazione di un processo in memoria[6-7]
  - 2.3 Stack[7]
  - 2.4 Heap[7-8]
  - 2.5 Buffer overflow: definizione[8]
  - 2.6 Stack overflow[9]
    - 2.6.1 Inserire codice arbitrario nel programma[9-10]
    - 2.6.2 Shellcode[10]
    - 2.6.3 Far saltare il programma al codice d'attacco[10-11]
    - 2.6.4 Combinare le due fasi[11]
  - 2.7. Heap overflow[11-13]
  
- 3. Difesa da buffer overflow[14-15]
  - 3.1 Scrivere codice sicuro[14-15]
  - 3.2 Buffer non eseguibile[15]
  - 3.3 Controllo della dimensione degli array[15]
  - 3.4 Controllo d'integrità dei puntatori[15]
  
- 4. Un esempio di heap overflow[16-18]
  
- 5. Conclusioni[19]
- 6. Riferimenti[20]

## Un preambolo

Prima di trattare nello specifico le tecniche del buffer overflow occorre introdurre il problema più generale della sicurezza informatica, ponendo maggiore enfasi sul problema della sicurezza dei programmi che diventa ogni giorno più ricorrente nella progettazione di software sicuro e di qualità. Come vedremo tra poco la sicurezza informatica in generale tenta di mantenere intatti alcuni aspetti di protezione che stanno alla base della salvaguardia dei dati. Vi sono moltissime tecniche in grado di mettere in pericolo la sicurezza informatica, ma in questa sede ci occuperemo di una di queste, riconosciuta come una delle più avanzate e pericolose poiché, se messa a punto bene, può mettere a repentaglio l'intero sistema. Questa tecnica sfrutta delle falle nei software, che corrispondono ad errori di programmazione molto comuni riguardanti, nella maggior parte dei casi, il mancato controllo nella ricezione dei dati dall'esterno e il rispettivo overflow(traboccamento) di un buffer. Ne consegue la scrittura o la sovrascrittura di dati in porzioni di memoria non lecite e la conseguente esecuzione di codice arbitrario. Le conseguenze possono essere disastrose se si pensa che questo tipo di attacco permette all'attaccante esperto di avere l'accesso alla macchina(da remoto) e di averne, nella peggiore delle ipotesi i privilegi di amministratore. Verrà trattato anche un esempio di heap overflow basato sulla sovrascrittura di un puntatore.

# 1. Sicurezza informatica

## Contenuto:

- Definizione di "sicurezza informatica"
- Obiettivi della protezione
- Sicurezza dei programmi

## 1.1 Definizione

E' stato affermato che qualsiasi sistema informatico è caratterizzato da debolezze teoriche che lo rendono un probabile bersaglio da attacchi atti alla violazione di dati riservati. La sicurezza informatica è quella branca dell'informatica che si occupa della salvaguardia di questi sistemi da potenziali violazioni. Occorre, però, capire cosa significa il termine "sicurezza" e in cosa si esplicita nell'ambito dell'informatica. Il termine "sicurezza" viene utilizzato molto spesso nella vita quotidiana e possiede molti significati. Ad esempio può essere utilizzato nell'ambito della "sicurezza fisica" dei bambini e si traduce nella loro protezione da eventuali danni, o nella "sicurezza finanziaria" se si fa riferimento a tutte quelle strategie atte ad una attenta gestione del patrimonio familiare. In definitiva il termine sicurezza viene sempre utilizzato come risposta ad eventuali rischi e quindi al mantenimento di alcuni aspetti base, che se violati possono portare ad uno stato di "non sicuro". Per quanto riguarda la sicurezza informatica i tre aspetti fondamentali sono: riservatezza, integrità e disponibilità.

## 1.2 Vincoli di sicurezza e difesa

La riservatezza è l'aspetto più semplice da capire e si basa sul fatto che l'accesso a determinate risorse o dati, è consentito solamente alle persone autorizzate. Tale vincolo è gestito diversamente a seconda del sistema che si deve gestire e, in molti casi, può essere difficoltoso rispettarlo. L'integrità è più difficile da definire, tuttavia è facile da capire se si pensa che tale vincolo è rispettato quando un elemento è utilizzato secondo alcuni criteri e gestito secondo uno schema di autorizzazioni. Infine la disponibilità è quel vincolo che, in riferimento ad un dato elemento, denota la sua possibilità di utilizzo in base a determinati criteri di accettabilità e coerenza. Detto questo, occorre in questa sede descrivere brevemente i due concetti di "sicurezza passiva" e "sicurezza attiva". Il concetto di sicurezza passiva è molto semplice e si basa su un atteggiamento difensivo atto alla difesa del sistema, il cui obiettivo è quello di proteggere le risorse da eventuali accessi non autorizzati per mezzo di adeguati sistemi di protezione, quali l'identificazione etc... Per sicurezza attiva si intendono, invece, quegli strumenti mediante i quali le informazioni e i dati

sono intrinsecamente sicuri, mantenendo inalterati i vincoli di confidenzialità e integrità. Sembra logico, a questo punto, che i due concetti sopraccitati risultino complementari nel mantenimento di un sistema in uno stato “sicuro”.

### 1.3 Sicurezza dei programmi

Il problema della sicurezza dei programmi si è posto agli sviluppatori di software in conseguenza alla maggiore diffusione dei sistemi informatici e dall'avvento di internet. A tal proposito si pone il problema di assicurare la protezione dell'invio e della ricezione di dati confidenziali, soprattutto nella rete. Ma cosa vuol dire che un programma è “sicuro”? La valutazione di un software potrebbe essere effettuata in base ai criteri di sicurezza utilizzati nella sua progettazione, ma il problema di questa analisi sta nella sua intrinseca soggettività derivata dal fatto che valutatori diversi potrebbero esigere condizioni diverse, dando luogo a diverse interpretazioni sulla sicurezza del programma testato. Un altro metodo potrebbe tradursi nell'attenersi a dei requisiti richiesti dal richiedente, in modo da fornire tutte quelle soluzioni atte ad soddisfare le specifiche. Comunque, in generale, le due caratteristiche fondamentali che esplicano il concetto di sicurezza dei programmi sono:

**Safety(sicurezza):** una serie di accorgimenti atti ad eliminare la produzione di danni irreparabili al sistema.

**Reliability(affidabilità):** prevenzione da eventi che possono produrre danni di qualsiasi gravità al sistema.

Quindi un programma è tanto più sicuro quanto minori sono le possibilità di successo di un guasto e la gravità del danno conseguente al guasto stesso. Come si era già accennato risulta utile effettuare dei controlli, in modo da ricercare la maggior parte dei difetti presenti per passare alla loro eliminazione. Esistono diversi modelli basati su due metodi differenti:

**Semantic-based security model(modelli di sicurezza basati sulla semantica):** la sicurezza del programma viene misurata in termini di comportamento del programma stesso.

**Security-typed language(modelli di sicurezza basati sul linguaggio):** i tipi delle variabili sono seguiti dall'esplicazione delle politiche adottate per l'uso dei dati tipati.

In questo modo un programma per essere sicuro dev'essere controllato nelle sue specifiche e dev'essere privo di difetti nel codice.

## 2. Buffer overflow(BOF)

### Contenuto:

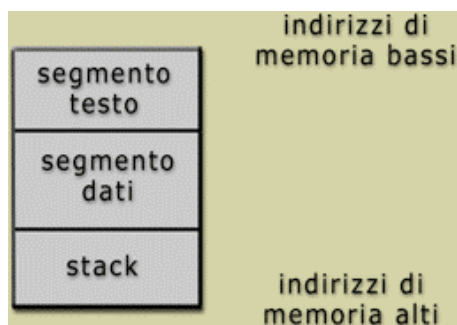
- Definizioni
- Organizzazione dei processi
- tipi di buffer overflow

### 2.1 Definizione di buffer

Un buffer, termine che in inglese vuol dire letteralmente “tampone” e in italiano “memoria tampone”, è un blocco di memoria contigua che contiene più istanze dello stesso tipo di dato e viene usato come memoria temporanea per effettuare I/O. Nei linguaggi di programmazione tipo il C un buffer può essere rappresentato da un array. L’overflow di un buffer consiste nel riempirlo oltre la sua massima capacità.

### 2.2 Organizzazione di un processo in memoria

Un processo prima di poter essere eseguito dev’essere caricato in memoria centrale. Un processo a differenza di un programma non è caratterizzato solamente dal suo codice sorgente ma ad esso sono associate diverse informazioni tra cui l’attività corrente, rappresentata dal contatore di programma e dal contenuto dei registri della CPU ; normalmente comprende anche la propria pila(**stack**), contenente a sua volta dati temporanei come i parametri di un metodo, gli indirizzi di rientro e le variabili locali, e una sezione di dati contenente le variabili globali. Un processo può includere anche una sezione di memoria chiamata **heap**, ossia la memoria allocata dinamicamente dal processo durante la sua esecuzione. La struttura di un processo in memoria è rappresentata nella figura sotto.



Nella maggior parte dei casi le aree di memoria sfruttate per effettuare un attacco di buffer overflow sono la regione dello stack(pila) e la regione dello heap. In entrambi i casi si sfruttano le vulnerabilità del software causando un traboccamento delle suddette aree di memoria.

## 2.3 Stack

Lo stack è una struttura dati astratta, avente una proprietà fondamentale: “last in first out”(LIFO). Questa proprietà riguarda l’inserimento dei dati nella struttura e la loro estrazione, entrambi avvengono in cima alla struttura al contrario delle code di tipo FIFO. Le due operazioni di inserimento ed estrazione sono rispettivamente: “push” e “pop”. Ma perché i processi utilizzano lo stack? I moderni linguaggi di programmazione utilizzano i costrutti di chiamate a funzione. Una chiamata a funzione altera il flusso esecutivo delle istruzioni come una istruzione di salto(jump) però, rispetto a quest’ultima, la funzione deve ritornare al punto chiamante del programma per poter eseguire la successiva istruzione. Questo meccanismo viene implementato tramite uno stack. Lo stack è memorizzato su locazioni contigue di memoria e può contenere le variabili allocate dinamicamente dal processo e memorizza i parametri delle funzioni e i valori restituiti. Il registro che punta alla cima dello stack viene chiamato “stack pointer”. Uno stack è caratterizzato da più segmenti logici chiamati “stack frame” allocati durante le operazioni di push, la loro dimensione varia a seconda del kernel. A seconda dell’implementazione lo stack cresce verso il basso o verso l’alto, ad esempio nelle architetture intel lo stack cresce verso il basso. Uno stack, oltre ad essere caratterizzato dallo stack pointer, possiede un frame pointer che punta ad una locazione fissa della memoria. Quando avviene una chiamata a funzione, la prima cosa che si fa è salvare il frame pointer precedente in modo da poterlo ripristinare al ritorno, poi si copia lo stack pointer sullo stack frame in modo da avere il nuovo stack frame e poi si incrementa lo stack pointer in modo da farlo puntare alla successiva variabile locale. Al termine della procedura lo stack viene ripulito, ad esempio le istruzioni intel per fare ciò sono ENTER e LEAVE.

## 2.4 Heap

Per capire di cosa si tratta occorre capire, in primis, cosa è l’allocazione dinamica della memoria. Quest’ultima è quel tipo di memoria allocata durante l’esecuzione di un programma, al quale serve per salvare dati utili alla sua elaborazione. La particolarità di questo tipo di memoria è quella di essere persistente, cioè non viene deallocata fino a quando il programmatore non adopera esplicitamente una chiamata atta a liberarla. Questo fattore la differenzia dallo stack, dove la memoria viene liberata dopo la terminazione della procedura che lo ha utilizzato. L’allocazione della memoria dinamica avviene in un’area vuota chiamata heap. La dimensione della memoria da

allocare può essere decisa a runtime e l'accesso a tale porzione di memoria avviene tramite un riferimento. Le modalità di allocazione in questo tipo di memoria sono diverse e dipendono dal tipo di algoritmo utilizzato. Tra i vari ricordiamo: Free list, paging e algoritmo buddy.

## 2.5 Buffer overflow: definizione

Il buffer overflow è una vulnerabilità di sicurezza che può affliggere un programma software. Tale vulnerabilità consiste nel fatto che tale programma non controlla in anticipo la lunghezza dei dati in arrivo, ma si limita a scriverli in un buffer di lunghezza prestabilita, confidando che l'utente non immetta più dati di quanti esso possa contenerne. Ciò accade molto spesso quando si utilizzano funzioni di libreria di I/O che non effettuano controlli sull'ingresso dei dati. Quando, quindi, per errore o per malizia vengono inviati più dati di quelli richiesti dal programma, accade che i dati in più vanno a sovrascrivere variabili interne al programma stesso o addirittura il proprio stack, causando errori oppure risultati imprevedibili che possono anche provocare il blocco del programma. Se effettuato intenzionalmente, un buffer overflow può essere caratterizzato all'immissione di codice malevolo che se eseguito darebbe il controllo del sistema nelle mani dell'attaccante. La figura sotto mostra una statistica sui tipi di vulnerabilità e mostra come il buffer overflow sia uno dei più sfruttati.

<b>Vulnerability Type</b>	<b>2004</b>	<b>2003</b>	<b>2002</b>	<b>2001</b>
Buffer Overflow	160 (20%)	237 (24%)	287 (22%)	316 (21%)
Access Validation Error	66 (8%)	92 (9%)	123 (9%)	126 (8%)
Exceptional Condition Error	114 (14%)	150 (15%)	117 (9%)	146 (10%)
Environment Error	6 (1%)	3 (0%)	10 (1%)	36 (2%)
Configuration Error	26 (3%)	49 (5%)	68 (5%)	74 (5%)
Race Condition	8 (1%)	17 (2%)	23 (2%)	50 (3%)
Design Error	177 (22%)	269 (27%)	408 (31%)	399 (26%)
Other	49 (6%)	20 (2%)	1 (0%)	8 (1%)

## 2.6 Stack overflow

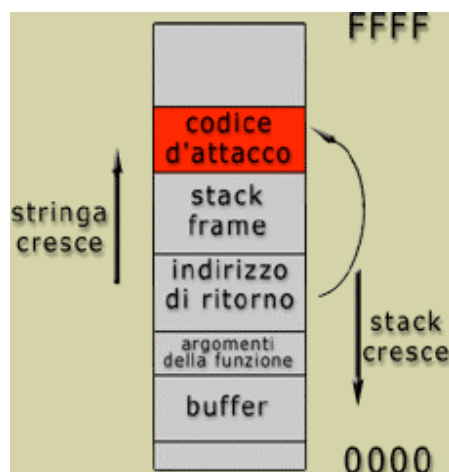
Come ho già detto in precedenza un buffer overflow si ha quando le variabili non rispettano il loro spazio di indirizzamento e perciò vanno a scrivere al di là di questo, sovrascrivendo dati precedentemente contenuti. Questo tipo di attacco è quello di gran lunga più diffuso e interessa lo stack. Anche se può essere implementato in diversi modi, il fine è sempre quello di alterare il normale flusso di esecuzione del programma secondo lo scopo dell'attaccante. Se il programma è abbastanza privilegiato e in particolare se è di tipo SUID(super user), solitamente l'attaccante cerca di attivare una shell locale in modo da poter effettuare qualunque tipo di operazione. I due passi principali su cui si basa uno stack overflow sono due:

**1)fare in modo che il codice arbitrario da far eseguire sia nell'address space del programma;**

**2)fare in modo che il programma salti a questo codice e lo esegua.**

### 2.6.1 Inserire codice arbitrario nel programma

Vi sono due metodi per inserire del codice arbitrario all'interno di un programma. Il primo metodo consiste nell'inserire il codice manualmente(code injection), ad esempio il programma richiede una stringa in input che verrà inserita dall'attaccante in modo da far eseguire delle istruzioni alla CPU. Essendo che la stringa verrà messa in un buffer, lo scopo di base è quello di provocare l'overflow di questo buffer che magari non ha controlli sui confini o li ha scarsi. Il secondo metodo prescinde dal fatto che il codice si trova già lì, quindi bisogna solamente parametrizzarlo. Ad esempio se si ha in UNIX il codice "exec(arg)", dove arg è un puntatore ad una stringa, si deve fare in modo che lo si faccia puntare a una stringa del tipo /bin/sh per avere una shell in locale.



## 2.6.2 Shellcode

Lo shellcode è una porzione di codice macchina scritto in esadecimale che traduce delle precise istruzioni da fare eseguire alla CPU. Nel caso di un buffer overflow viene utilizzato assieme ad un exploit per far eseguire una shell in locale o in remoto, solitamente una shell UNIX(/bin/sh). La stesura di un buon shellcode non è facile poiché le sue istruzioni devono essere precise e funzionanti fino alla fine, per il semplice fatto che anche un piccolo errore di programmazione potrebbe causare il crash dell'applicazione attaccata. Inoltre la particolarità di uno shellcode è quella di non essere assolutamente portabile, poiché dipende totalmente dall'architettura del sistema e, quindi, dal suo codice macchina. Sotto riporto un esempio di shellcode che esegue una shell UNIX.

```
char main[] =  
"\xeb\x19\x5e\xb0\x46\x31\xdb\x31"  
"\xc9\xcd\x80\x31\xc0\xb0\x0b\x89"  
"\xf3\x31\xff\x57\x56\x89\xe1\x31"  
"\xd2\xcd\x80\xe8\xe2\xff\xff\xff"  
"/bin/sh";
```

## 2.6.3 Far saltare il programma al codice d'attacco

Per mettere in pratica ciò si può effettuare:

1)**Activation records:** si tratta della tipologia più diffusa, nota con la frase "Smashing the stack". Si utilizza all'interno di una funzione, e consiste nel provocare un buffer overflow cercando di sovrascrivere il EIP, cioè l'indirizzo di ritorno della funzione. Ricordiamo che se questo fosse sovrascritto per sbaglio si avrebbe solamente un errore di Segmentation fault, ma se venisse sovrascritto con l'indirizzo di una procedura di attacco, essa verrebbe eseguita.

2)**Puntatori a funzioni:** si tratta di trovare un buffer vicino ad un puntatore a funzione, provocarne un overflow e corrompere, quindi, anche il puntatore stesso in modo da farlo puntare alla porzione di codice malevolo. Questo tipo di BOF può riguardare anche lo heap e altre regioni di memoria.

3)**Longjmp buffers:** sfrutta un meccanismo presente in C il quale dà la possibilità di salvare lo stato(checkpoint) di un buffer mediante il comando longjmp(buffer). Come nel caso dei puntatori a funzioni, se abbiamo un buffer vicino di cui fare l'overflow potremmo corrompere l'area di memoria dove

viene salvato lo stato del buffer e sovrascriverlo con l'indirizzo del codice d'attacco.

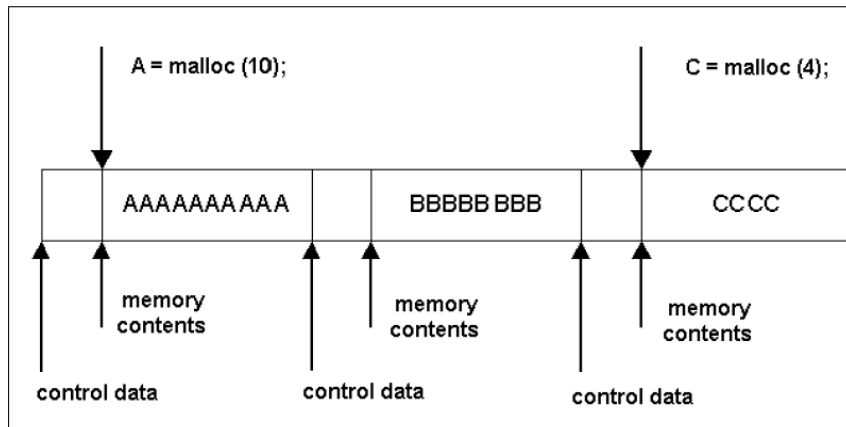
## 2.6.4 Combinare le due fasi precedenti

Per far sì che l'attacco abbia successo le due fasi precedenti devono essere combinate, anche se non per forza devono essere staccate l'una dall'altra. Si può, infatti, avere l'inserimento del codice e la sua esecuzione in un'unica fase. Pensiamo al fatto che un buffer si trovi vicino al EIP, se si inserisce una stringa opportuna che provoca un overflow sovrascrivendo il EIP, si può eseguire codice malevolo tramite activation records. Può anche capitare di avere più fasi; può, infatti, capitare che il buffer da attaccare non può contenere il codice d'attacco e quindi si deve fare una code injection in un altro buffer e poi utilizzare il primo per realizzare l'activation records.

## 2.7 Heap overflow

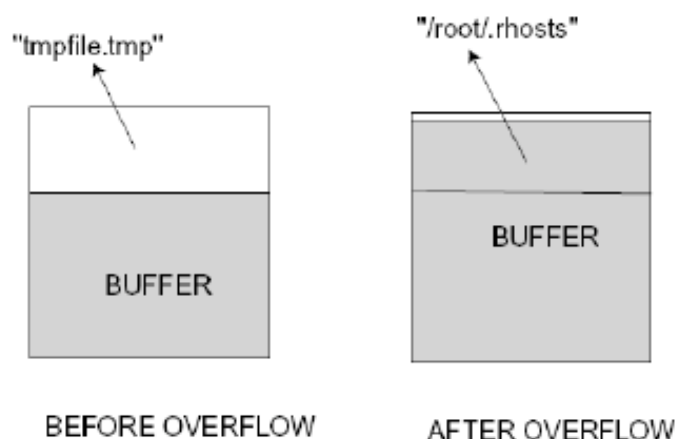
I BOF di heap prendono questo nome poiché prendono il nome dell'area di memoria che vanno a colpire, che è proprio quella dello heap, che contiene le variabili allocate dinamicamente. Questo tipo di memoria è diverso dallo stack, poiché lo spazio di memoria occupato verrà liberato solo se esplicitamente richiesto; questo comporta il fatto che un heap overflow può essere scoperto anche molto tempo dopo l'avvenuto attacco. In questo tipo di attacco non c'è il concetto di EIP ma può essere effettuato secondo diverse tecniche. Rispetto allo stack overflow, è usato più raramente poiché la sua riuscita è molto più difficile, tuttavia può essere molto pericoloso. Tra i diversi tipi di attacchi si possono avere:

**Attacchi basati su malloc() e funzioni simili:** questo tipo di funzioni servono ad allocare dinamicamente le variabili, ad esempio malloc() del C, HeapAlloc() di windows oppure new() del C++. Il buffer overflow può essere effettuato poiché queste funzioni allocano, solitamente, in aree contigue e non effettuano controlli, quindi è possibile andare a sovrascrivere dati e variabili vicine.



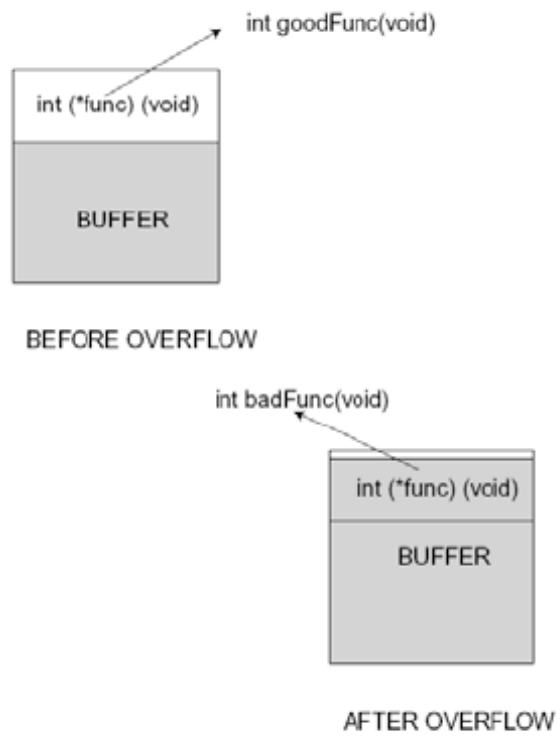
La stessa cosa può accadere con l'area di memoria BSS, che contiene le variabili non inizializzate. Anche in questo caso è possibile, quando si inizializzano questi dati, inserire più dati del previsto per cercare di sovrascrivere dati contigui. Vi sono diverse tecniche di attacco basate su questi schemi e solitamente sono architecture dependent. Ad esempio uno dei più noti è la vulnerabilità di malloc() su UNIX ed in particolare sulla macro unlink() della funzione free(), che possiede alcuni bit che posso essere exploitati. Ma qual è la logica di questo tipo di BOF? L'intento è quello di causare l'overflow di un buffer A in modo da andare a scrivere il programma d'attacco nel buffer B; quando il programma andrà ad utilizzare i dati contenuti in B, attiverà il programma malevolo.

**Attacchi basati sulla sovrascrittura dei puntatori:** questo tipo di attacco tenta di effettuare un overflow di un buffer adiacente ad un puntatore, in modo da corrompere quest'ultimo e farlo puntare ad un'altra locazione di memoria. Questo tipo di attacco è estremamente portabile e può interessare l'area BSS.



**Attacchi basati su puntatori a funzioni:** come nel caso dei BOF basati su stack, anche qui è possibile sfruttare i puntatori a funzione. In questo caso poiché i puntatori possono trovarsi sia sullo stack che sullo heap (ma anche sulla BSS), si

cercherà di effettuare un overflow di un buffer adiacente ad un puntatore, in modo da farlo puntare ad una locazione contenente il codice d'attacco. Vedi figura sotto.



## 3. Difesa da buffer overflow

### Contenuto:

- Tipi di difesa

I metodi più utilizzati per prevenire attacchi di buffer overflow possono essere raggruppati in quattro categorie principali:

- 1) **Scrivere codice sicuro;**
- 2) **Rendere l'area di memoria delle variabili non eseguibile;**
- 3) **Approccio diretto orientato al compilatore;**
- 4) **Approccio indiretto orientato al compilatore.**

### 3.1 Scrivere codice sicuro

Scrivere codice sicuro probabilmente sarebbe il metodo migliore e più efficace contro gli attacchi di buffer overflow, purtroppo è una soluzione difficile da realizzare e molto costosa. Infatti, linguaggi come il C mettono a disposizione costrutti e tipi creati maggiormente per la loro efficienza e non per la loro affidabilità, per questo motivo per poter prevenire attacchi di questo tipo ci vorrebbe una loro manipolazione che inciderebbe nelle prestazioni dei programmi. L'approccio più semplice sarebbe quello di non utilizzare tutte quelle funzioni di libreria che possiedono falle di sicurezza accertate e sostituirle con altre più sicure. In generale per una programmazione sicura si dovrebbero rispettare le seguenti regole principali:

**Principio del minor privilegio possibile.** Un programma dovrebbe possedere solamente i privilegi indispensabili per poter effettuare le proprie operazioni. Deve, appunto, avere il minore privilegio possibile e richiedere qualche privilegio in più solamente quando è necessario alla sua esecuzione. In questo modo si limitano i danni in caso di un eventuale exploit, poiché il programma corrotto non possiede i diritti necessari per poter effettuare determinate azioni dannose.

**Scrivere codice semplice.** Scrivere codice semplice è una pratica che dovrebbe essere utilizzata a prescindere da problemi di sicurezza, però risulta utile per la limitazione di falle soprattutto nei programmi di grossa dimensione. Inoltre, scrivere codice semplice aiuta nella sua manutenzione.

**Non fidarsi di nessuno.** Ogni qualvolta si riceve un dato in input prima di utilizzarlo lo si deve analizzare perché potrebbe essere potenzialmente pericoloso.

In ogni caso le vulnerabilità dovute al buffer overflow può, in molti casi, essere di difficile individuazione. Anche tramite l'utilizzo di software di monitoraggio

l'unica cosa possibile è limitare le falle di sicurezza e non eliminarle completamente.

### 3.2 Buffer non eseguibili

L'approccio generale è quello di limitare l'esecuzione dei segmenti di memoria del programma vittima in modo da prevenire l'esecuzione di codice malevolo. Questo approccio non è del tutto utilizzabile, soprattutto dai sistemi operativi più recenti che utilizzano molto l'inserimento di codice dinamico per aumentare le prestazioni dei programmi. Questo schema, infatti, non è utilizzabile in molti programmi dove il fattore prestazioni risulta critico. E' possibile rendere lo stack non eseguibile dato che la maggior parte dei sistemi operativi non ha codice eseguibile nello stack e quindi non vi sono problemi di incompatibilità. Però vi è un'eccezione: Linux. Quest'ultimo si serve di codice eseguibile nello stack per implementare i segnali. Questo metodo risulta efficiente per prevenire iniezioni di codice, ma poco utile per altri tipi di attacco.

### 3.3 Controllo della dimensione degli array

Questo approccio elimina completamente il rischio di un buffer overflow, poiché consiste nell'impossibilità di scrivere dati di dimensione maggiore del buffer atto a contenerli. L'approccio diretto è quello di testare tutti i riferimenti degli array, anche, attraverso tecniche di ottimizzazione.

### 3.4 Controllo d'integrità dei puntatori

L'intento di questo approccio è quello di evitare l'utilizzo di un buffer che è stato precedentemente sovrascritto. Questo metodo è abbastanza utile, però, non risolve il problema di attacchi ad aree di memoria che non contengano puntatori. Questa tecnica presenta, comunque, vantaggi di tipo prestazionale e di implementazione.

## 4. Un esempio di heap overflow

Analizziamo il seguente codice C per identificare le sue sezioni critiche ideali ad un attacco di heap overflow:

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

int main(int argc, char *argv[]) {
    FILE *fd;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    // copia i dati nella memoria dello heap
    strcpy(outputfile, "/tmp/note");
    strcpy(userinput, argv[1]);

    // stampa alcuni messaggi di Debug
    printf("--DEBUG--\n");
    printf("* userinput @ %p: %s\n", userinput, userinput);
    printf("* outputfile @ %p: %s\n", outputfile, outputfile);
    printf("* distanza: %d bytes\n", outputfile - userinput);
    printf("-----\n\n");

    // scriviamo i dati
    printf("Scrittura di %s alla fine di %s", userinput, outputfile)
    fd = fopen(outputfile, "a");

    fprintf(fd, "%s\n", userinput);
    fclose(fd);

    return 0;
}
```

Il problema fondamentale risiede nelle due istruzioni:

```
char *userinput = malloc(20);
char *outputfile = malloc(20);
```

Infatti la variabile \*output file potrebbe essere sovrascritta dalla variabile \*userinput. Le altre due istruzioni critiche sono:

```
strcpy(outputfile, "/tmp/note");
strcpy(userinput, argv[1]);
```

Se il programma è di proprietà della root, la sicurezza del sistema potrebbe venir compromessa. Compiliamo il programma, diamo i privilegi di root al nostro programma ed eseguiamolo:

```
Terminale — bash — bash — 80x24
Artista:Desktop Simone$ gcc -o heap heap.c
Artista:Desktop Simone$ sudo su
sh-3.2# chown root heap
sh-3.2# chmod u+s heap
sh-3.2# exit
exit
Artista:Desktop Simone$ ./heap test
--DEBUG--
* userinput @ 0x100120: test
* outputfile @ 0x100140: /tmp/note
* distanza: 32 bytes
-----

Scrittura di "test" alla fine di /tmp/note...
Artista:Desktop Simone$ █
```

L'intento del programma è quello di scrivere all'interno di un file il contenuto di una stringa passata da riga di comando. Come possiamo notare l'indirizzo della variabile `userinput` precede quello della variabile `output file`, essendo che si trovano su locazioni contigue. Questo ci permette di scrivere nella seconda variabile dati arbitrari tramite un buffer overflow della prima. Se ad esempio immettiamo da riga di comando i seguenti dati:

```
Terminale — bash — bash — 80x24
Artista:Desktop Simone$ ./heap 1234567891234567891234567891234
--DEBUG--
* userinput @ 0x100120: 1234567891234567891234567891234
* outputfile @ 0x100140: /tmp/note
* distanza: 32 bytes
-----

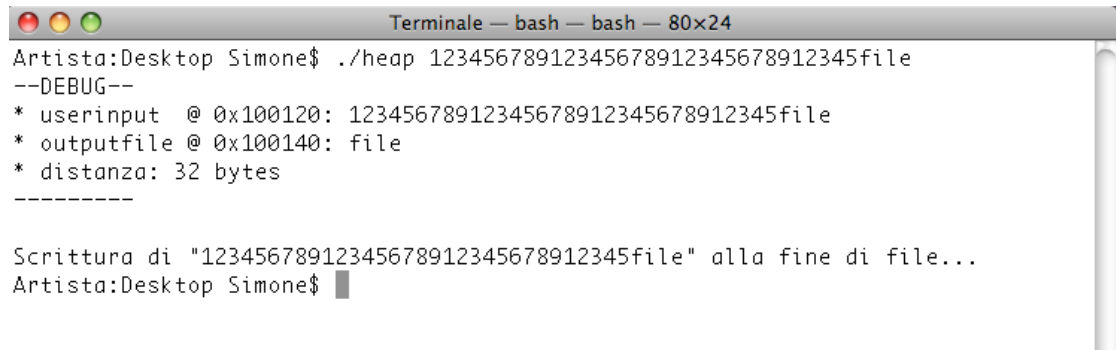
Scrittura di "1234567891234567891234567891234" alla fine di /tmp/note...
Artista:Desktop Simone$ █
```

Eseguendo il programma immettendo sulla riga di comando 31 byte, il programma funziona correttamente poiché l'argomento rientra esattamente all'interno del buffer di `userinput` e quindi il suo contenuto viene stampato sul file. Ma se immettiamo anche un solo byte in più si verifica un buffer overflow che provoca il fallimento della scrittura sul file. Vediamo:

```
Terminale — bash — bash — 80x24
Artista:Desktop Simone$ ./heap 12345678912345678912345678912345
--DEBUG--
* userinput @ 0x100120: 12345678912345678912345678912345
* outputfile @ 0x100140:
* distanza: 32 bytes
-----

Scrittura di "12345678912345678912345678912345" alla fine di ...
Errore nell'apertura di
Artista:Desktop Simone$ █
```

Come possiamo notare l'aggiunta del byte causa un buffer overflow e la locazione output file viene parzialmente sovrascritta facendo fallire la scrittura sul file. Se provochiamo l'overflow inserendo dati validi possiamo ottenere la scrittura su un file arbitrario:



```
Terminale — bash — bash — 80x24
Artista:Desktop Simone$ ./heap 12345678912345678912345678912345file
--DEBUG--
* userinput @ 0x100120: 12345678912345678912345678912345file
* outputfile @ 0x100140: file
* distanza: 32 bytes
-----
Scrittura di "12345678912345678912345678912345file" alla fine di file...
Artista:Desktop Simone$
```

In questo caso siamo riusciti, provocando un buffer overflow, a sovrascrivere la memoria occupata da outputfile e farlo puntare al nome di un file arbitrario. Allo stesso modo, avendo i privilegi di root possiamo andare a scrivere su un file di sistema come /etc/passwd, riuscendo anche ad aggiungere un utente di sistema.

## 5. Conclusioni

Come ho avuto modo di spiegare, il problema della sicurezza è divenuto ormai un problema molto comune in una società nella quale i sistemi informatici hanno messo piede ormai da diversi anni e che ogni anno assumono maggiore importanza nella nostra vita. Per questo motivo si deve porre molta attenzione nella progettazione di programmi cercando di evitare gli errori più comuni che incoraggiano attacchi da parte di persone esperte e non. In questo contesto, l'attacco di buffer overflow si pone come una strategia molto potente e pericolosa, ma anche difficile da attuare in molti casi. L'unica buona difesa da questi tipi di attacchi è il rispetto delle giuste regole di programmazione che pur limitando i rischi non rendono i sistemi immuni da questi attacchi.

## 6. Riferimenti

- <http://www.wikipedia.org>
- <http://mزالug.tuxfamily.org>
- <http://www.siforge.org>
- <http://www.dia.unisa.it>
- <http://www.intilinux.com>
- Sistemi operativi concetti ed esempi, Silberschatz-Galvin-Gagne
- Sicurezza in informatica, Charles P.Pfleeger-Shari Lawrence Pfleeger