

# Capitolo 3

## Heap visualization tool

In questo capitolo sarà presentato il tool di heap visualization realizzato per il presente lavoro di tesi.

```
toto@xps: ~/pin/source/tools/MyPinTool

:::free(0x556f6b1728b0):::

[-----Info-----]

MainArena:    0x7f2c9db04b80
HeapBase:     0x556f6b15e000
TopChunk:     0x556f6b176b60
FastBins:     0x7f2c9db04b90
Tcache:       0x556f6b15e090

[-----Chunks-----]

ADDRESS      SIZE      STATE      FD      BK
0x556f6b15e000 0x290    Used      -      -
0x556f6b15e290 0x11c10  Used      -      -
0x556f6b16fea0 0x410    Used      -      -
0x556f6b1702b0 0x7e0    Used      -      -
0x556f6b170a90 0x3f0    Tcache    0000000000000000 -
0x556f6b170e80 0x5f0    Used      -      -
0x556f6b171470 0x2260   Free      0x556f6b173810 0x7f2c9db04be0
0x556f6b1736d0 0x20     Tcache    0000000000000000 -
0x556f6b1736f0 0x20     Tcache    0x556f6b1736e0 -
0x556f6b173710 0x20     Tcache    0x556f6b173700 -
0x556f6b173730 0x20     Used      -      -
0x556f6b173750 0x20     Tcache    0x556f6b173720 -
0x556f6b173770 0x20     Tcache    0x556f6b173760 -
0x556f6b173790 0x20     Tcache    0x556f6b173780 -
0x556f6b1737b0 0x20     Tcache    0x556f6b1737a0 -
0x556f6b1737d0 0x40     Tcache    0000000000000000 -
0x556f6b173810 0x2720   Free      0x7f2c9db04be0 0x556f6b171470
0x556f6b175f30 0x70     Tcache    0000000000000000 -
0x556f6b175fa0 0xbc0    Used      -      -

[-----Bins-----]

Unsorted = 0x556f6b171470 --> 0x556f6b173810

Tcache[0] = 0x556f6b1737c0 --> 0x556f6b1737a0 --> 0x556f6b173780 --> 0x556f6b173760 --> 0x556f6b173720
--> 0x556f6b173700 --> 0x556f6b1736e0
Tcache[2] = 0x556f6b1737e0
Tcache[5] = 0x556f6b175f40
Tcache[61] = 0x556f6b170aa0

toto@xps:~/pin/source/tools/MyPinTool$
```

Figura 3.1: Heap visualization tool

Tramite l'utilizzo di questo heap visualizer è possibile eseguire e strumentare un qualsiasi programma, con lo scopo di stampare a video lo stato della memoria heap dopo ogni funzione di allocazione o deallocazione di memoria. Per poter strumentare correttamente ogni programma è stato utilizzato Pin, il noto tool di dynamic binary instrumentation presentato nel capitolo precedente. Com'è possibile vedere dall'immagine 3.1, il tool realizzato si presenta con una grafica semplice, simile a pwndbg [18], e mostra a video la lista di chunks in uso o liberi, e la lista dei vari bin, siano essi unsorted, small, large, fast o tcache. All'avvio del pintool, in base all'inserimento o meno del flag "-s", sarà possibile decidere se avere direttamente una stampa totale di tutti i chunks e i bin per ogni chiamata a `malloc`, `free` ecc, o se effettuare un'analisi più "interattiva" step by step. In quest'ultimo caso, ad ogni chiamata intercettata verrà richiesto all'utente se stampare solo i chunks, solo i bin, entrambi, o andare alla prossima chiamata senza effettuare alcuna stampa.

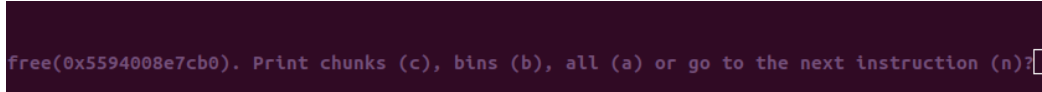


Figura 3.2: Step by step attivo

Con questa soluzione è possibile effettuare un'analisi sistematica dell'heap, senza avere necessariamente centinaia di stampe nel caso in cui si vadano ad analizzare programmi di lunghezza medio-grande. Nei paragrafi seguenti sarà analizzato il tool più nel dettaglio, focalizzando l'attenzione sul main, le routine di analisi e le routine di strumentazione.

## 3.1 Main

Per poter realizzare un pintool, bisogna implementare un file `.cpp` che deve necessariamente includere `pin.H`, in maniera tale da accedere alle API di Pin. Il main del programma si presenta nel seguente modo:

```
1 int main(int argc, char *argv[]){  
2     // Initialize pin & symbol manager
```

```
3  PIN_InitSymbols();
4  if( PIN_Init(argc,argv) ){
5      return Usage();
6  }
7  // Register Image to be called to instrument functions.
8  IMG_AddInstrumentFunction(Image, 0);
9  PIN_AddFiniFunction(Fini, 0);
10
11  PIN_StartProgram();
12
13  return 0;
14 }
```

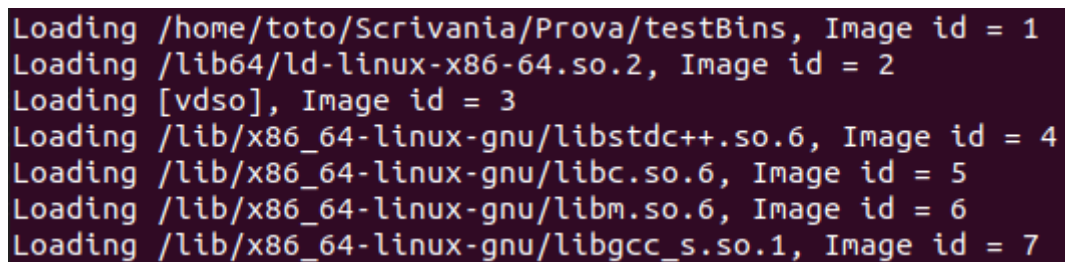
Listing 3.1: main

La prima funzione presente nel main è “PIN\_InitSymbols”, tramite la quale si permette a Pin di leggere la tabella dei simboli e dare quindi la possibilità di accedere alle funzioni per nome. Pin fornisce infatti l’accesso ai nomi delle funzioni utilizzando l’oggetto simbolo (SYM). È importante sottolineare che PIN\_InitSymbols deve essere necessariamente chiamato prima di PIN\_Init(). Quest’ultima è la funzione che inizializza Pin e restituisce true nel caso in cui viene sollevato qualche errore durante l’inizializzazione. Proprio per questo motivo è opportuno inserire PIN\_Init all’interno di un if, in maniera tale da avvisare opportunamente l’utente in caso di errore. La funzione successiva è IMG\_AddInstrumentFunction(Image, 0), utilizzata per registrare la routine di strumentazione denominata Image e catturare il caricamento di ogni immagine. Si noti che il secondo argomento di questa funzione, in questo caso settato a 0, può essere usato per passare qualsiasi informazione aggiuntiva alla funzione di strumentazione. Ovviamente, come specificato nel capitolo precedente, è possibile inserire altre routine di strumentazione con differenti livelli di granularità. Per concludere, viene inserita la funzione PIN\_AddFiniFunction(Fini, 0) che permette di chiamare una funzione (in questo caso Fini) subito prima della exit del programma. Anche in questo caso, come per IMG\_AddInstrumentFunction, il secondo parametro

serve per passare eventuali informazioni aggiuntive. Ultima funzione presente all'interno del main è sempre `PIN_StartProgram()`, che avvia l'esecuzione dell'applicazione. Importante sottolineare che `PIN_Init()` deve essere chiamata sempre prima di `PIN_StartProgram()`.

## 3.2 Routine di strumentazione

Le routine di strumentazione determinano in quale punto inserire le chiamate alle routine di analisi. In questo caso, come già anticipato, è presente una routine di strumentazione di immagine, denominata `Image`. È importante sottolineare che Pin analizza il programma strumentato dalla prima istruzione all'ultima, comprese quindi tutte le istruzioni eseguite nelle varie librerie condivise. Proprio per questo motivo, le immagini trovate da Pin sono molteplici, come per esempio



```
Loading /home/toto/Scrivania/Prova/testBins, Image id = 1
Loading /lib64/ld-linux-x86-64.so.2, Image id = 2
Loading [vdso], Image id = 3
Loading /lib/x86_64-linux-gnu/libstdc++.so.6, Image id = 4
Loading /lib/x86_64-linux-gnu/libc.so.6, Image id = 5
Loading /lib/x86_64-linux-gnu/libm.so.6, Image id = 6
Loading /lib/x86_64-linux-gnu/libgcc_s.so.1, Image id = 7
```

Figura 3.3: Esempio di immagini catturate da Pin

In questo caso si è però scelto di strumentare solo la `libc.so.6`, specificandolo con un `if` proprio all'inizio della routine di strumentazione `Image`.

```
1 VOID Image(IMG img, VOID *v){
2     if(IMG_Name(img) == "/lib/x86_64-linux-gnu/libc.so.6"){
```

Listing 3.2: Strumentazione `libc.so.6`

Tale scelta è dettata dal fatto che si vogliono mostrare all'utente solo le istruzioni di `malloc` e `free` del programma, e non quelle chiamate per esempio durante l'esecuzione del dynamic linker `ld-linux-x86-64.so.2`. Arrivati a questo punto, per poter catturare correttamente le funzioni `malloc`, `free`, `calloc` e `realloc`, vengono in aiuto le routine `RTN`.

```
1 RTN mallocRtn = RTN_FindByName(img, MALLOC);
2 if (RTN_Valid(mallocRtn)){
3     RTN_Open(mallocRtn);
4     RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)
5         ↪ BeforeMalloc, IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
6     RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)AfterMalloc,
7         ↪ IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
8     RTN_Close(mallocRtn);
9 }
```

Listing 3.3: Routine mallocRtn

Più nello specifico, com'è possibile vedere dal codice, viene utilizzata la funzione `RTN_FindByName()`, che prende in input un'immagine e una stringa e restituisce un oggetto RTN che rappresenta la routine desiderata. In questo caso, infatti, viene cercato all'interno dell'immagine `img` la routine `malloc`, e se questa ricerca va a buon fine (lo si capisce effettuando il check tramite l'utilizzo di `RTN_Valid()`), allora sarà possibile strumentare a proprio piacimento la `malloc`. Per inserire le routine di analisi nei punti desiderati, si utilizza la funzione `RTN_InsertCall()`. Quest'ultima prende in input i seguenti parametri:

1. La routine che si vuole strumentare;
2. Un enum `IPOINT` che determina in quale punto dell'oggetto strumentato viene inserita la funzione di analisi. Più nello specifico viene utilizzato `IPOINT_BEFORE` per inserire la funzione subito prima dell'esecuzione della routine, mentre viene utilizzato `IPOINT_AFTER` per inserirla subito dopo, prima della return della routine. Nella documentazione, Pin specifica che non è sempre possibile trovare correttamente l'istruzione di ritorno;
3. La routine di analisi che deve essere "iniettata";
4. L'insieme degli argomenti necessari alla routine di analisi. La lista deve necessariamente terminare con `IARG_END`.

Nel caso della `malloc`, è stato necessario inserire gli argomenti `IARG_FUNCARG_ENTRYPOINT_VALUE` e `IARG_FUNCRET_EXITPOINT_VALUE` rispettivamente nelle chiamate `BeforeMalloc` e `AfterMalloc`. `IARG_FUNCARG_ENTRYPOINT_VALUE` permette di passare alla routine di analisi un parametro presente nella routine strumentata, quindi in questo caso permette di passare a `BeforeMalloc` la size data in input alla `malloc`. Al contrario, `IARG_FUNCRET_EXITPOINT_VALUE` permette di passare il valore di ritorno alla routine di analisi, passando quindi l'indirizzo di ritorno della `malloc` alla funzione `AfterMalloc`. Il funzionamento delle due routine di analisi sarà spiegato più nello specifico nel paragrafo successivo. Lo stesso identico meccanismo utilizzato per la `malloc` è stato replicato anche per la `calloc` e la `realloc`. Per quanto riguarda la `free` invece, occorre fare un discorso a parte. In linea generale, l'approccio utilizzato per la `free` è lo stesso di quello impiegato per la `malloc`, ma nel caso della `free` ci sono stati problemi per quanto concerne l'inserimento della routine di analisi subito dopo la sua esecuzione. Questo probabilmente è dovuto al fatto che non sempre Pin riesce a trovare l'istruzione di ritorno e di conseguenza, non riesce a strumentare correttamente la routine. Proprio per questo motivo, non avendo un punto d'aggancio subito dopo l'esecuzione della `free`, non è possibile stampare le modifiche apportate da quest'ultima all'interno dell'heap (in particolare in questo caso non sarà possibile stampare i vari bin). Per ovviare a questo problema, si è deciso quindi di stampare questi cambiamenti dell'heap non subito dopo la `free` in questione, ma subito prima dell'esecuzione della prossima funzione, sia essa una `malloc`, `realloc`, `calloc` o un'altra `free`. Tale comportamento sarà spiegato più dettagliatamente nel paragrafo successivo.

### 3.3 Routine di analisi

Le routine di analisi del tool realizzato vengono inserite o subito prima l'esecuzione della routine strumentata o subito dopo. Più nello specifico, le routine di analisi denominate `BeforeMalloc()`, `BeforeCalloc()`, `BeforeRealloc()` e `BeforeFree()` vengono inserite rispettivamente subito prima dell'esecuzione

di `malloc`, `calloc`, `realloc` e `free`, e sostanzialmente hanno il compito di stampare le modifiche apportate dalla `free` precedente (se esiste), in maniera tale da risolvere il problema spiegato nel paragrafo precedente. Sono state inoltre realizzate altre tre routine di analisi, denominate `AfterMalloc()`, `AfterCalloc` e `AfterRealloc` che invece vengono inserite rispettivamente subito prima la return di `malloc`, `calloc` e `realloc`, stampando a video le modifiche apportate all'heap. Verrà ora analizzato più nel dettaglio il funzionamento di queste routine di analisi, focalizzando l'attenzione solo su `BeforeMalloc()` e `AfterMalloc()` per una trattazione più semplice ma comunque esaustiva, dato che le restanti routine si comportano in maniera molto simile.

### 3.3.1 BeforeMalloc()

```

1 VOID BeforeMalloc(ADDRINT size){
2     if(printFree){
3         string input;
4         if(StepByStep.Value()) {
5             cout << BOLDMAGENTA << "\nfree("<< lastFree <<").";
6             cout << " Print chunks (c), bins (b), all (a) or go to
↪ the next instruction (n)?" << RESET;
7             std::cin >> input;
8         }
9         cout << MALL << "\n::::::::::::::::::::::::::::free("<<
↪ lastFree <<")::::::::::::::::::::::::::::\n"<< RESET;
10        if(!StepByStep.Value() || input == "c" || input == "a"){
11            printInfo();
12            printChunks();
13        }
14        if(!StepByStep.Value() || input == "b" || input == "a"){
15            printBins();
16        }
17        printFree = false;
18    }
19    lastMall = size;

```

20 }

Listing 3.4: BeforeMalloc()

Com'è possibile vedere dal codice, `BeforeMalloc()` si basa principalmente sulla stampa delle modifiche apportate da un eventuale `free` precedente. Il booleano "`printFree`" verrà settato a `true` all'interno di `BeforeFree()`, in maniera tale da far capire alla routine di analisi successiva che vi è una stampa delle modifiche dell'heap in sospeso. `StepByStep.Value()` restituisce un booleano e sarà analizzato nel dettaglio più avanti. In sostanza, il suo valore sarà `true` nel caso in cui si vuole effettuare l'analisi del codice step by step, mentre sarà `false` nel caso in cui si vuole avere direttamente la stampa completa di chunks e bin per ogni istruzione strumentata. Nel primo caso, all'utente sarà comunicato qual è l'istruzione corrente (`free`, `malloc` ecc) e gli verrà chiesto se stampare solo chunk, bin, entrambi, o se andare alla prossima istruzione senza stampare nulla (Figura 3.2). Tralasciando ora le varie stampe, i due metodi principali sono `printChunks()` e `printFree()`, che sono i principali metodi utilizzati in ognuna delle routine di analisi.

```

1 VOID printChunks(){
2     cout << CHUNKS << "[-----]
    ↪ Chunks -----]\n\n" << RESET
    ↪ ;
3     ADDRINT* current = heapBase;
4     cout << "ADDRESS \t\t" << "SIZE \t\t" << " STATE \t\t" <<
    ↪ RESET << "FD \t\t\t" << "BK" << "\n";
5     while((ADDRINT)current != topChunk){
6         ADDRINT size = sizeAMP(current[1]);
7         ADDRINT* next = (ADDRINT*) ((ADDRINT)current + size);
8         sizeAMP(next[1]);
9         bool tc = inTcache(current);
10        bool fs = inFast(current);
11        if(prev){
12            if(tc){
13                cout << BLUE << hex << setw(14) << (ADDRINT)current <<
    ↪ "\t\t" << YELLOW << size << "\t\t" << MAGENTA << "Tcache"
```



```

14     }
15     else if(fs){
16         cout << BLUE << hex << setw(14) << (ADDRINT)current <<
17         ↪ "\t\t" << YELLOW << size << "\t\t" << MAGENTA << "Fast\t\t\
18         ↪ t"<< RESET << hex << setw(14) << current[2] << "\t\t" <<
19         ↪ "-" << "\n";
20     }else{
21         cout << BLUE << hex << setw(14) << (ADDRINT)current <<
22         ↪ "\t\t" << YELLOW << size << "\t\t" << RED << "Used" <<
23         ↪ RESET << "\t\t - \t\t" << "-" << "\n";
24     }
25     }else{
26         cout << BLUE << hex << setw(14) << (ADDRINT)current << "\
27         ↪ t\t" << YELLOW << size << "\t\t" << GREEN << " Free\t\t"
28         ↪ << RESET << hex << setw(14) << current[2] << "\t\t"<<
29         ↪ current[3] << "\n";
30     }
31     current = next;
32 }
33 cout<<"\n\n";
34 }

```

Listing 3.5: printChunks()

Per la stampa dei chunks, si è deciso di partire dall'inizio dell'heap (heap-Base, spiegato in seguito) e proseguire stampando ogni chunk fino all'arrivo del topChunk. Più nel dettaglio, ad ogni chunk (partendo dall'heapBase) si è aggiunta la size del chunk stesso, ottenendo così l'indirizzo del chunk successivo. Successivamente, in base al capo prev del next chunk (ovvero al bit P del campo size) si capisce se il chunk corrente è in uso o meno, differenziando così le varie stampe. Così facendo, sarà possibile stampare il valore del chunk (`current`), la sua grandezza (`size`) e i valori di fd (`current[2]`) e bk (`current[3]`). Bisogna inoltre ricordare che i chunk free presenti all'interno della tcache e fast bin lasciano il campo P del next chunk settato ad 1, quindi

per riuscire a distinguerli dai reali chunk in uso vengono utilizzati i metodi `inTcache()` e `inFast()`, che in sostanza iterano rispettivamente all'interno della tcache e dei fastbin per vedere se è presente il chunk corrente. Ovviamente, come specificato nel primo capitolo, i tre bit più significativi del campo `size` vengono utilizzati come flag di stato, ed occorre quindi "ripulirli" prima di effettuare la somma tra il valore della `size` e quello del chunk. Per fare ciò, è stato utilizzato il metodo denominato `sizeAMP()`.

```
1 ADDRINT sizeAMP(ADDRINT size){
2     ADDRINT ad= size;
3     stringstream s1;
4     s1 << hex << ad;
5     unsigned n;
6     s1 >> n;
7     bitset<64> b(n);
8
9     prev = b[0];
10    memoryArea = b[1];
11    mainA = b[2];
12    b[0] = 0; b[1] = 0; b[2] = 0;
13
14    std::stringstream s2;
15    s2<<b.to_ulong();
16    ADDRINT result;
17    s2 >> result;
18    return result;
19 }
```

Listing 3.6: `sizeAMP()`

In questo caso, vengono estratti gli ultimi tre bit dal campo `size` passato in input, vengono opportunamente settate le tre variabili (`prev`, `memoryArea`, `mainA`) e infine si azzerano questi tre bit e si restituisce il valore corretto della `size`. Verrà ora analizzato il metodo `printBins()`, che si occupa della stampa dei vari bin presenti. Per semplicità di visualizzazione, non sarà mostrato il codice del metodo per intero ma solo i vari pezzi relativi ad alcune tipologie di bin. I primi bin analizzati sono gli `unsorted`.

```

1 if(arrayBins[1] != (ADDRINT)& (main_arena[12])){
2     ADDRINT* first = (ADDRINT*) arrayBins[1];
3     ADDRINT* last = (ADDRINT*) arrayBins[2];
4     if(first == last){ //there is only 1 unsorted
5         cout << "Unsorted = " << hex<< setw(14) << first << "\n";
6     }else{ //there are more than 1 unsorted
7         ADDRINT* next = (ADDRINT*) first[2];
8         cout<< "Unsorted = "<< hex<< setw(14)<< (ADDRINT) first;
9         while(next != last){
10             cout << " --> " << hex << setw(14) << (ADDRINT) next;
11             next = (ADDRINT*) next[2];
12         }
13         cout<< " --> " << hex<< setw(14)<< (ADDRINT) last << "\n";
14     }
15     cout<<"\n";
16 }

```

Listing 3.7: printBins() - Unsorted bins

Il puntatore denominato `arrayBins` fa riferimento all'array (descritto nel primo capitolo) che memorizza unsorted, small e large bin. Se `arrayBins[1]` è uguale all'indirizzo di `main_arena[12]` (ovvero l'indirizzo in cui si trova il valore del top chunk) allora significa che non ci sono unsorted bin disponibili. Altrimenti, in `arrayBins[1]` è presente l'indirizzo del primo unsorted bin (`first`), mentre in `arrayBins[2]` è presente l'indirizzo dell'ultimo unsorted bin (`last`). Se questi due valori corrispondono, significa che esiste un solo unsorted bin, che verrà stampato, altrimenti si parte dal primo unsorted e di volta in volta si prende il valore di `fd` (nel primo caso corrisponderà a `first[2]`) e si prosegue fin quando `fd` avrà il valore dell'ultimo unsorted bin, quindi di `last`. Così facendo, si partirà dal primo unsorted e si stamperanno tutti i bin fino all'ultimo. La stampa di small e large bin avviene nello stesso modo degli unsorted, con la sola eccezione di un `for` esterno che itera per ogni dimensione disponibile di questi bin. Si ricorda infatti che sono presenti ben 62 small bin e 63 large bin. Per quanto riguarda la stampa degli small bin quindi, si ha un `for` che scorre l'`arrayBins` dalla posizione 3 alla posizione 126, mentre per

quanto riguarda i large bin si parte dalla posizione 127 dell'arrayBins, fino ad arrivare alla 252. Così facendo, in maniera analoga a quanto accade per gli unsorted, vengono effettuate anche le stampe dei large e small bin. Un discorso differente va invece fatto per quanto riguarda i fast bin e tcache.

```

1 for(int i=0, j=0; i<=9; i++, j++){
2     if(fastBins[i] != 0){
3         binCheck = true;
4         cout << "Fast[" << std::dec << j << "] = " << hex << setw
↪ (14) << fastBins[i] ;
5         ADDRINT* current = (ADDRINT*) fastBins[i];
6         while(current[2] != 0){
7             cout << " --> " << hex << setw(14) << current[2];
8             current = (ADDRINT*) current[2];
9         }
10    }
11 }

```

Listing 3.8: printBins() - Fast bins

In entrambi i casi, infatti, sono presenti linked list singole, quindi per ogni dimensione è presente solo il primo chunk della lista. Nel campo fd del chunk sarà quindi presente l'indirizzo del prossimo chunk se esiste, altrimenti il campo fd sarà pari a 0. Il codice relativo alla tcache è molto simile a quello dei fast bin, e differisce solo per i valori del for (in quanto si hanno 64 tcache bin e 10 fast bin) e per la posizione del campo fd, che nel caso della tcache non si troverà in `current[2]` ma in `current[0]`. Questo è dovuto al fatto che nella tcache si ha una linked list con collegamenti che puntano direttamente al payload e non all'header del chunk. Per concludere questa parte, è opportuno specificare anche l'origine del booleano `StepByStep.Value()` incontrato nel codice. Questo booleano si ottiene direttamente da riga di comando durante l'avvio del tool, in quanto Pin fornisce una classe denominata `Knob` e usata appunto per creare opzioni della linea di comando.

```

1 KNOB<bool> StepByStep(KNOB_MODE_WRITEONCE, "pintool",
2     "s", "0", "Step by step");

```

Listing 3.9: StepByStep

Com'è possibile vedere dal codice infatti, tramite il flag -s impostato a riga di comando viene settato il booleano `StepByStep` a `true`, altrimenti, se il flag viene omesso, il booleano viene settato di default a `false`.

### 3.3.2 AfterMalloc()

Analizzata la parte relativa a `BeforeMalloc()` e ad i metodi che utilizza, si passa ora all'analisi di `AfterMalloc()`, di cui il codice è di seguito riportato

```

1 VOID AfterMalloc(ADDRINT ret){
2     if(first){
3         getHeapBase();
4         getMainArena();
5         arrayBins = &(main_arena[13]); //main_arena + 0x68
6         fastBins = &(main_arena[2]); //main_arena + 0x10
7         tcache = &(heapBase[18]); //heapBase + 0x90
8         cout << showbase << internal << setfill('0') << "\n";
9         printInfo();
10        cout<< MALL << "\n::::::::::::::::::::::::::::malloc("<< dec
    ↪ << lastMall <<") = "<<std::hex<<ret<<"
    ↪ ::::::::::::::::::::::::::::::"<< RESET <<"\n\n";
11        printChunks();
12        first = false;
13    }else{
14        string input;
15        if(StepByStep.Value()){
16            cout<<BOLDMAGENTA<< "\nmalloc("<< dec << lastMall <<").";
17            cout << " Print chunks (c), bins (b), all (a) or go to
18                the next instruction (n)?" << RESET;
19            std::cin >> input;
20        }
21        cout<< MALL << "\n::::::::::::::::::::::::::::malloc("<< dec
    ↪ << lastMall <<") = "<<std::hex<<ret<<"
    ↪ ::::::::::::::::::::::::::::::"<< RESET <<"\n\n";
22        if(!StepByStep.Value() || input == "chunks" || input == "c"
    ↪ || input == "a"){
23            printInfo();

```

```
24     printChunks();
25 }
26 if(!StepByStep.Value() || input == "bins" || input == "b"
↪ || input == "a"){
27     printBins();
28 }
29 }
30 }
```

Listing 3.10: AfterMalloc()

Si può subito notare che la parte interna all'`else` è molto simile a quanto precedentemente descritto in `BeforeMalloc()`. Anche in questo caso infatti, vengono utilizzati i metodi `printChunks()` e `printBins()` per stampare i cambiamenti dell'heap causati questa volta non dalla `free` ma dalla `malloc`. La parte più interessante del codice però, risiede all'interno dell'`if`. Infatti, nel momento in cui una `malloc` viene effettuata per la prima volta all'interno del programma, si andranno ad estrapolare una serie di valori fondamentali per l'analisi dell'heap, come la base dell'heap, l'indirizzo del `main_arena` e anche gli indirizzi dei vari bin. Il primo metodo chiamato è infatti `getHeapBase()`, tramite il quale si vuole ottenere la base dell'heap.

```
1 VOID getHeapBase(){
2     std::ifstream infile ("/proc/self/maps",std::ifstream::in);
3     std::string line;
4     void* heapB ;
5     while (std::getline(infile, line)){
6         if(line.find("[heap]")!= std::string::npos){
7             std::string::size_type pos = line.find('-');
8             string str = line.substr(0, pos);
9             std::stringstream ss;
10            ss << std::hex<<str;
11            ss >>heapB;
12        }
13    }
14    heapBase = (ADDRINT*)heapB;
```

15 }

Listing 3.11: getHeapBase()

Com'è possibile vedere dal codice, per riuscire ad ottenere il valore desiderato viene interrogato il file `/proc/self/maps`, che contiene le regioni di memoria attualmente mappate ed i loro permessi di accesso. Una volta aperto il file, occorre scorrere ogni riga fino al raggiungimento della riga contenente la stringa “[heap]”, in corrispondenza della quale sarà presente l'indirizzo base della memoria heap. L'ottenimento di tale valore è di fondamentale importanza per la stampa dei vari chunks, così com'è stato precedentemente spiegato in merito al metodo `printChunks()`. Una volta ottenuto l'indirizzo base dell'heap, occorre calcolare quello che è il valore del `main_arena`. Per fare ciò, è stato realizzato il metodo `getMainArena()` di seguito riportato:

```

1 void getMainArena() {
2     int fd = open("/usr/lib/debug//lib/x86_64-linux-gnu/libc
    ↪ -2.31.so", O_RDONLY);
3     off_t fsize;
4     fsize = lseek(fd, 0, SEEK_END);
5     char *base = (char *) mmap(NULL, fsize, PROT_READ,
    ↪ MAP_PRIVATE, fd, 0);
6     Elf64_Ehdr *header = (Elf64_Ehdr *)base;
7     Elf64_Shdr *secs = (Elf64_Shdr*)(base+header->e_shoff);
8     for(unsigned secinx = 0; secinx < header->e_shnum; secinx++){
9         if (secs[secinx].sh_type == SHT_SYMTAB) {
10             Elf64_Sym *symtab = (Elf64_Sym *) (base+secs[secinx].
    ↪ sh_offset);
11             char *symnames = (char *) (base + secs[secs[secinx].
    ↪ sh_link].sh_offset);
12             unsigned symcount = secs[secinx].sh_size/secs[secinx].
    ↪ sh_entsize;
13             for(unsigned syminx = 0; syminx < symcount; syminx++) {
14                 if (strcmp(symnames+symtab[syminx].st_name, "main_arena
    ↪ ") == 0) {
15                     void *mainarena = ((char *)start)+symtab[syminx].
    ↪ st_value;

```

```
16         main_arena = (ADDRINT *) mainarena;
17     }
18 }
19 }
20 }
21     close(fd);
22 }
```

Listing 3.12: malloc\_chunk

Il problema di fondo è che `main_arena` è un simbolo locale e quindi non viene esportato. La soluzione consiste quindi nell'ottenere tale valore sfruttando la symbol table del file `"/usr/lib/debug//lib/x86_64-linux-gnu/libc-2.31.so"`. Più nello specifico, è stata utilizzata la section header table del file, che permette di localizzare tutte le sue sezioni, tra cui la `syntab`. La section header table in sostanza corrisponde ad un array di strutture `Elf32_Shdr` o `Elf64_Shdr`. Tramite l'utilizzo di `e_shoff` si ottiene l'offset in byte dall'inizio del file alla section header table. Fatto ciò, si itera all'interno della tabella (tramite `e_shnum` si ottiene infatti il numero di entry della section header table) fino al raggiungimento della sezione `SHT_SYMTAB`. Arrivati a questo punto è sufficiente iterare all'interno della `syntab` fino al raggiungimento del `main_arena`. Quest'ultima, essendo una struct `malloc_state`, è stata fondamentale per poter ottenere tutti gli altri indirizzi e informazioni. Infatti, com'è possibile vedere dal codice di `AfterMAlloc()`, a partire dal `main_arena` sono stati calcolati gli indirizzi dell'`arrayBins` (corrispondente a `main_arena + 0x68`), quello dei `fastBins` (`main_arena + 0x10`) e quello della `tcache` (`main_arena + 0x90`). Questi sono proprio gli indirizzi base a partire dai quali sono stati ricavati i vari chunks e bin, come precedentemente spiegato.