HACKUTT

Exploitation de binaires

Part 1

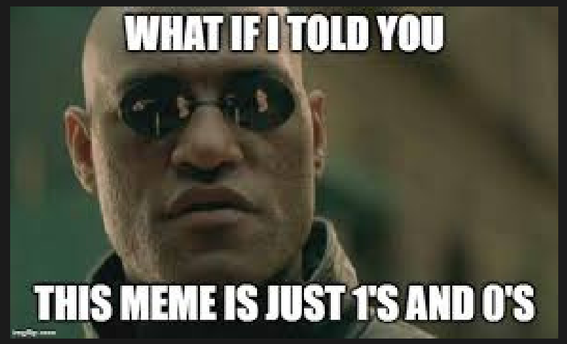
La HEAP

Mer. 04 Octobre 2023

SOMMAIRE

UN BINAIRE ? C'EST QUOI ?

00101011
01101010
10111010
11011000
10100110



CE SONT DES FICHIERS CONTENANT UNE SUITE D'OCTETS NON INTERPRETABLE DIRECTEMENT PAR L'HUMAIN MAIS QUI PEUT ÊTRE *INTERPRÉTÉ PAR UNE MACHINE* COMME UNE SUITE D'INSTRUCTIONS

LES ARCHITECTURES

CISC
(Complexe Instructions Set Computer)

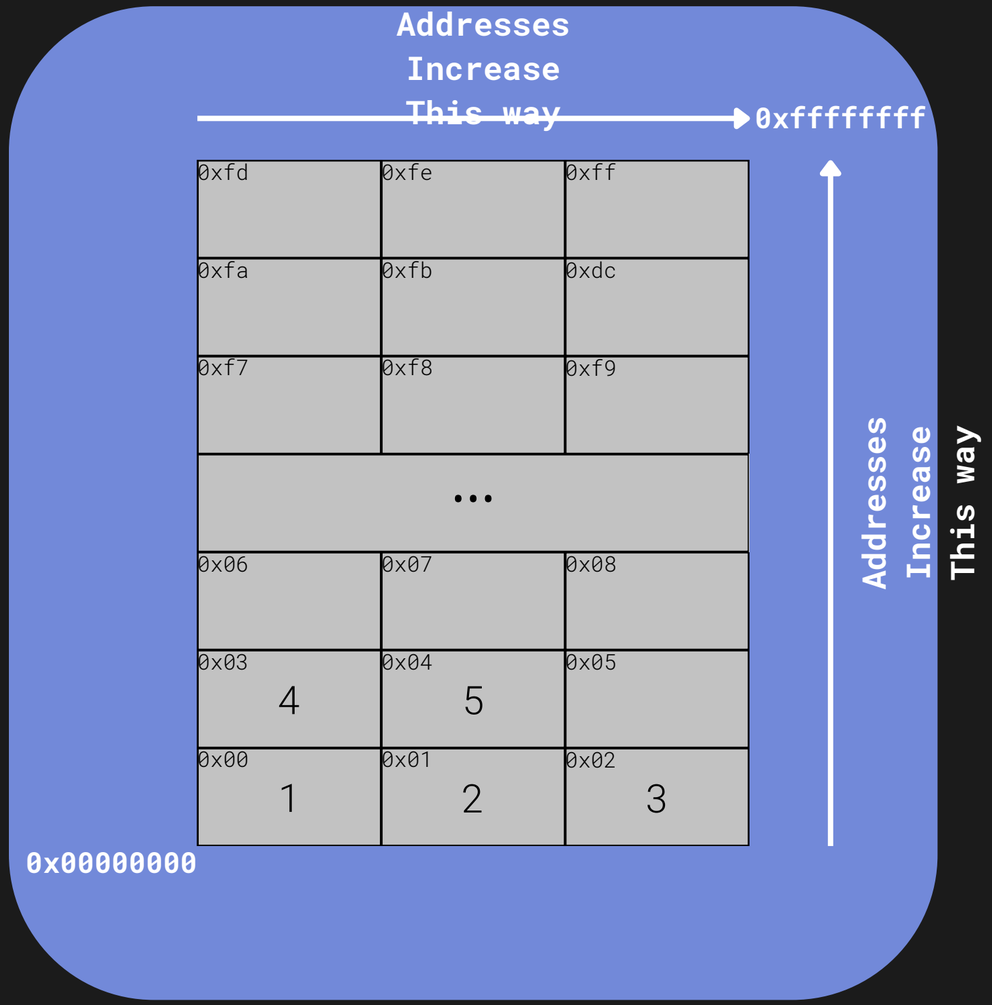
- Exemple : x86
- Architecture des PCs et serveurs
- Nombres d’instructions disponibles importantes
- version 32 ou 64 bits

ARM
(Complexe Instructions Set Computer)

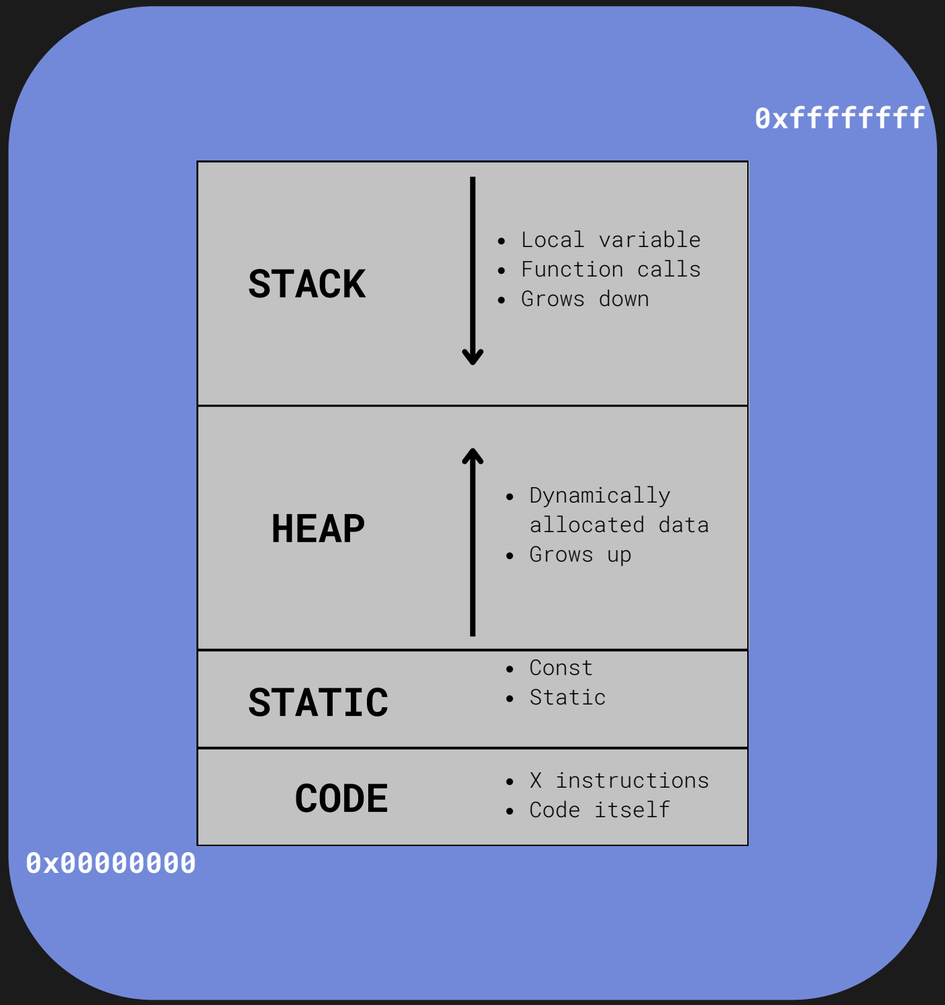
- Architecture des objets connectés et systèmes embarqués
- Instructions simples
- version 32 ou 64 bits

LA MÉMOIRE

32BITS



PROGRAM EXECUTION

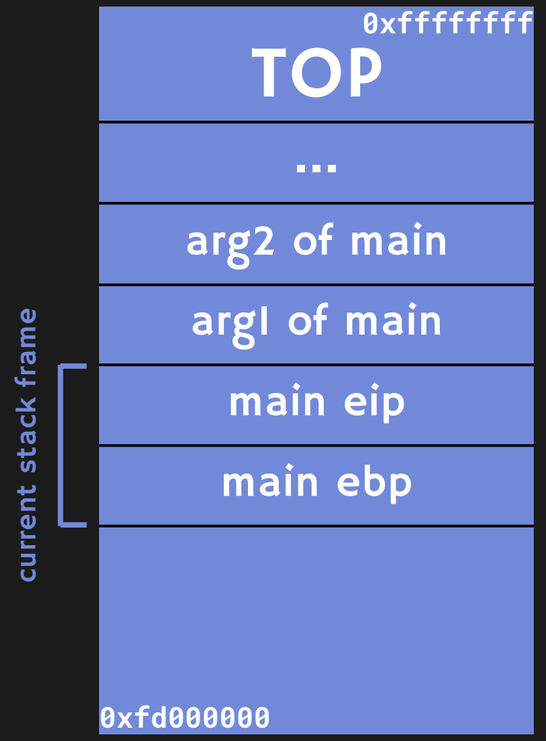


STACK OR HEAP ?

```
1  int foo(int arg1, int arg2){
2      int a;
3      int *p;
4      a = 10;
5      p=(int *)malloc(sizeof(int));
6      *p = 20;
7      free(p);
8      p=(int*)malloc(20*sizeof(int));
9      free(p);
10 }
```

eip

STACK



HEAP

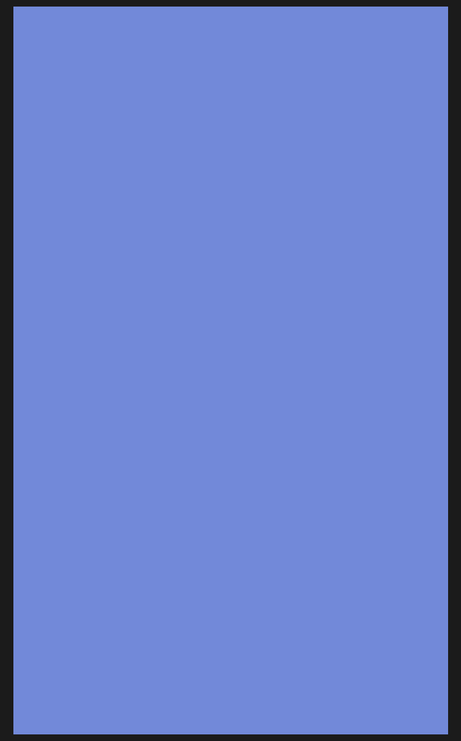
STACK OR HEAP ?

```
1  int foo(int arg1, int arg2){
2      int a;
3      int *p;
4      a = 10;
5      p=(int *)malloc(sizeof(int));
6      *p = 20;
7      free(p);
8      p=(int*)malloc(20*sizeof(int));
9      free(p);
10 }
```

eip

STACK

HEAP



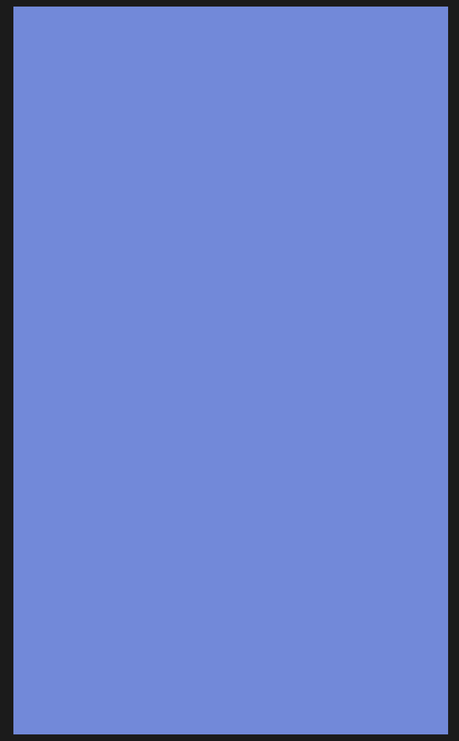
STACK OR HEAP ?

```c
int foo(int arg1, int arg2){
    int a;
    int *p;
    a = 10;
    p=(int *)malloc(sizeof(int));
    *p = 20;
    free(p);
    p=(int*)malloc(20*sizeof(int));
    free(p);
}
```

eip

STACK

HEAP



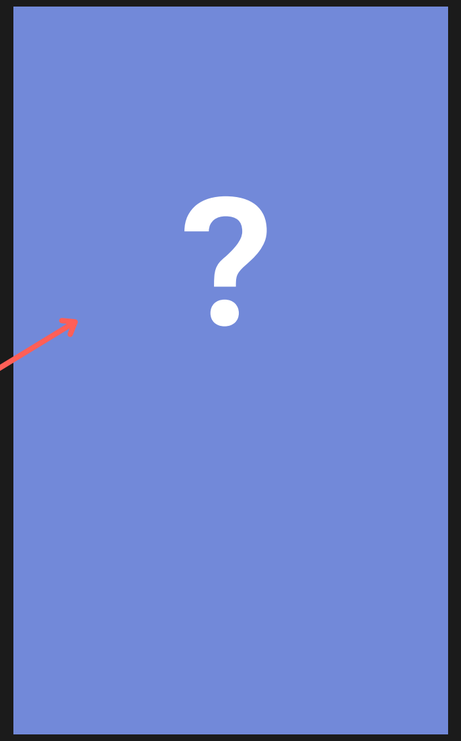
STACK OR HEAP ?

```
1  int foo(int arg1, int arg2){
2      int a;
3      int *p;
4      a = 10;
5      p=(int *)malloc(sizeof(int));
6      *p = 20;
7      free(p);
8      p=(int*)malloc(20*sizeof(int));
9      free(p);
10 }
```

eip

STACK

HEAP



STACK OR HEAP ?

Pourquoi la heap ?

- variables à durée de vie non connue
- durée de vie longue (plusieurs fonctions)

HEAP

Comment on l’utilise ?

- Fonctions malloc et free de la libc
- Malloc permet d'allouer dynamiquement de la mémoire
- Free libère un bloc de mémoire alloué dynamiquement

La libc fait le travail pour nous, mais si on veut l’exploiter,
il va falloir aller plus loin.

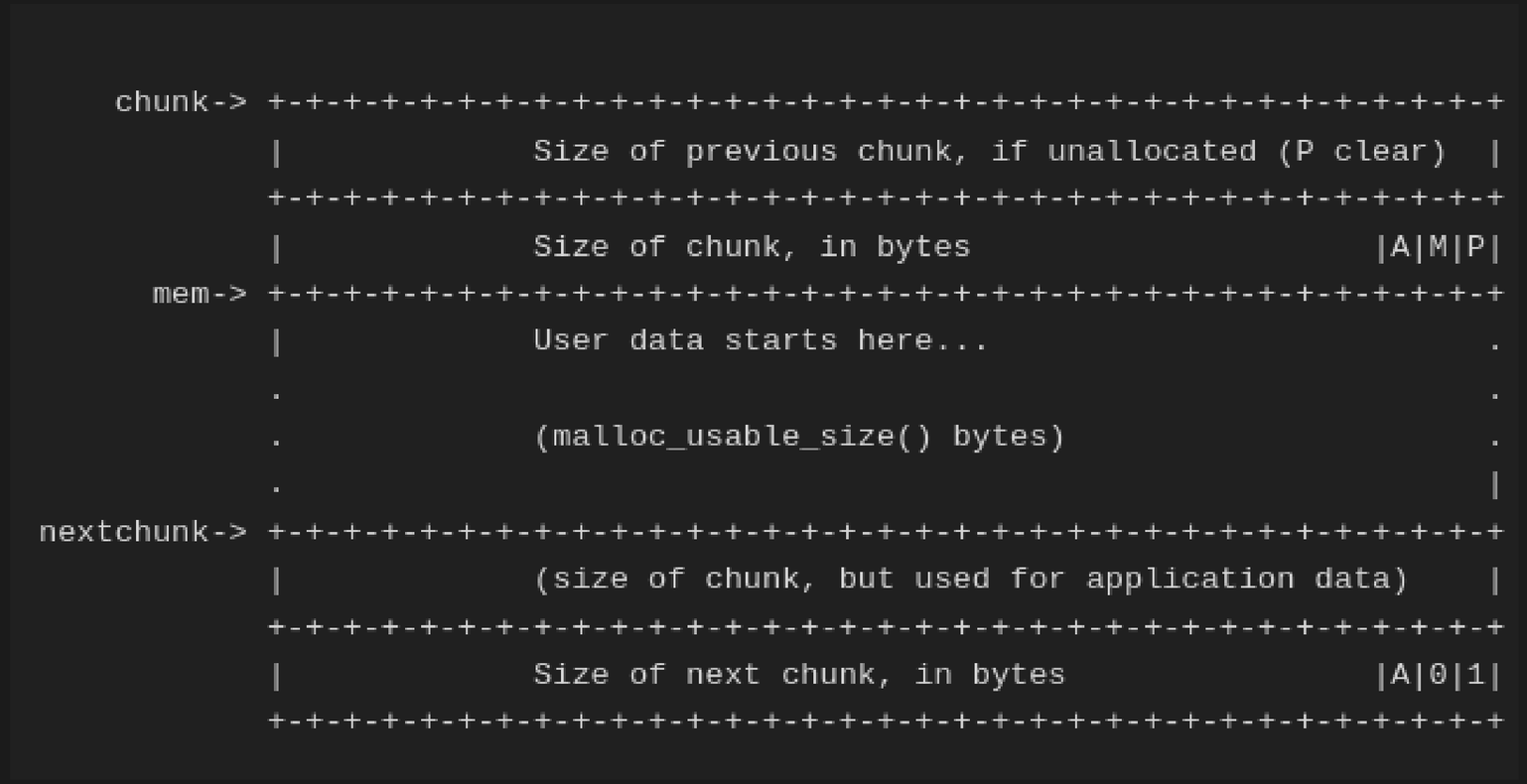
HEAP

Qu’est ce qu’on trouve dans la Heap ?

Des blocs de données appellés Chunk.
À chaque appel de malloc, un nouveau chunk est créé.
La libc garde une liste de tous les chunks en circulation.

HEAP

Qu’est ce qu’on trouve dans la Heap ?



Mem est le pointeur renvoyé à l’utilisateur lors d’un “malloc”. Chunk est gardé par la libc.

HEAP

Allocateur : ptmalloc2

Utilisé par malloc
Alloue de la mémoire dynamiquement
Fragmentation de la mémoire
Réutilisation de la mémoire libérée

HEAP

ptmalloc2 : Poubelles

4 types de poubelles
fastbin : petit bloc, allocation rapide
-> 10 fast bin : 8, 16, 24, ..., 88
-> liste simplement chainée

unsorted bin : unique, dès qu'un block est free, il va dedans et sera trié par la suite.
-> liste doublement chainée

HEAP

ptmalloc2 : Poubelles

4 types de poubelles :
small bins : listes doubles chainées de chunks de même taille.

62 *small bins*. Les taille des chunks vont de 16 à 504 octets.

HEAP

ptmalloc2 : Poubelles

4 types de poubelles :

large bins : listes doubles chainées de chunks rangés par taille décroissante.

Dans la 1ère, la taille des chunks est comprise entre 512 et 568 octets.
Dans la 2nd, la taille des chunks est comprise entre 576 et 632 octets etc. Et la dernière contient tous les blocks qui n'ont pu être rangés null part.

HEAP

ptmalloc2 : Malloc

Priorité d’allocation :

- (1) Si un chunk de taille exacte est dispo dans la fastbin, alors on l'utilise.
- (2) Si un chunk de taille exacte est dispo dans la smallbin, alors on l'utilise.
- (3) Si la requête est grosse, on transfère le contenu des fastbins dans la unsorted bin en les regroupant si possible
- (4) On trie les chunks de la unsorted bin dans les small et fast bins en les regroupant si possible. Si un chunk de taille exacte est dispo, on l'utilise
- (5) Si la requête est grosse, on cherche dans les largebins si un chunk est assez gros pour satisfaire la requête
- (6) S'il reste des chunks dans les fastbins (cela peut arriver pour de petites requêtes), on les regroupe et les tranfère dans la unsorted bin. Et on refait l'étape 6.
- (7) Si on a toujours pas trouvé notre bonheur, alors on avance le pointeur top de la heap de la quantité nécessaire.

HEAP

ptmalloc2 : Free

Priorité dans les poubelles :

- **(1) S'il y a une place en fastbin, on l'utilise**
- **(2) Si le chunk avait été mmap, alors on le munmap**
- **(3) Si le chunk est adjacent avec un autre, on les regroupe.**
- **(4) On place ce chunk dans la unsorted bin, sauf s'il est le top chunk**
- **(5) Si le chunk est gros, on regarde s'il est possible de faire reculer le top (en regroupant des fastbin si besoin)**

HEAP

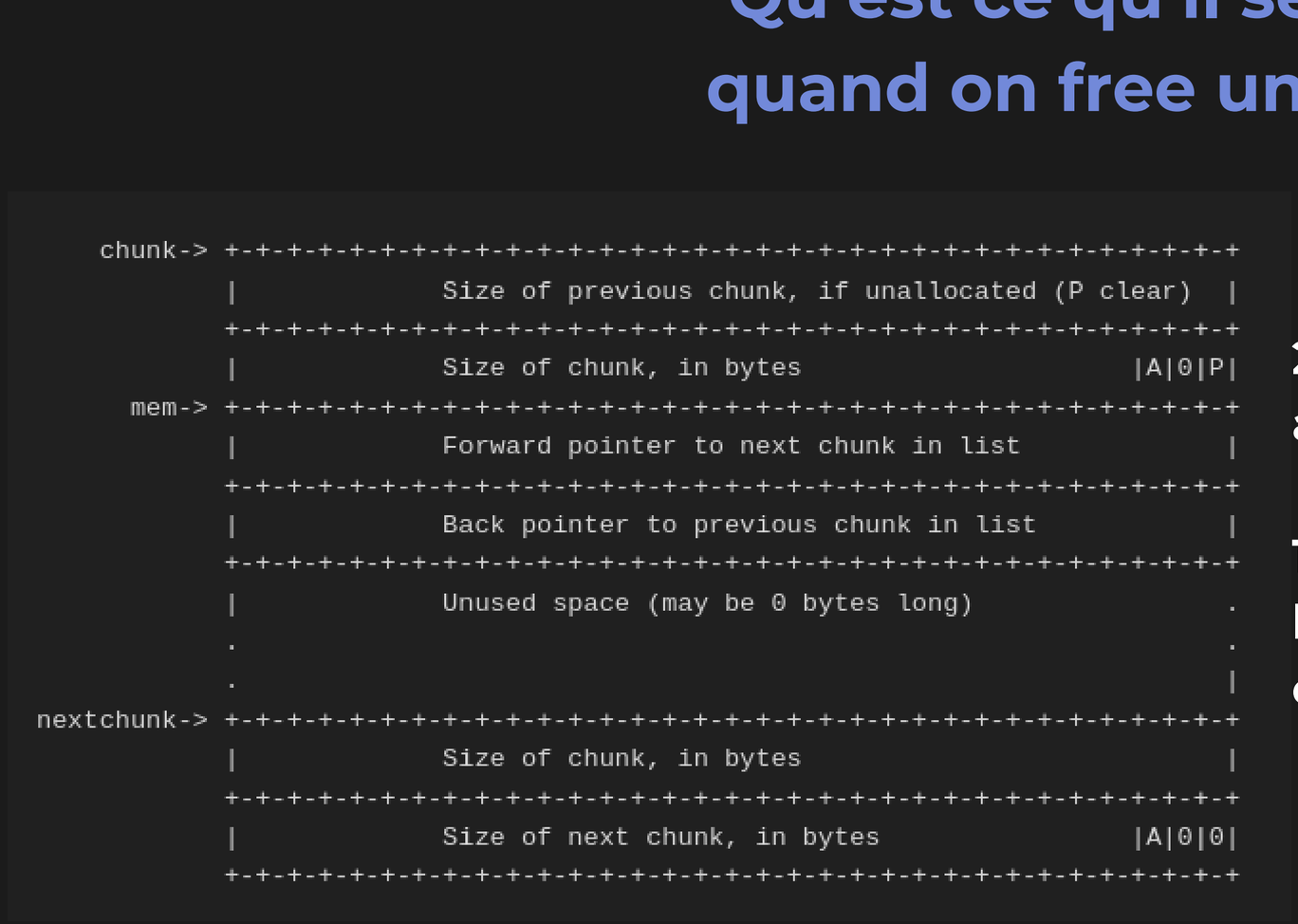
ptmalloc2 : Free

Priorité dans les poubelles :

- **(1) S'il y a une place en fastbin, on l'utilise**
- **(2) Si le chunk avait été mmap, alors on le munmap**
- **(3) Si le chunk est adjacent avec un autre, on les regroupe.**
- **(4) On place ce chunk dans la unsorted bin, sauf s'il est le top chunk**
- **(5) Si le chunk est gros, on regarde s'il est possible de faire reculer le top (en regroupant des fastbin si besoin)**

HEAP

Qu’est ce qu’il se passe quand on free un block?



2 pointeurs sont ajoutés dans notre ancienne user data.

1 vers le précédent chunk et l’autre vers le suivant. Le reste de user data ne change pas de valeur.

LES VULNS

double free

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

int main(){
  void *a,*b,*c;
  a=malloc(10);
  b=malloc(10);
  c=malloc(10);

  free(a);
  free(b);
  free(a);
  // A ce moment, la poubelle ressemble à ça:
  // chunk A -> chunk B -> chunk A

  // Comme il y a des chunks en poubelle
  // la libc les donne en priorité
  a=malloc(10);
  b=malloc(10);
  c=malloc(10);

  printf("%p %p %p\\n",a,b,c);
  // a et c ont la même adresse
}
```

appel plusieurs fois la fonction free() sur le même chunk.

Le chunck se retrouve plusieurs fois dans la poubelle qui lui correspond.

Possibilité d’avoir plusieurs pointeurs qui pointe vers le même chunk.

LES VULNS

use after free

```c
int main(int argc, char ** argv) {
        // Deux pointeurs admin et prénom, qui n'ont rien à voir l'un et l'autre dans le code
        char *admin = NULL;
        char *prenom = NULL;

        // Par défaut, l'utilisateur qui lance ce programme n'est pas administrateur. C'est tout.
        admin = malloc(32);
        admin[0] = 0;
        // Un moment, dans le code, la zone mémoire de admin est libérée, mais la variable admin n'est pas réinitialisée !
        free(admin);
        // Et puis une autre allocation de mémoire est faite.
        // Sauf que comme admin a été libéré, cette nouvelle zone mémoire réutilise cet espace !
        prenom = malloc(32);
        strncpy(prenom, "pixis", 5);
        // Ici, admin pointe toujours vers la zone mémoire initiale, qui a été réutilisée par "prenom".
        // Du coup, admin[0] vaut "p", admin[1] vaut "i", etc.
        // Ainsi, d'après cette vérification, nous sommes administrateur !
        if (admin == NULL || admin[0] == 0) {
                printf("Cette section est interdite !\n");
                return -1;
        }

        printf("Zone d'administration super secrète !\n");

        /*
         * Et puis du code [...]
         */

        free(prenom);
        prenom = NULL;
        return 0;
}
```

utilisation d’un chunk après qu’il soit libéré

A VOUS DE JOUER

Site : root-me
catégorie : app -system
Chall : ELF x64 - Double free
ELF x86 - Use After Free - basic