# Programmation C avancée

## Concepts & Outils
## pour le développement

maj 01/2023

# L'allocation dynamique

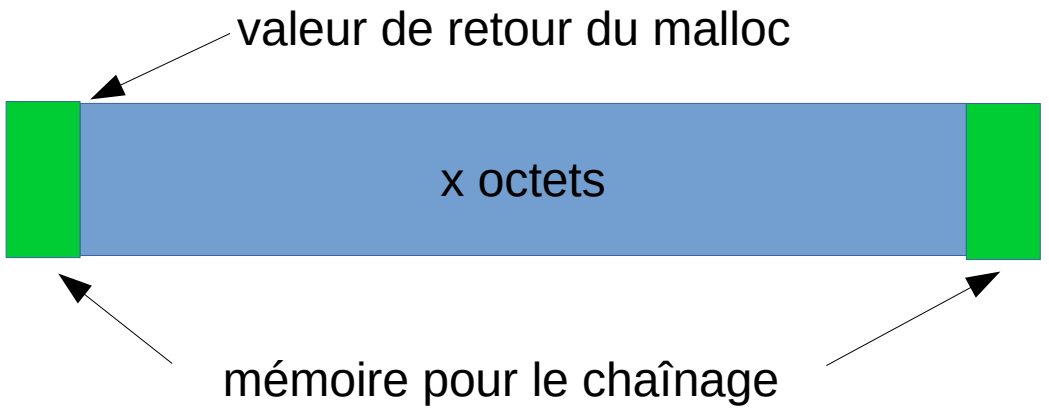
- Au lancement du processus, il y a une plage d'adresse non réservée (aucune page n'y est associée).
- L'appel à la fonction sbkr permet d'étendre cette plage d'adresse. La mémoire ainsi réservée peut être utilisée par le processus.
- Les fonctions malloc/free permettent une gestion de cette mémoire.

# L'allocation dynamique

- La structuration et la gestion du tas est liée à l'interprétation de cette mémoire par le gestionnaire : ici malloc.
- Il est tout à fait possible de remplacer ce gestionnaire par un autre.
- Tant qu'aucun appel à sbrk n'est fait pour rendre au système des pages mémoires, du point de vu du système, les pages correspondent à des zones accessibles (valides).
- malloc et free fonctionnent sur un système de listes chaînées

# malloc

- L'appel à malloc(x) permet de demander à celui-ci l'adresse d'une zone mémoire de x octets.
- malloc va réserver cette espace dans son système et renvoyer l'adresse de cette zone.
- La zone va être préfixée et post fixée par des pointeurs correspondants au chaînage de malloc :



# malloc

- démonstration :

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

int main(){
  int i=0;
  for(i=0;i<100000000;++i)
    malloc(4);
  sleep(60);
}
```

100M Allocations par malloc de 4 octets

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

int main(){
  int i=0;
  malloc(4*100000000);
  sleep(60);
}
```

Allocation par malloc de 400 Moctets

```
$ top -p
PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND

23020 allali    20   0 3129296 2,981g   1020 S   0,0 38,8   0:04.24 mem
23139 allali    20   0  394824    648    568 S   0,0  0,0   0:00.00 mem2
```

# Allocation dynamique

- Tant que l'on a pas indiqué à malloc qu'une zone mémoire n'est pas réutilisable, il ne va pas sans servir pour de nouvelles allocations.
- Contrairement à l'allocation automatique, la « durée de vie » de l'allocation dynamique n'est pas liée à l'appel ou au retour de fonction.
- Si à un moment il n'existe plus de variable (dans la pile ou statique) qui pointe directement ou indirectement vers une allocation dynamique alors il y a une « fuite mémoire » (memory leak).

# Allocation dynamique

- Sous linux, le fonctionnement de malloc peut être paramétré via des variables d'environnement ou la fonction mallopt :
  - MALLOC_CHECK_ : vérification au moment de la libération (double free)
  - MALLOC_PERTURB_ : permet de remplir la mémoire avec une valeur à l'allocation et de la ré-initialiser à une autre valeur à la libération. Très utile !

# Allocation : conclusion

- IMPORTANT : toute variable est soit dans la pile, soit dans le segment de donnée (statique).
- En aucun cas une variable de votre programme peut avoir une adresse qui soit dans le tas.
- L'utilisation de la mémoire dynamique suppose donc l'utilisation de pointeurs permettant de gérer les adresses mémoires.
- La structuration des allocations n'est pas connue par malloc : il est important d'interpréter ces zones correctement (taille).

# malloc : question

#QDLE#Q#A*B#30#

- L'utilisation de malloc au profit de l'allocation automatique dans des fonctions récursives permet d'élargir la capacité de traitement de celle-ci :
  - A. vrai
  - B. faux

# Encodage mémoire

- L’encodage mémoire correspond à la question de savoir comment interpréter une suite de bits.
- Le code ASCII donne une correspondance entre certains caractères et une valeur sur 1 octet: ‘A’ => 65 => 0100 0001
- Les entiers sont encodés en base 2: 65=64+1=2^6+2^0
- Les nombres réel (float) utilise un encodage normé dit basé sur une mantisse et un exposant
- Il existe d’autres encodage (wild char par exemple).
- Ainsi le type d’une variable indique combien d’octets doivent être considérés et comment interpréter ces octets.

# Encodage mémoire

- D’après la norme:
  - ’x’ vaut la représentation sur un octet prévue dans le code ASCII pour le symbole x
  - 0xFF est la notation hexadécimal, chaque symbole représente la valeur de 4 bits (0...9ABCDEF)
  - 012 : si le nombre commence par 0, alors c’est la représentation octale qui est utilisée, chaque chiffre est compris entre 0 et 7 et correspond à 3 bits. Ainsi 012 est la valeur décimale 10 ou encore 0xA
  - 0.1f par défaut, les nombres réels (avec un . ) sont des « double » en  ajoutant un f, on indique que le nombre est un float.
- Hors norme: 0b110111001  écriture binaire

# Les structures

- Le « padding » correspond à l'ajout d'octets dans une structure pour que la taille de celle-ci soit un multiple d'une valeur (1, 4 ou 8 par exemple).
- Cela dépend des types des champs:

```
struct A{
  char c;
}; => 1 octet
```

```
struct B{
  char c;
  int i;
}; => 8 octet
sizeof(int)=4
```

```
struct C{
  char c;
  int i;
  char d;
}; => 12 octet
sizeof(int)=4
```

```
struct D{
  char c;
  char d;
  int i;
}; => 8 octet
sizeof(int)=4
```

```
struct E{
  char c;
  char d;
  long i;
}; => 16 octet
sizeof(long)=8
```

- Ainsi, cela va dépendre des types car il y a des contraintes d'alignement pour les int, float, long etc... Les char représentant un octet, il n'y a pas de contrainte particulière.
- On fera attention à l'ordre des champs pour réduire le padding!

# Structures et bitfield

- Il est possible de déclarer des champs sur quelques bits:

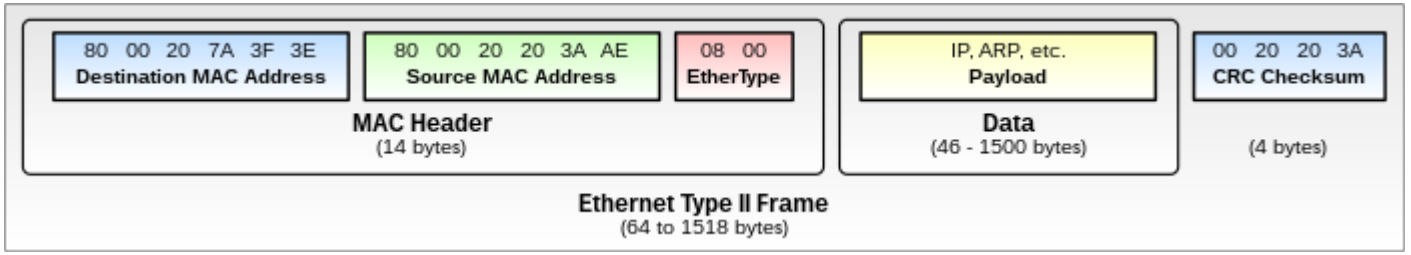
```
struct BF{
  unsigned a :3;
  unsigned b :5;
  unsigned c :2;
  unsigned d :1;
};

int main(){
  int i;
  struct BF a={0,0,0,0};
  for(i=0;i<14;++i){
    printf("a:%d b:%d c:%d d:%d\n",a.a,a.b,a.c,a.d);
    a.a++; a.b++; a.c++; a.d++;
  }
}
```

```
a:0 b:0 c:0 d:0
a:1 b:1 c:1 d:1
a:2 b:2 c:2 d:0
a:3 b:3 c:3 d:1
a:4 b:4 c:0 d:0
a:5 b:5 c:1 d:1
a:6 b:6 c:2 d:0
a:7 b:7 c:3 d:1
a:0 b:8 c:0 d:0
a:1 b:9 c:1 d:1
a:2 b:10 c:2 d:0
a:3 b:11 c:3 d:1
a:4 b:12 c:0 d:0
a:5 b:13 c:1 d:1
```

# Structures et bitfield

- bien que supporté par certains compilateurs, la norme ne prévoit les bitfield que sur les types unsigned int et int
- C’est le compilateur qui ajoute les opérations nécessaire pour les calculs (masque et décalage), ce n’est pas pris en charge nativement par le processeur.
- C’est un technique souvent utilisée dans des protocoles de communication car les trames (données) transmises suivent une description fixe:



# struct

- Au final, il est important de se rappeler que les struct n’existent pas dans les binaires
- C’est le compilateur qui ajuste les accès mémoires en fonction des champs
- Tout ce qu’on peut observer dans le code produit, ce sont des décalages d’adresses

- Il en est de même pour les noms de variables!

# Core

- Lors d'une faute mémoire (accès invalide à une page), le système génère une copie de la mémoire du processus sur le disque : un fichier core
- La commande ulimit permet de contrôler cette option : ulimit -c / ulimit -c unlimited

```
#include <stdlib.h>

int main(){
        char msg[]="ceci est un tres long message...";
       char *p=NULL;
       *p=0;
}
```

```
$ gcc bug.c
$ ulimit -c unlimited
$ ./a.out
Segmentation fault (core dumped)
$ ls -l core
-rw------- 1 allali 262144 core
$ strings core | grep ceci
ceci estH
ceci estH
ceci est un tres long message...
```

# Core: note

- attention, sur certaines distributions linux, la génération de fichiers core nécessite plusieurs manipulation:
- « $sysctl kernel.core_pattern »  permet de savoir ou sera généré le core et avec quel nom
- Si « apport » est utilisé au dessus, alors il faut démarrer ce service et les core sont placés par défaut dans /var/crash
- Le plus rapide étant de ne pas passer par « apport » en faisant:
- $sudo sysctl -w kernel.core_pattern=core.%u.%p.%t

# Core

- Le core contient l'ensemble des données mémoire : pile, tas, données, code…
- Il est possible de créer un core pour un processus actif avec la commande gcore
- Cela permet de contrôler la mémoire d'un processus à divers étapes de son exécution.
- On peut analyser un fichier core avec n'importe quel éditeur, par exemple :

  $ hexdump -c core | less

  mais l'interprétation en est très difficile !

# core

0000000 177 E L F 002 001 001 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000010 004 \0 > \0 001 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000020 @ \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000030 \0 \0 \0 \0 @ \0 8 \0 023 \0 \0 \0 \0 \0 \0 \0
0000040 004 \0 \0 \0 \0 \0 \0 h 004 \0 \0 \0 \0 \0 \0
0000050 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000060 d \v \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000070 \0 \0 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 005 \0 \0 \0
0000080 \0 020 \0 \0 \0 \0 \0 \0 \0 @ \0 \0 \0 \0 \0
0000090 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00000a0 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00000b0 001 \0 \0 \0 004 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00000c0 \0 \0 ` \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00000d0 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00000e0 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
00000f0 \0 0 \0 \0 \0 \0 \0 \0 \0 020 ` \0 \0 \0 \0 \0
0000100 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000110 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000120 001 \0 \0 \0 005 \0 \0 \0 \0 @ \0 \0 \0 \0 \0 \0
0000130 \0 0 <C0> S <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000140 \0 020 \0 \0 \0 \0 \0 \0 \0 <A0> 033 \0 \0 \0 \0 \0
0000150 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 \0 \0 \0 \0
0000160 \0 P \0 \0 \0 \0 \0 \0 \0 <D0> <DB> S <BF> 177 \0 \0
0000260 \0 0 002 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000270 001 \0 \0 \0 006 \0 \0 \0 \0 020 001 \0 \0 \0 \0 \0
0000280 \0 <A0> 034 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000290 \0 0 \0 \0 \0 \0 \0 \0 \0 0 \0 \0 \0 \0 \0 \0
00002a0 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
00002b0 \0 @ 001 \0 \0 \0 \0 \0 \0 200 036 T <BF> 177 \0 \0
00002c0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00002d0 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00002e0 001 \0 \0 \0 004 \0 \0 \0 \0 ` 001 \0 \0 \0 \0 \0
00002f0 \0 <A0> 036 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000300 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000310 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
0000320 \0 p 001 \0 \0 \0 \0 \0 \0 <B0> 036 T <BF> 177 \0 \0
0000330 \0 \0 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000340 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000350 001 \0 \0 \0 006 \0 \0 \0 \0 200 001 \0 \0 \0 \0 \0
0000360 \0 <C0> 036 T <BF> 177 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000370 \0 020 \0 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
0000380 \0 020 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 006 \0 \0 \0
0000390 \0 220 001 \0 \0 \0 \0 \0 \0 <F0> 200 \t <FF> 177 \0 \0
00003a0 \0 \0 \0 \0 \0 \0 \0 \0 \0 002 \0 \0 \0 \0 \0
00003b0 \0 002 \0 \0 \0 \0 \0 \0 020 \0 \0 \0 \0 \0 \0
00003c0 001 \0 \0 \0 005 \0 \0 \0 \0 <B0> 003 \0 \0 \0 \0 \0

# gdb

- gdb : GNU Debugger
- permet d'analyser la mémoire avec une couche d'interprétation.
- Il est possible de
  - prendre le contrôle d'un processus en cours d’exécution
  - de lancer un nouveau processus dans gdb
  - d'analyser la mémoire à posteriori, hors exécution (core).

# gdb : run

- On lance gdb en indiquant le binaire que l'on souhaite analyser, possiblement suivi d'un pid (processus en cours) ou d'un fichier core
- gdb va charger le fichier binaire
- Dans le cas d'un pid, il va stopper le processus
- Il est possible de lancer un nouveau processus avec la commande :

  run arg1 arg2 …

- Ctrl-C permet de suspendre l'exécution du programme, « continue » de la reprendre.

# gdb

- En l'absence d'informations additionnelles, gdb ne peut pas associer le contenu mémoire à des variables structurées et donc interpréter la mémoire.
- Il peut cependant indiquer la pile d'appel :

```
(gdb) backtrace
#0  0x00000000004004fa in f ()
#1  0x00000000040050f in g ()
#2  0x0000000000400522 in h ()
#3  0x0000000000400535 in main ()
```

- On voit ici, les adresses dans la pile correspondant aux débuts d'appels.

# gdb

- Si le binaire contient des symboles de debug alors on peut :
  - lister le code (list)
  - placer un arrêt d'exécution sur une ligne spécifique (break)
  - afficher le contenu d'une variable (print ou display)
  - avancer d'une ligne de code (next ou step)
- Pour avoir une binaire avec symboles de debug, il faut compiler avec l’option « -g ». L’option « -O0 » est aussi préconisée pour éviter des optimisions excessives du binaire.

# gdb commandes de base

- « **help** » ou « help command »
- **list** (l) : affiche 10 lignes du fichier source (point d'execution courant)
- **backtrace** (bt) : permet de voir la pile d'appel des fonctions, chaque « zone » d'appel s'appelle une *stack frame*
- **run** (r) : execute le binaire chargé
- **continue** (c): reprend l'exécution
- **call** f(arg) : permet l'appel d'une fonction
- **print** (p) variable: affiche une variable, permet de modifier sa valeur: p i=10
- **display** (disp) expression: permet l'affichage après chaque arrêt (undisp pour desactiver un affichage récurrent).
- **next** (n) : execute une ligne de code, ne rentre pas dans une fonction
- **step** (s): execute une ligne de code mais rentre dans une fonction

# gdb les points d’arrêt

- Il n’y  a que deux états possibles: soit le programme est suspendu et dans ce cas on peut analyser le contenu mémoire, soit il s’execute et dans ce cas on ne peut rien faire
- Pour arrêter l’execution on peut:
  - utiliser ctrl-c mais le moment de l’arrêt n’est pas maitrisé
  - placer un break point
- **break** (b) : permet de placer un point d’arrêt. On peut préciser le nom d’un fonction: « b main » ou bien un numéro de ligne du code source courant: « b 3 » ou « b source.c:10 »
- On peut afficher la liste des breakpoints avec « **info breakpoints** »
- La suppression se fait avec « **del** N », la désactivation avec « **disa** N» et l’activation avec « **ena** N»

# gdb: commandes++

- Quelques commandes avancées :
  - unroll : sortie de boucle
  - finish : sortie de fonction
  - cond : conditionne l'activation d'un point d'arrêt
  - watch (surveiller le changement de contenu d'une case mémoire)
  - disa / ena / del : gestion des points d'arrêt.
  - up / down : déplacement dans la pile
  - Info / show

# gdb : cas d'école !

- mon programme fait un SEGFAULT
- 1 ⇒ j'exécute (dans gdb ou bien core)
- 2 ⇒ j'identifie la cause directe du problème :

```
Reading symbols from a.out...done.
(gdb) r
Starting program: /tmp/a.out

Program received signal SIGSEGV, Segmentation fault.
__strcpy_sse2_unaligned ()
    at ../sysdeps/x86_64/multiarch/strcpy-sse2-unaligned.S:682
682     ../sysdeps/x86_64/multiarch/strcpy-sse2-unaligned.S: No such file or directory.
(gdb) bt
#0  __strcpy_sse2_unaligned ()
    at ../sysdeps/x86_64/multiarch/strcpy-sse2-unaligned.S:682
#1  0x00000000004006cc in main (argc=1, argv=0x7fffffffe058) at s.c:13
(gdb) up
#1  0x00000000004006cc in main (argc=1, argv=0x7fffffffe058) at s.c:13
13          strcpy(p,argv[i]);
(gdb) p p
$1 = 0x0
(gdb) p i
$2 = 0
(gdb) p argv[i]
$3 = 0x7fffffffe38d "/tmp/a.out"
(gdb)
```

# gdb : cas d'école !

- je remonte la pile jusqu'à mon code
- je trouve la valeur qui pose problème (ici p)
- je liste le code :

```
(gdb) l
8       l+=1;
9       s=malloc(sizeof(char)*l);
10      if (s=NULL) return EXIT_FAILURE;
11      p=s;
12      for(i=0;i<argc;++i) {
13        strcpy(p,argv[i]);
14        p+=strlen(argv[i]);
15      }
16      puts(s);
17      return EXIT_SUCCESS;
```

- La valeur de p est « fixée » ligne 11. Je vérifie cela :
- « break 11 »
- « run »

# gdb : cas d'école !

```
Starting program: /tmp/a.out

Breakpoint 1, main (argc=1,
argv=0x7fffffffe058) at s.c:11
11      p=s;
(gdb) p s
$4 = 0x0
(gdb) l
6       int i,l=0;
7       for(i=0;i<argc;++i) l+=strlen(argv[i]);
8       l+=1;
9       s=malloc(sizeof(char)*l);
10      if (s=NULL) return EXIT_FAILURE;
11      p=s;
12      for(i=0;i<argc;++i) {
13        strcpy(p,argv[i]);
14        p+=strlen(argv[i]);
15      }
(gdb) b 9
Breakpoint 2 at 0x40066e: file s.c, line 9.
```

- apparemment la valeur de s est déjà à 0
- s est initialisée ligne 9
- « b 9 »
- « r »

# gdb : cas d'école !

```
Breakpoint 2, main (argc=1,
argv=0x7fffffffe058) at s.c:9
9       s=malloc(sizeof(char)*l);
(gdb) next
10      if (s=NULL) return EXIT_FAILURE;
(gdb) p s
$5 = 0x602010 ""
(gdb) display s
1: s = 0x602010 ""
(gdb) n

Breakpoint 1, main (argc=1,
argv=0x7fffffffe058) at s.c:11
11      p=s;
1: s = 0x0
(gdb)
```

- je passe la ligne avec « next » puis je contrôle la valeur de s
- apparemment s change de valeur (de 0x602010 à 0x0)
- je fais un *display* puis j'exécute pas à pas.
- après la ligne 10, la valeur a changé...

# gdb : autres

- Il existe d'autres interfaces pour gdb :
  - xxgdb
  - ddd
  - kgdb
  - etc...
- La plus part des environnements de développement propose un *debuger* intégré :
  - visual studio / vscode / CLion
  - qtcreator
  - kdevelop
  - etc...

# question

#QDLE#Q#AB*C#20#

- malloc renvoie une adresse située :
  - A. dans le segment de donnée
  - B. dans le tas
  - C. dans la pile

# tracer les accès mémoires

- valgrind [vælgrɪnd] est un outil qui intègre de nombreux tests pour l'exécution de programmes.
- Par défaut, c’est l’outil **memcheck** qui est utilisé:
  - Permet de détecter des erreurs d'exécution qui ne sont pas relevées par le système comme le dépassement d'un tableau par exemple.
  - L'exécution du programme par valgrind est contrôlée pas à pas, ce qui ralenti beaucoup le programme. Il y a également une surconsommation mémoire importante.

# memcheck

- memcheck propose de nombreuses options pour contrôler son execution, cela peut permettre d’exécuter valgrind sur des programmes « lourds »

--leak-check=<no|summary|yes|full> [default: summary]
--leak-resolution=<low|med|high> [default: high]
--show-leak-kinds=<set> [default: definite,possible]
--errors-for-leak-kinds=<set> [default: definite,possible]
--leak-check-heuristics=<set> [default: all]
--leak-check-heuristics=stdstring,length64,newarray,multipleinheritance.
--show-reachable=<yes|no> , --show-possibly-lost=<yes|no>
--xtree-leak=<no|yes> [no]
--xtree-leak-file=<filename> [default: xtleak.kcg.%p]
--undef-value-errors=<yes|no> [default: yes]
--track-origins=<yes|no> [default: no]
....

# memcheck

- memcheck propose de nombreuses options pour contrôler son execution, cela peut permettre d’exécuter valgrind sur des programmes « lourds »

--partial-loads-ok=<yes|no> [default: yes]
--expensive-definedness-checks=<no|auto|yes> [default: auto]
--keep-stacktraces=alloc|free|alloc-and-free|alloc-then-free|none [default: alloc-and-free]
--freelist-vol=<number> [default: 20000000]
--freelist-big-blocks=<number> [default: 1000000]
--workaround-gcc296-bugs=<yes|no> [default: no]
--ignore-range-below-sp
--ignore-range-below-sp=<number>-<number>
--ignore-range-below-sp=8192-8189. Only one range may be specified.
--show-mismatched-frees=<yes|no> [default: yes]
--ignore-ranges=0xPP-0xQQ[,0xRR-0xSS]
--malloc-fill=<hexnumber>
--free-fill=<hexnumber>

# memcheck

- les erreurs détectés par memcheck sont:
  - Invalid read of size 4
  - Conditional jump or move depends on uninitialised value(s)
  - Invalid free()
  - Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)
  - fuites mémoires...

# valgrind

- Pour pouvoir fonctionner efficacement avec valgrind, le programme doit avoir été compilé avec -g et -O0
- ensuite, on lance le programme dans valgrind :

  valgrind ./a.out arg1 arg2 ...
- valgrind va afficher les erreurs mémoires au fur et à mesure de leurs apparitions
- Il est possible de demander à valgrind de lancer gdb à chaque erreur :

  valgrind --vgdb-error=1 ./a.out

# exemple :

```
allali@hebus:/tmp$ valgrind --vgdb-error=1 ./a.out
==9539== Memcheck, a memory error detector
==9539== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9539== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==9539== Command: ./a.out
==9539==
==9539==
==9539== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==9539==   /path/to/gdb ./a.out
==9539== and then give GDB the following command
==9539==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=9539
==9539== --pid is optional if only one valgrind process is running
==9539==
==9539== Invalid write of size 4
==9539==    at 0x400570: main (s.c:7)
==9539==  Address 0x51fc050 is 0 bytes after a block of size 16 alloc'd
==9539==    at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck.so)
==9539==    by 0x40054E: main (s.c:4)
==9539==
==9539== (action on error) vgdb me ...
```

Toute erreur signalée par valgrind mérite une analyse et, à de très rares exceptions, doit être corrigée !

```
allali@hebus:/tmp$ gdb a.out
...
Reading symbols from a.out...done.
(gdb) target remote | /usr/lib/valgrind/../../bin/vgdb --pid=9539
…
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x0000000000400570 in main (argc=1, argv=0xffeffff98) at s.c:7
7           t[i]=0;
(gdb) p t
$1 = (int *) 0x51fc040
(gdb) p i
$2 = 4
```

```
1 #include<stdlib.h>

3 int main(int argc, char **argv){
4   int *t=malloc(sizeof(int)*4);
5   int i;
6   for(i=0;i<6;++i)
7     t[i]=0;
8    return 0;
9 }
```

# valgrind: lire le message

```
int main(){
  int *p=malloc(sizeof(int)*3);
  int i;
  for(i=0;i<5;++i)
    p[i]=0;
}
```

```
==253883== Invalid write of size 4
==253883==    at 0x109180: main (a.c:8)
==253883==  Address 0x4ab304c is 0 bytes after a block of size 12 alloc'd
==253883==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==253883==    by 0x10915E: main (a.c:5)
```

# valgrind: lire le message

```
int main(){
  int *p=malloc(sizeof(int)*3);
  int i,s=0;
  for(i=0;i<5;++i)
    s+=p[i];
}
```

```
==254030== Invalid read of size 4
==254030==    at 0x109187: main (a.c:8)
==254030==  Address 0x4ab304c is 0 bytes after a block of size 12 alloc'd
==254030==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux
==254030==    by 0x10915E: main (a.c:5)
==254030==
```

# valgrind: lire le message

```
int main(){
  int *p=malloc(sizeof(int)*3);
  int i,s=0;
  if (p[i]==0){
    s=1;
  }
}
```

```
==254181== Use of uninitialised value of size 8
==254181==    at 0x10917E: main (a.c:7)
==254181==
==254181== Conditional jump or move depends on uninitialised value(s)
==254181==    at 0x109182: main (a.c:7)
==254181==
```

# valgrind: limite avec la pile

- Valgrind détecte bien les erreurs liées à la mémoire sur le tas mais moins bien lorsque la mémoire concernée est dans la pile:
- ce code ne génère pas d'erreur alors qu'on utilise de la mémoire non initialisée!

```
int f(){
  int p[4];
  return p[2];
}

int main(){
  int i=0;
  i+=f();
  i+=f();

}
```

- Si l'on utilise i plus tard, alors une erreur peut apparaître.

# valgrind : fuite mémoire

#QDLE#Q#ABCD*#45#

- Une fuite mémoire correspond à de la mémoire encore réservée à la fin de votre programme
- Celle-ci peut être ?
  - A. dans la pile et le tas
  - B. dans la pile, le tas et le segment de données
  - C. uniquement dans la pile
  - D. uniquement dans le tas

# valgrind : fuite mémoire

- A la fin de l'exécution, valgrind liste les blocs non libérés et les classe selon :
  - que plus rien ne pointe sur ce segment (definitively lost)
  - qu'un autre segment de mémoire pointe encore dessus (indirectly lost)
  - encore accessible par une variable statique (still reachable)

# valgrind : definitively lost

```
allali@hebus:/tmp$ valgrind --leak-check=full ./a.out
==9900== Memcheck, a memory error detector
==9900== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9900== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==9900== Command: ./a.out
==9900==
==9900==
==9900== HEAP SUMMARY:
==9900==     in use at exit: 1 bytes in 1 blocks
==9900==   total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==9900==
==9900== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9900==    at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9900==    by 0x400543: main (s.c:4)
==9900==
==9900== LEAK SUMMARY:
==9900==    definitely lost: 1 bytes in 1 blocks
==9900==    indirectly lost: 0 bytes in 0 blocks
==9900==      possibly lost: 0 bytes in 0 blocks
==9900==    still reachable: 0 bytes in 0 blocks
==9900==         suppressed: 0 bytes in 0 blocks
==9900==
==9900== For counts of detected and suppressed errors, rerun with: -v
==9900== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
#include<stdlib.h>

int main(){
  malloc(sizeof(char));
}
```

# valgrind : indirectly lost

```
allali@hebus:/tmp$ valgrind --leak-check=full --show-reachable=yes ./a.out
==9973== Memcheck, a memory error detector
==9973== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9973== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==9973== Command: ./a.out
==9973==
==9973==
==9973== HEAP SUMMARY:
==9973==     in use at exit: 9 bytes in 2 blocks
==9973==   total heap usage: 2 allocs, 0 frees, 9 bytes allocated
==9973==
==9973== 1 bytes in 1 blocks are indirectly lost in loss record 1 of 2
==9973==    at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9973==    by 0x400555: main (s.c:5)
==9973==
==9973== 9 (8 direct, 1 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
==9973==    at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9973==    by 0x400547: main (s.c:4)
==9973==
==9973== LEAK SUMMARY:
==9973==    definitely lost: 8 bytes in 1 blocks
==9973==    indirectly lost: 1 bytes in 1 blocks
==9973==      possibly lost: 0 bytes in 0 blocks
==9973==    still reachable: 0 bytes in 0 blocks
==9973==         suppressed: 0 bytes in 0 blocks
==9973==
==9973== For counts of detected and suppressed errors, rerun with: -v
==9973== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
#include<stdlib.h>

int main(){
  char **p=malloc(sizeof(char *));
  p[0]=malloc(sizeof(char ));
}
```

# valgrind : still reachable

```
allali@hebus:/tmp$ valgrind --leak-check=full --show-reachable=yes ./a.out
==9996== Memcheck, a memory error detector
==9996== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9996== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==9996== Command: ./a.out
==9996==
==9996==
==9996== HEAP SUMMARY:
==9996==     in use at exit: 1 bytes in 1 blocks
==9996==   total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==9996==
==9996== 1 bytes in 1 blocks are still reachable in loss record 1 of 1
==9996==    at 0x4C2ABA0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9996==    by 0x400543: main (s.c:5)
==9996==
==9996== LEAK SUMMARY:
==9996==    definitely lost: 0 bytes in 0 blocks
==9996==    indirectly lost: 0 bytes in 0 blocks
==9996==      possibly lost: 0 bytes in 0 blocks
==9996==    still reachable: 1 bytes in 1 blocks
==9996==         suppressed: 0 bytes in 0 blocks
==9996==
==9996== For counts of detected and suppressed errors, rerun with: -v
==9996== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
#include<stdlib.h>

char *p;
int main(){
  p=malloc(sizeof(char ));
}
```

# valgrind

#QDLE#Q#AB*CD#30#

```
int main(){
    int **p=malloc(sizeof(*p)*3) ;
    p[2]=malloc(sizeof(**p)*4) ;
    free(p) ;
}
```

- Sur cet exemple, quel est le type de la fuite ?

A. direct  B. indirect  C. still  D. pas de fuite

# valgrind

- options :
  - --leak-check=full --show-reachable=yes
  - --malloc-fill=<hexnumber>
  - --free-fill=<hexnumber>
  - …
- Attention : un programme n'ayant pas de problème avec valgrind n'est pas nécessairement un programme sans bug !

# SSL BuG in Debian

- May 2006, bug report (debian maintainer):

The problems are the following 2 pieces of code in crypto/rand/md_rand.c:

247:

```
        MD_Update(&m,buf,j);
```

467:

```
#ifndef PURIFY
        MD_Update(&m,buf,j); /* purify complains */
#endif
```

What it's doing is adding uninitialised numbers to the pool to create random numbers.

Valgrind: ”Use of uninitialised value of size ...”

# SSL BuG in Debian

- patch:

The problems are the following 2 pieces of code in crypto/rand/md_rand.c:

246:

```
                /**Don't add uninitialised datas
                 * MD_update(&m, buf, j);
                */
```

467:

```
#ifndef PURIFY
                /* MD_Update(&m,buf,j); /* purify complains */ */
#endif
```

What it's doing is adding uninitialised numbers to the pool to create random numbers.

=> Le générateur de nombre aléatoire de ssl ne dépend plus que du pid du process (32768 valeurs possibles) et devient donc prédictible!

# Moralité

- Identifier un bug est une chose, le corriger en est une autre

- Réfléchir à deux fois avant de mettre quelque chose en production!