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  0x000000000040050f 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...

exercices

- Télécharger le fichier array.zip et extraire le fichier source array.c
- Compiler ce fichier et executer le programme
- Lancer gdb sur le programme
- Dans gdb:
  - Tester la commande **run**
  - Placer un point d'arrêt : **b main**
  - Relancer le programme puis tester les commande **n** puis **s**. Utiliser **help n** et **help s**
  - Afficher la valeur d'une variable avec **p**

Exercices

- Placer un point d’arrêt sur la fonction **array_set**
- Vérifier avec **run** suivi de **continue**
- **info breakpoints**
- Utiliser **cond** pour faire que **array_set** ne s’arrête que si le paramètre **p** vaut **2**
- Faites **run** et identifier la raison du SegFault
- Utiliser la commande **watch** pour identifier les moments dans l’exécution où la donnée posant soucis est modifiée.
- Corriger le bug

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 ...
```

```
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 }
```

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

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

- 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

- 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 !

Exercices

- Utiliser le fichier **array_valgrind.c** d'origine et compiler avec les symboles de debug.
- Lancer l'exécution avec **valgrind**
- Lisez attentivement les messages et expliquer leurs significations
- Relancer avec **valgrind --malloc-fill=0x42 --free-fill=0x15** et observer les changements
- Enfin, relancer avec la commande :

  valgrind --malloc-fill=0x42 --free-fill=0x15 --vgdb-error=1

-fsanitize

- AddressSanitizer est est détecteur de problèmes mémoires développé par Google.
- C’est un outil rapide et à faible empreinte qui permet de détecter des problèmes comme les fuites mémoires, la réutilisation de mémoire libérée, le dépassement de mémoire (tableaux) …
- Pour l’utiliser, il suffit de compiler le code avec l’option : -fsanitize=address

-fsanitize=address

```
#include <stdlib.h>
int main() {
  char *x = (char*)malloc(10 * sizeof(char*));
  free(x);
  return x[5];
}
```

```
$ gcc -fsanitize=address -g a.c
$ ./a.out
=================================================================
==1120085==ERROR: AddressSanitizer: heap-use-after-free on address 0x607000000105 at pc 0x55b05950f22b bp 0x7fff4fb85710 sp
0x7fff4fb85700
READ of size 1 at 0x607000000105 thread T0
    #0 0x55b05950f22a in main /tmp/a.c:5
    #1 0x7fa2ff029d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7fa2ff029e3f in __libc_start_main_impl ../csu/libc-start.c:392
    #3 0x55b05950f104 in _start (/tmp/a.out+0x1104)


0x607000000105 is located 5 bytes inside of 80-byte region [0x607000000100,0x607000000150)
freed by thread T0 here:
    #0 0x7fa2ff4b4537 in __interceptor_free ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127
    #1 0x55b05950f1ee in main /tmp/a.c:4
    #2 0x7fa2ff029d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58


previously allocated by thread T0 here:
    #0 0x7fa2ff4b4887 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x55b05950f1de in main /tmp/a.c:3
    #2 0x7fa2ff029d8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58


SUMMARY: AddressSanitizer: heap-use-after-free /tmp/a.c:5 in main
Shadow bytes around the buggy address:
  0x0c0e7fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0e7fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0e7fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c0e7fff8000: fa fa fa fa fd fd fd fd fd fd fd fd fd fa fa fa
  0x0c0e7fff8010: fa fa 00 00 00 00 00 00 00 00 05 fa fa fa fa fa
=>0x0c0e7fff8020:[fd]fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x0c0e7fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Exercices:

- Recompiler le fichier **array.c** avec l'option

  -fsanitize=address
- Executer votre programme
- Refaire de même avec le fichier **array_valgrind.c**

clang-tidy

- clang-tidy est un analyseur statique de code
- Il permet de faire de nombreux tests sur les fichiers sources à la recherche de bug potentiels.

clang-tidy

```
#include <stdio.h>

int main(int argc, char **argv){
  int i;
  if (argc>3) i=0;
  printf("%d\n",i);
}
```

```
$ clang-tidy foo.c -- -I.
1 warning generated.
/tmp/pg120/foo.c:6:3: warning: 2nd function call argument is an uninitialized value [clang-analyzer-core.CallAndMessage]
  printf("%d\n",i);
  ^           ~
/tmp/pg120/foo.c:4:3: note: 'i' declared without an initial value
  int i;
  ^~~~
/tmp/pg120/foo.c:5:7: note: Assuming 'argc' is <= 3
  if (argc>3) i=0;
      ^~~~~
/tmp/pg120/foo.c:5:3: note: Taking false branch
  if (argc>3) i=0;
  ^
/tmp/pg120/foo.c:6:3: note: 2nd function call argument is an uninitialized value
  printf("%d\n",i);
  ^           ~
```

Exercices

- Lancer le programme **clang-tidy**:

  clang-tidy array.c
- Lire attentivement les messages fournis par cet outil.
- Faire de même avec **array_valgrind.c**
- Écrire un Makefile qui compile les programmes en passant par les .o
- Dans la ligne de production d’un .o, ajouter avant la compilation une utilisation de **clang-tidy**