GNU Debugger gdb

Notion de débogueur

Un débogueur (ou débugueur, de l'anglais *debugger*) est un logiciel qui aide un développeur à analyser les *bugs* d'un programme.

Pour cela, il permet d'exécuter le programme pas-à-pas, d'afficher la valeur des variables à tout moment, de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme ...

Le programme à débuguer est exécuté à travers le débogueur et s'exécute normalement. Le débogueur offre alors au programmeur la possibilité d'observer et de contrôler l'exécution du programme.

gdb

GNU Debugger, également appelé `gdb`, est le débogueur standard du projet GNU. Il est portable sur de nombreux systèmes type Unix et fonctionne pour plusieurs langages de programmation, comme le C et le C++. Il fut écrit par Richard Stallman en 1988.

gdb est un logiciel libre, distribué sous la licence GNU GPL.

gdb permet de déboguer un programme en cours d'exécution (en le déroulant instruction par instruction ou en examinant et modifiant ses données), mais il permet également un débogage post-mortem en analysant un fichier `core` qui représente le contenu d'un programme terminé anormalement.

L'interface de gdb est une simple ligne de commande, mais il est souvent invoqué en arrière-plan par les environnements de développement intégré.

Lien : https://www.gnu.org/software/gdb/

Exemple

Code source

```
$ vim exemple1.c
```

```
#include <stdlib.h>
#include <string.h>

int var1;
char var2[] = "buf1";

int main(int argc, char **argv)
{
    int var3 = 3;
    static int var4;
    static char var5[] = "buf2";
    char *var6;
    var6 = malloc(512);
    strcpy(var6, "hello");
    return 0;
}
```

Compilation

Les programmes C/C++ doivent être compilés avec l'option `-g` pour pouvoir être débuggés par `gdb`.

```
$ gcc -g exemple1.c

$ ls -l
-rwxr-xr-x  1 tv tv 9,6K juin   7 06:25 a.out
-rw-rw-r--  1 tv tv  265 juin   7 06:22 exemple1.c
```

Démarrage

```
$ gcc -g exemple1.c

$ gdb -q ./a.out
(gdb) help
(gdb) help run
...
```

- Voir le code source C/C++ :

```
(gdb) list
```

- Voir le code assembleur :

```
(gdb) disassemble main
(gdb) disassemble -r main
```

- Avec des arguments :

```
$ gdb -q --args ./a.out azertyuiop qsdfghjklm
```

- Quitter `gdb` :

```
(gdb) quit
```

Exécution

- Démarrer le débugage :

```
(gdb) start
(gdb) where
#0  main (argc=1, argv=0x7fffffffdec8) at exemple1.c:9
(gdb) backtrace
```

- Arrêter le débugage :

```
(gdb) kill
```

- Lancer le programme :

```
(gdb) run
[Inferior 1 (process 14852) exited normally]
```

- Ajouter un point d'arrêt sur une ligne :

```
(gdb) break 13
(gdb) run
```

- Exécuter en pas à pas :

```
(gdb) step

(gdb) next
```

- Continuer :

```
(gdb) continue
```

Point d'arrêt (*breakpoint*)

- Informations sur les points d'arrêt d'arrêt :

```
(gdb) info break
Num     Type           Disp Enb Address            What
3       breakpoint     keep y   0x0000555555554659 in main at exemple1.c:9
4       breakpoint     keep y   0x000055555555467e in main at exemple1.c:16
```

- Désactiver/Activer un point d'arrêt :

```
(gdb) disable breakpoints 5
(gdb) enable breakpoints 5
```

- Supprimer un point d'arrêt ligne :

```
(gdb) clear 9
(gdb) delete 4
```

Point de surveillance (*watchpoint*)

Un point de surveillance (*watchpoint*) arrête l'exécution du programme chaque fois que la valeur d'une expression change.

- Ajouter un point de surveillance :

```
(gdb) watch (var3 == 6)
```

Un point d'arrêt (*breakpoint*) permet d'arrêter l'exécution du programme à certains moments.

- Ajouter un point d'arrêt conditionnel :

```
(gdb) break 14 if var6 != 0
Breakpoint 6 at 0x55555555466e: file exemple1.c, line 14.
```

Les variables

- Afficher l'adresse d'une variable :

```
(gdb) print &var6
$1 = (char **) 0x7fffffffddd8
```

- Afficher une variable :

```
(gdb) print var6
$2 = 0x555555756260 "hello"
```

- Afficher une zone mémoire :

```
(gdb) print /x*0x5555575626O@2
$3 = {0x6c6c6568, 0x6f}
```

- Examiner une zone mémoire :

```
(gdb) x/2x 0x555555756260
0x555555756260: 0x6c6c6568      0x0000006f
(gdb) x/4b 0x555555756260
0x555555756260: 0x68 0x65 0x6c 0x6c
```

- Modifier une variable :

```
(gdb) print var6="autre"
$1 = 0x7ffff7fb8f10 "autre"

(gdb) print &var3
$6 = (int *) 0x7fffffffddd4
(gdb) print *0x7fffffffddd4=5
$7 = 5
(gdb) print var3
$8 = 5
```

- Afficher les variables locales :

```
(gdb) info locals
var3 = 5
var4 = 0
var6 = 0x555555756260 "autre"
```

- Afficher le type d'une variable :

```
(gdb) ptype var3
type = int
```

- Afficher des informations sur le stockage de la variable :

```
(gdb) info symbol var2
var2 in section .data of a.out
(gdb) info address var2
Symbol "var2" is static storage at address 0x555555755010.
```

Les arguments

```
$ gcc -g exemple1.c

$ gdb -q --args ./a.out azertyuiop qsdfghjklm
(gdb) break 13
(gdb) run
```

```
(gdb) info args
argc = 3
argv = 0x7fffffffde98

(gdb) print *argv@argc
$4 = {0x7fffffffe1fa "a.out", 0x7fffffffe22b "azertyuiop", 0x7fffffffe236 "qsdfghjklm"

(gdb) print &argv
$27 = (char ***) 0x7fffffffdd90

(gdb) print &argv[0]
$28 = (char **) 0x7fffffffde98
(gdb) print &argv[1]
$29 = (char **) 0x7fffffffdea0
(gdb) print &argv[2]
$30 = (char **) 0x7fffffffdea8

(gdb) print argv[0] // ou print *argv
$32 = 0x7fffffffe1fa "a.out"
(gdb) print argv[1] // ou print *(argv+1)
$33 = 0x7fffffffe22b "azertyuiop"
(gdb) print argv[2]
$34 = 0x7fffffffe236 "qsdfghjklm"
```

Les registres

```
(gdb) info registers
rax            0x555555756260   93824994337376
rbx            0x0      0
rcx            0x555555756260   93824994337376
rdx            0x555555756260   93824994337376
rsi            0x0      0
rdi            0x555555756460   93824994337888
rbp            0x7fffffffddb0   0x7fffffffddb0
rsp            0x7fffffffdd90   0x7fffffffdd90
r8             0x2      2
...
r15            0x0      0
rip            0x55555555466e   0x55555555466e <main+36>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0

(gdb) print $rsp
$1 = (void *) 0x7fffffffdde0

(gdb) print $rbp
$2 = (void *) 0x7fffffffdde0
```

```
(gdb) print $rip
$3 = (void (*)()) 0x555555554604 <foo+10>

(gdb) print $pc
$4 = (void (*)()) 0x555555554604 <foo+10>

(gdb) x/5i $pc-6
   0x5555555545fe <foo+4>:      mov    %edi,-0x14(%rbp)
   0x555555554601 <foo+7>:      mov    %esi,-0x18(%rbp)
=> 0x555555554604 <foo+10>:     mov    -0x14(%rbp),%eax
   0x555555554607 <foo+13>:     mov    %eax,-0xc(%rbp)
   0x55555555460a <foo+16>:     mov    -0x18(%rbp),%eax
```

Informations

```
(gdb) info stack
#0  main (argc=3, argv=0x7fffffffde98) at exemple1.c:14

(gdb) info frame
Stack level 0, frame at 0x7fffffffddc0:
 rip = 0x55555555466e in main (exemple1.c:14); saved rip = 0x7ffff7a05b97
 source language c.
 Arglist at 0x7fffffffddb0, args: argc=3, argv=0x7fffffffde98
 Locals at 0x7fffffffddb0, Previous frame's sp is 0x7fffffffddc0
 Saved registers:
  rbp at 0x7fffffffddb0, rip at 0x7fffffffddb8

(gdb) info program
        Using the running image of child process 21982.
Program stopped at 0x55555555466e.
It stopped at breakpoint 1.

(gdb) show endian
The target endianness is set automatically (currently little endian)
```

Core dump

Un *core dump* est une copie de la mémoire vive et des registres d'un processeur, permettant d'avoir un instantané de l'état d'un système sous forme d'un fichier.

Il sert généralement à des fins d'analyse, suite à une exception, forcée ou provoquée par une erreur. Le *core dump* doit être enregistré dans un fichier qui peut être utilisé ensuite dans un débogueur comme `gdb`.

> Actuellement, les système GNU/Linux désactive par défaut la génération des fichiers *core dump*.

Activation sous Ubuntu 18.04 :

```
$ ulimit -c 100000
$ sudo systemctl stop apport
```

Exemple :

```
$ vim exemple2.c
```

```
#include <string.h>

int main(int argc, char **argv)
{
    char buffer[12];
    if (argc > 1)
        strcpy(buffer, argv[1]);
    return 0;
}
```

Accrochez vos ceintures :

```
$ gcc -g exemple2.c

$ ./a.out azertyuiopqsdfghjklm
Abandon (core dumped)

$ ls -al
-rwxr-xr-x  1 tv tv   9464 juin   8 08:34 a.out
-rw-------  1 tv tv 253952 juin   8 08:47 core
```

Ouverture du fichier `core` avec `gdb` :

```
$ gdb -c core -q
Core was generated by `./a.out azertyuiopqsdfghjklm'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00007fbda40b2e97 in ?? ()
(gdb) print $rip
$1 = (void (*)()) 0x7fbda40b2e97
(gdb) print $rsp
$2 = (void *) 0x7ffd02934920
(gdb) info frame
Stack level 0, frame at 0x7ffd02934928:
 rip = 0x7fbda40b2e97; saved rip = 0x0
 called by frame at 0x7ffd02934930
 Arglist at 0x7ffd02934918, args:
 Locals at 0x7ffd02934918, Previous frame's sp is 0x7ffd02934928
 Saved registers:
  rip at 0x7ffd02934920
(gdb) backtrace
#0  0x00007fbda40b2e97 in ?? ()
```

Voir aussi

- http://tvaira.free.fr/bts-sn/poo-c++/tp-debug/Voyage-au-coeur-d-un-programme-executable-Episode-1.pdf

- http://tvaira.free.fr/bts-sn/poo-c++/tp-debug/Voyage-au-coeur-d-un-programme-executable-Episode-2.pdf
- http://tvaira.free.fr/bts-sn/poo-c++/tp-debug/Voyage-au-coeur-d-un-programme-executable-Episode-3.pdf

> L'épisode 3 décrit en détails l'exploitation d'un dépassement de tampon ou débordement de tampon (*buffer overflow*). Il montre l'utilisation de `gdb` dans une situation de *hacking*.
