1 – Playing in the stack :

En lançant le programme **stack** sur gdb, et en mettant un breakpoint sur f, on affiche la valeur de x, son adresse, ainsi que le pointeur *x qu'on comparera ensuite avec les valeurs contenues dans la pile en utilisant pframe.

x	**0xffffcd60**
&x	**0xffffcd50**
***x**	**2**

Observons désormais la pframe :

```
(gdb) pframe
0xffffcd88           0x00000000
0xffffcd84           0xf7f28000
0xffffcd80           0xf7f28000
0xffffcd7c           0xf7d68b41
0xffffcd78           0x00000000
0xffffcd74           0xffffcd90
0xffffcd70           0xf7fe4520
0xffffcd6c           0x00000001
0xffffcd68           0x080491a4
0xffffcd64           0xffffcd78
0xffffcd60           0x00000002
0xffffcd5c    ...    0x08049213
0xffffcd58    arg3   0x00000013
0xffffcd54    arg2   0x00040000
0xffffcd50    arg1   0xffffcd60
0xffffcd4c    ret@   0x08049187
0xffffcd48      bp sp 0xffffcd64
```

D'après la pile, on voit apparaître la valeur de x qui est un des arguments de la fonction f qui correspond à un pointeur de type int *. On voit aussi dans la pile que l'adresse de ce int * estt 0xffffcd50 ce qui correspond bien à la valeur que l'on avait eut en affichant &x.
Et si on regarde la valeur à l'adresse 0xffffcd60 on voit 2 qui correspond aussi à l'affichage que nous avions obtenu en déréférençant le pointeur x. (*x)

En utilisant **bt,** on affiche la trace des appels de fonctions effectué avec leurs paramètres avec un ordre commençant par l'appel de fonction le plus récent, la fonction f est montré en premier ensuite g () et en dernier main ().

En effet, on observant le programme **stack.c,** dans la fonction main () on fait appel a la fonction g() qui a son tour fait appel a f(), ceci justifie l'ordre des fonctions affiché par la **backtrace.**
Les arguments présents dans la pile correspondent aux paramètres de la fonction f().

```
(gdb) bt
#0  f (x=0xffffcd60) at stack.c:4
#1  0x08049187 in g (y=1) at stack.c:9
#2  0x080491a4 in main () at stack.c:13
```

L’adresse de retour présente dans la pile (**0x08049187**), correspond à l’adresse de retour dans la fonction **g ()** qu’on peut observer également dans la **backtrace** dans la 2 éme ligne de la fonction g().

Comme on se retrouve dans le contexte de la fonction f (), on ne peut pas accéder aux variables de la fonction g () tel que y et z. En utilisant, « **up** » on se met dans le contexte de la fonction g ().

```
(gdb) up
#1  0x08049187 in g (y=1) at stack.c:9
9               return f(&z);
(gdb) p z
$2 = 2
(gdb) p y
$3 = 1
(gdb) p x
No symbol "x" in current context.
```

Désormais, on peut afficher les variables y et z (ainsi que leurs adresses), qui sont des variables de la fonction g(), mais pas la variable x parce qu’elle n’existe pas dans le contexte actuel qui est celui de la fonction g().

y	1
&y	0xffffcd6c
z	2
&z	0xffffcd60

En regardant le code source, la variable y est un paramètre de la fonction g(), et z est une variable locale, vérifions cela sur la pile à l’aide de pframe dans le contexte de la fonction g ().

```
0xffffcd90           0x00000001
0xffffcd8c           0xf7d68b41
0xffffcd88           0x00000000
0xffffcd84           0xf7f28000
0xffffcd80           0xf7f28000
0xffffcd7c           0xf7d68b41
0xffffcd78           0x00000000
0xffffcd74    arg3   0xffffcd90
0xffffcd70    arg2   0xf7fe4520
0xffffcd6c    arg1   0x00000001
0xffffcd68    ret@   0x080491a4
0xffffcd64      bp   0xffffcd78
0xffffcd60           0x00000002
0xffffcd5c           0x08049213
0xffffcd58           0x00000013
0xffffcd54           0x00040000
0xffffcd50        sp 0xffffcd60
```

On voit que y dont la valeur est 1 avec pour adresse **0xffffcd6c**, est un des arguments de la fonction g(), en dessous du pointeur %**ebp** on voit la présence d’une valeur 2 qui correspond à la variable z qui est une variable locale de la fonction g(). (Dans l’affichage proposé par pframe on voit apparaître des arg2,3 etc alors que nous savons grâce au code source que g() ne prend qu’un seul paramètre, c’est par ce que pframe ne connaît pas le nombre d’arguments, il se contente de considérer comme argument toutes les valeurs au-dessus de l’adresse de retour mais il sait que 0xffffcd78 ne peut pas être l’adresse d’un paramètre car bp pointe vers 0xffffcd64 qui contient 0xffffcd78 la valeur

précédente de bp, ainsi 0xffffcd78 est censé contenir encore la valeur précdente de bp donc pas un paramètre)

On observe également, le pointeur **%esp** qui pointe sur 0xffffcd60 qui est l'adresse de la variable de z et l'argument de la fonction f (). l'adresse de retour affiché dans la pile (**0x080491a4**) dans le contexte de la fonction g(), est l'adresse de retour dans la fonction main () qui est également affiché dans la backtrace dans la ligne correspondante à la fonction de main ().
On peut se déplacer à travers l'empilement d'appels de fonctions grâce à up ou down, ou directement en identifiant le numero de la frame qui nous intéresse et en faisant frame 'x'. la commande **bt full,** permet d'afficher les appels de fonctions ainsi que les variables locales de chaque fonction.

```
(gdb) bt full
#0  f (x=0xffffcd60) at stack.c:4
No locals.
#1  0x08049187 in g (y=1) at stack.c:9
        z = 2
#2  0x080491a4 in main () at stack.c:13
No locals.
```

2- Damn optimizations :

À présent, on s'intéressera au même programme précédant (stack.c) mais cette fois-ci en étant optimisé.

En essayant d'afficher l'adresse de la variable z, en étant dans le contexte de la fonction g(). gdb nous dit qu'on ne peut afficher l'adresse de z parce qu'il n'est pas une valeur.

```
(gdb) p &z
Can't take address of "z" which isn't an lvalue.
(gdb)
```

Pour comprendre au mieux ce qui se passe, on désassemble le code de la fonction g () comme cidessous :

```
(gdb) disas
Dump of assembler code for function g:
   0x080491c0 <+0>:     push   %ebx
   0x080491c1 <+1>:     sub    $0x8,%esp
   0x080491c4 <+4>:     mov    0x10(%esp),%eax
   0x080491c8 <+8>:     add    $0x1,%eax
   0x080491cb <+11>:    push   %eax
   0x080491cc <+12>:    call   0x80491b0 <f>
=> 0x080491d1 <+17>:    sub    $0x4,%esp
   0x080491d4 <+20>:    push   %eax
   0x080491d5 <+21>:    mov    %eax,%ebx
   0x080491d7 <+23>:    push   $0x804a008
   0x080491dc <+28>:    call   0x8049030 <printf@plt>
   0x080491e1 <+33>:    add    $0x18,%esp
   0x080491e4 <+36>:    mov    %ebx,%eax
   0x080491e6 <+38>:    pop    %ebx
   0x080491e7 <+39>:    ret
End of assembler dump.
```

Compte tenu de l'optimisation faite par le compilateur, z ne possède pas d'adresse parce que d'après le désassemblage, la variable z est contenu dans le registre **%eax** (les registres n'ont pas d'adresse), qui a été ensuite empilé dans la pile comme argument pour f suivi du call de f().

Essayons à présent d'afficher la variable a ou son adresse, qui est une variable locale de la fonction g().

```
(gdb) p a
$1 = <optimized out>
(gdb) p &a
Can't take address of "a" which isn't an lvalue.
```

```
(gdb) bt full
#0  f (x=2) at optim.c:4
No locals.
#1  0x080491d1 in g (y=1) at optim.c:9
        z = 2
        a = <optimized out>
#2  0x08049068 in main () at optim.c:15
No locals.
```

gdb n'arrive pas a affiché la valeur de **'a'** ainsi que son adresse, pareil dans la backtrace full qui est censé afficher les variables locales de chaque fonction.

En effet, la variable **'a'** n'existe pas encore, comme on a effectué un break sur la fonction f (), on a pas encore atteint l'adresse de retour dans la fonction g().

D'ailleurs en regardant la suite du code assembleur de g, on voit que a aussi sera stocké dans eax donc n'aura pas d'adresse.

De plus, la variable **'a'** va être stocké dans le registre **%eax** puis empilé afin qu'elle puisse être utilisée comme argument de printf(), par la suite on sauvegarde**%eax** dans **%ebx** car **printf()** lors de retour elle va écraser **%eax** comme on peut le voir dans le code désassemblé ici :

```
   0x080491cc <+12>:    call   0x80491b0 <f>
=> 0x080491d1 <+17>:    sub    $0x4,%esp
   0x080491d4 <+20>:    push   %eax
   0x080491d5 <+21>:    mov    %eax,%ebx
   0x080491d7 <+23>:    push   $0x804a008
   0x080491dc <+28>:    call   0x8049030 <printf@plt>
   0x080491e1 <+33>:    add    $0x18,%esp
   0x080491e4 <+36>:    mov    %ebx,%eax
   0x080491e6 <+38>:    pop    %ebx
   0x080491e7 <+39>:    ret
End of assembler dump.
```

3 Watchpoints :

Dans cette partie, nous voudrions suivre les modifications effectuées sur une variable lors d'exécution d'un programme. Dans notre cas, c'est facile parce qu'on sait déjà que c'est la fonction f qui effectue des modifications.

En utilisant la commande **watch,** on met un watch point sur une variable qui permet de nous montrer à chaque fois que cette variable subit une modification au fil de l'exécution du programme.

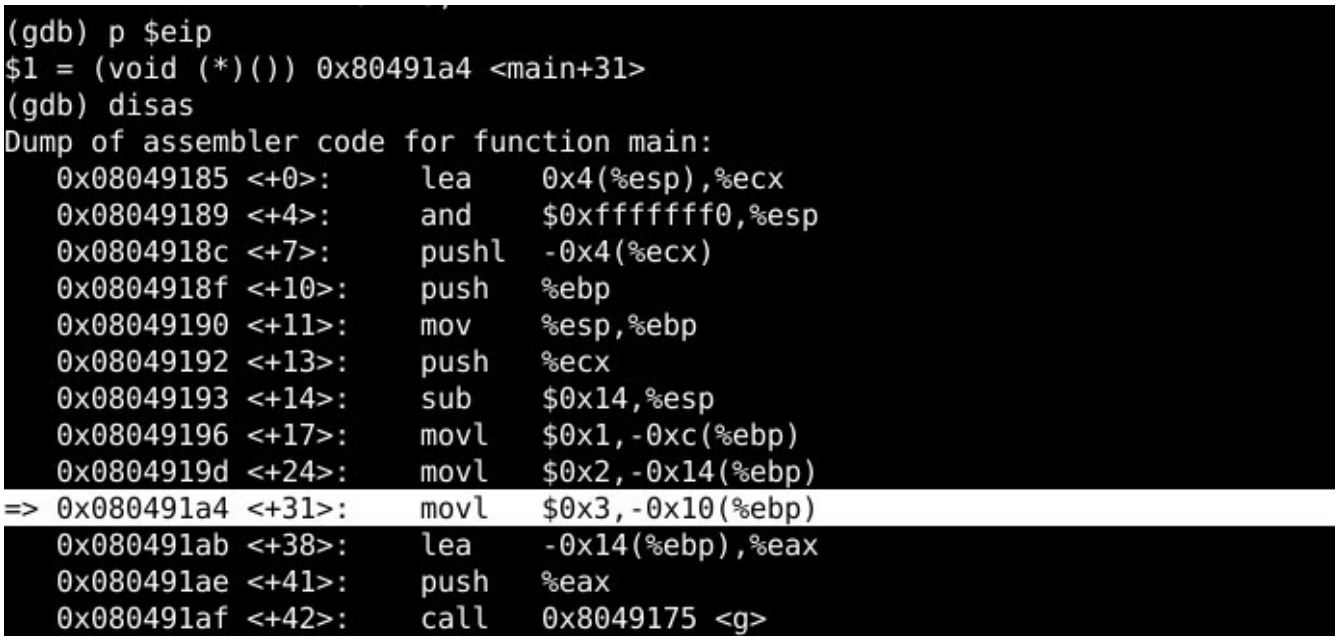
```
(gdb) wa a
Hardware watchpoint 2: a
```

Gdb nous informe qu'il s'agit d'un **hardware watchpoint,** comme la variable 'a' est sur seulement 4 octets (int), gdb était capable de prendre l'adresse de 'a' et dire au processeur de regarder pour cet emplacement mémoire dans le hardware. On exécute le programme a pleine vitesse et le processeur s'arrêtera au bon moment.

```
(gdb) c
Continuing.

Hardware watchpoint 2: a

Old value = -12764
New value = 2
main () at modif.c:14
```



On voit que le programme s'arrête dès que la variable a été initialisé. En effet, a avait une certaine valeur en mémoire avant son initiliasation donc le watch point considère cette initiliasation comme une modification de la valeur de a. De ce fait, gdb s'arrête juste après l'initialisation. En observant le pointeur EIP, on voit qu'il pointe sur l'instruction qui suit juste après initialisation de la variable 'a'.

Supposant qu'on connaît pas le nom d'une variable mais seulement son adresse, on peut utiliser **watch *(int *) adresse de la variable** pour traquer les modifications que subit cette variable. Cela peut être utile lorsqu'on travaille sur un binaire dont nous n'avons accès au code source et donc nous ne connaissons pas le nom de symboles présents.

4 Real debugging witch watchpoints :

Dans cette partie, nous utiliserons les watchpoints pour debugger. En effet, on dispose d'un programme **modif2.c** qui apparemment un des entiers initialisé dans le main () se fait écraser quelques parts, nous souhaitons savoir comment cela a été fait sans regarder le code source des deux fonctions f() et g() qui interviennent dans le programme.

En commençant par mettre des watch sur les deux entiers, on continue l'exécution jusqu'à ce que l'un des deux entiers subissent une modification (Hors initialisation).

```
Hardware watchpoint 3: c

Old value = 1234567890
New value = 1234567680
__memset_sse2_rep () at ../sysdeps/i386/i686/multiarch/memset-sse2-rep.S:184
```

```
Dump of assembler code for function f:
   0x08049172 <+0>:     push   %ebp
   0x08049173 <+1>:     mov    %esp,%ebp
   0x08049175 <+3>:     sub    $0x8,%esp
   0x08049178 <+6>:     mov    0xc(%ebp),%eax
   0x0804917b <+9>:     sub    $0x4,%esp
   0x0804917e <+12>:    push   %eax
   0x0804917f <+13>:    push   $0x0
   0x08049181 <+15>:    pushl  0x8(%ebp)
   0x08049184 <+18>:    call   0x8049050 <memset@plt>
=> 0x08049189 <+23>:    add    $0x10,%esp
   0x0804918c <+26>:    nop
   0x0804918d <+27>:    leave
   0x0804918e <+28>:    ret
End of assembler dump.
```

On voit que la valeur de ‘c’ a été modifié au milieu de l’implémentation de **memset,** en effectuant un désassemblage du code de la fonction f(), on voit que le pointeur **eip** pointe sur l’instruction qui suit juste après l’appel a la fonction **memset**.

En effectuant un breakpoint sur l’appel a la fonction **memset** et on observant la pile, on a affiché les paramètres passés a **memset** pour les comparer ensuite avec les adresses de a,b et c sachant que **memset** prend 3 paramètres (void *s, int c, size_t n).
En effet si on désassemble le code on voit les 3 arguments de memset être empilé :

```
Dump of assembler code for function f:
   0x08049172 <+0>:     push   %ebp
   0x08049173 <+1>:     mov    %esp,%ebp
   0x08049175 <+3>:     sub    $0x8,%esp
   0x08049178 <+6>:     mov    0xc(%ebp),%eax
   0x0804917b <+9>:     sub    $0x4,%esp
   0x0804917e <+12>:    push   %eax
   0x0804917f <+13>:    push   $0x0
   0x08049181 <+15>:    pushl  0x8(%ebp)
=> 0x08049184 <+18>:    call   0x8049050 <memset@plt>
```

Que l’on fait correspondre avec l’affichage de la pile de f :

```
0xffffcd08                0x0000000d
0xffffcd04                0x00000000
0xffffcd00              sp 0xffffcd5c
```

```
(gdb) p &a
$1 = (char (*)[12]) 0xffffcd5c
(gdb) p &b
$2 = (int *) 0xffffcd6c
(gdb) p &c
$3 = (int *) 0xffffcd68
(gdb)
```

on voit que les arguments de la fonction **memset** sont **:**
premier argument :0xffffcd5c qui correspond a l'adresse de la variable 'a', deuxieme argument:0 qui signifie que memset remplira de 0 les n (troisième argument) premiers octets de l'espace mémoire pointé par le premier argument. Troisième argument :0x0000000d qui vaut 13.

Essayons à présent de comprendre ce qui se passe : **memset** va remplir les 13 premiers octets de la zone mémoire pointé par x avec l'octet **'\0'** mais on sait que 'a' est un tableau de caractères qui contient seulement 12 éléments, donc on va écraser ce qu'il y a vraiment a l'adresse de **a[12]** son adresse est donc l'adresse de a +0xc.

En regardant sur gdb a quoi correspond a[12] :

```
(gdb) p (char*) (&a)+0xc
$10 = 0xffffcd68 "\322\002\226I\322\002\226I E\376\367\220\315\377\377"
(gdb)
```

(le cast en char * n'était pas utile ici, c'est seulement la valeur
de &a+0xc qui nous interresse)

Et c'est exactement la même que celle de nombre variable c.
On comprend maintenant pourquoi c a été écrasé.

Si nous regardons le code source, on voit que l'erreur vient du fait que dans le main le deuxième argument passé a g() est sizeof(a)+1(13) au lieu de sizeof(a). Et dans f l'appel a memset sur l'espace mémoiré pointé par a, '\0' correspondant a la valeur 0 en entier et ce sizeof(a)+1. Ce qui explique avec ce qu'on a dit précédemment que c se fait écraser.

5 Combining valgrind and gdb :

En exécutant le programme **modif3,** aucun problème n'est signalé. Il nous renvoie une adresse qui correspond d'après le code source a l'adresse de la chaîne de caractères 'a'.

```
hbenarab@bodon:~/secu/TD8$ ./modif3
0xffd3546c
```

Essayons à présent de l'exécuter en utilisant **Valgrind :**

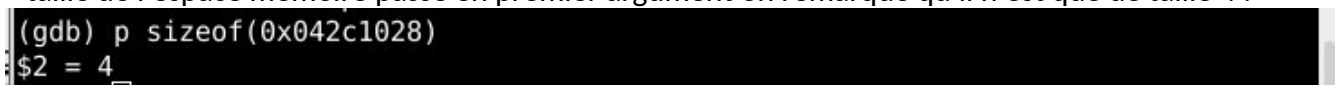
```
hbenarab@bodon:~/secu/TD8$ valgrind --leak-check=full --show-leak-kinds=all ./modif3
==618625== Memcheck, a memory error detector
==618625== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==618625== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==618625== Command: ./modif3
==618625==
==618625== Invalid write of size 1
==618625==    at 0x403B970: memset (vg_replace_strmem.c:1251)
==618625==    by 0x80491A8: f (modif3.c:6)
==618625==    by 0x80491C2: g (modif3.c:10)
==618625==    by 0x80491F9: main (modif3.c:15)
==618625==  Address 0x42c102c is 0 bytes after a block of size 4 alloc'd
==618625==    at 0x403463B: malloc (vg_replace_malloc.c:299)
==618625==    by 0x4162AA5: strdup (strdup.c:42)
==618625==    by 0x80491E5: main (modif3.c:14)
==618625==
0xfeb9fd0c
==618625==
==618625== HEAP SUMMARY:
==618625==     in use at exit: 0 bytes in 0 blocks
==618625==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==618625==
==618625== All heap blocks were freed -- no leaks are possible
==618625==
==618625== For counts of detected and suppressed errors, rerun with: -v
==618625== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind nous informe qu'il y'a une écriture invalide au niveau de la fonction **memset()** et qu'on essaie d'écrire à une adresse qui est en dehors de la zone mémoire alloué par **strdup ()** qui en effet, alloue une nouvelle zone de mémoire via la fonction **malloc()** afin d'y copier la chaîne de caractères initiale et s'il y a suffisament de mémoire pour produire la nouvelle chaîne, la fonction renvoie l'adresse de la chaîne dupliquée.

Ce n'est pas facile de trouver ce qui ne va pas seulement avec les résultats de valgrind, on voudra avoir accès a toutes les adresses afin d'y voir mieux, pour cela on combine gdb avec valgrind.
En faisant cela, gdb s'est arrête au milieu de la fonction memset() la ou l'accès invalide en mémoire a été causé. En utilisant backtrace, on voit que l'erreur à lieu dans memset qui a été appelé par la fonction f elle-même appelée par g et cette dernière appelée dans main.

En procèdant de la même manière que dans le 4), on remarque que dans la fonction f qui appelle memset, les paramètres fournis sont :

On a donc passé en 3ᵉ paramètre la valeur 5, memset va donc effectuer 5 écritures. Or si on affiche la taille de l'espace mémoire passé en premier argument on remarque qu'il n'est que de taille 4 :



On comprend donc maintenant pourquoi valgrind nous indiquait un accès mémoire interdit « 0 byte after a size of 4 allocated bloc »

En regardant le code source on voit que le problème vient du main :

```c
int main(void) {
    char *a = strdup("aaa");
    g(a, sizeof(a)+1);
    printf("%p\n", &a);
    free(a);
    return 0;
}
```

On passe à g, la taille de a +1 qui vaut 5, ce qui induit cette écriture invalide en mémoire lors de l'appel a memset. En effet, on lui demande d'écrire 5 '/0' dans un espace mémoire alloué de taille 4.