GDB Tutorial 2

In the previous Tutorial (GDB Tutorial-I)we looked into basic environment and commands of gnu source debugger gdb, we have also looked into how gnu debugger works , lets now run some debug sessions using sample buggy programs

Case 1:

let's consider the following sample source program , compile and run

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

    int display(char *dup)
    {
            char p[30];

        memcpy(p, "This is a test", 14);
        printf("\nP = %s", p);
        printf("\nDup = %s\n", dup);
        return 0;
    }

    char *mystrndup(char *str, int n)
    {
        char buf[30];
        char *p = buf;
        memcpy(p, str, n);
        return p;
    }
    int main(int argc, char **argv)
    {
        char *dup;
        if (argc < 2) {
            printf("\nUsage: strndup <string>\n");
            return 0;
        }
        dup = mystrndup(argv[1], strlen(argv[1]));
        display(dup);
```

```
        free(dup);
    }
raghu@techveda:~$ ./strndup veda
P = This is a test
Dup = This is a test
Segmentation fault
raghu@techveda:~$
```

program terminated with segmentation fault, let's use gdb and find out what happened.

```
raghu@techveda:~$ gdb strndup

(gdb) run veda
Starting program: /home/raghu/strndup veda

P = This is a test
Dup = This is a test

Program received signal SIGSEGV, Segmentation fault.
0x0019f3f9 in free () from /lib/i386-linux-gnu/libc.so.6
(gdb)
```

The above output says that there is a segmentation fault in the free() function at 0x0019f3f9 instruction address. And both P and Dup point to the same memory. Run the program step by step and see what happened.

```
(gdb) break main
Breakpoint 1 at 0x804859b: file strndup.c, line 26.
(gdb) r veda
Starting program: /home/raghu/strndup veda

Breakpoint 1, main (argc=2, argv=0xbffff444) at strndup.c:26
26                      if (argc < 2) {
(gdb) s
0x001a2d30 in strlen () from /lib/i386-linux-gnu/libc.so.6
(gdb)
Single stepping until exit from function strlen,
which has no line number information.
mystrndup (str=0xbffff5e2 "veda", n=4) at strndup.c:16
16              {
(gdb) s
19                      memcpy(p, str, n);
(gdb) print p
$1 = 0xbffff33e ""
```

```
(gdb)
```

First this is to understand why 'dup' and the 'p' print same string . For this we can print both addresses and see them. In the above output we have printed the output of p in the mystrndup() function which prints the 0xbffff33e.

```
(gdb) s
0x001a4520 in memcpy () from /lib/i386-linux-gnu/libc.so.6
(gdb)
Single stepping until exit from function memcpy,
which has no line number information.
mystrndup (str=0xbffff5e2 "veda", n=4) at strndup.c:20
20                              return p;
(gdb)
21              }
(gdb)
main (argc=2, argv=0xbffff444) at strndup.c:31
31                              display(dup);
(gdb)
display (dup=0xbffff33e "veda\023") at strndup.c:6
6               {
(gdb)
9                               memcpy(p, "This is a test", 14);
(gdb)
```

now print the address of 'p' in the display() function.

```
(gdb) print p
$2 =
"veda\023\000\364\277(\000\000\000\000\000\000\000\000\000\230\363\377\277@<\022\
000\002\000\000"
(gdb) print &p
$3 = (char (*)[30]) 0xbffff33e
(gdb)
```

from the above we can observe that both 'dup' and 'p' in the 'display()' point to the same memory address. lets find which segment the address falls into. As the p is a char array allocated in the stack, the address should belong to stack segment. This we can confirm by looking into the /proc/$pid/maps. Like this:

```
(gdb) info proc
process 2744
cmdline = '/home/raghu/strndup'
cwd = '/home/raghu'
```

```
exe = '/home/raghu/strndup'
(gdb)
```

info proc can be used to get the pid of the program. Here it is 2744. let's look into process memory map.

```
(gdb)shell cat /proc/2744/maps
00110000-0012c000 r-xp 00000000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
0012c000-0012d000 r--p 0001b000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
0012d000-0012e000 rw-p 0001c000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
0012e000-0012f000 r-xp 00000000 00:00 0          [vdso]
0012f000-00289000 r-xp 00000000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
00289000-0028a000 ---p 0015a000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028a000-0028c000 r--p 0015a000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028c000-0028d000 rw-p 0015c000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028d000-00290000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 08:01 973089     /home/raghu/strndup
08049000-0804a000 r--p 00000000 08:01 973089     /home/raghu/strndup
0804a000-0804b000 rw-p 00001000 08:01 973089     /home/raghu/strndup
b7fec000-b7fed000 rw-p 00000000 00:00 0
b7ffe000-b8000000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
(gdb)
```

above dump shows that address falls into the stack address range. With this we can clearly conclude that the address pointed by 'dup' is in stack frame of mystrndup() function. After 'mystrndup() 'returned, 'display()' function was called, which had overwritten the stack frame of the mystrndup(), and accidentally char array 'p' of 'display()' was pointing to same address since it was not explicitly initialized. last instruction of the main() function, we are calling free() to free the memory in the dup, which should not happen. Stack memory cannot be freed using the free() function, and will cause segmentation fault.

Case 2

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

    const char mem[30] = {"Hello, World"};

    int display(char *dup)
    {

        char *p = (char *)mem;
```

```
        memcpy(p, "This is a test", 14);
        printf("\nP = %s, Dup = %s", p, dup);
        return 0;
}

char *mystrndup(char *str, int n)
{
        char *p = malloc(n);
        memcpy(p, str, n);
        return p;
}
int main(int argc, char **argv)
{
        char *dup;

        if (argc < 2) {
                printf("\nUsage: strndup <string>\n");
                return 0;
        }
        dup = mystrndup(argv[1], strlen(argv[1]));
        display(dup);
        free(dup);
}
```

compile and run it.

```
raghu@techveda:~$ gcc -g strndup.c -o strndup
raghu@techveda:~$ ./strndup veda
Segmentation fault
raghu@techveda:~$
```

program crashed with segmentation fault, lets get into gdb session

```
raghu@techveda:~$ gdb ./strndup
(gdb) run veda
Starting program: /home/raghu/strndup veda

Program received signal SIGSEGV, Segmentation fault.
0x00244713 in ?? () from /lib/i386-linux-gnu/libc.so.6
(gdb)
```

above dump reports that fault occured in libc.so. At address '0x00244713' . let's inquire into details.

```
(gdb) break main
Breakpoint 1 at 0x8048539: file strndup.c, line 27.
(gdb) r veda
Starting program: /home/raghu/strndup veda

Breakpoint 1, main (argc=2, argv=0xbffff444) at strndup.c:27
27                              if (argc < 2) {
(gdb) s
31                              dup = mystrndup(argv[1], strlen(argv[1]));
(gdb)
0x001a2d30 in strlen () from /lib/i386-linux-gnu/libc.so.6
(gdb)
Single stepping until exit from function strlen,
which has no line number information.
mystrndup (str=0xbffff5e2 "veda", n=4) at strndup.c:19
19                              char *p = malloc(n);
(gdb)
20                              memcpy(p, str, n);
(gdb)
0x001a4520 in memcpy () from /lib/i386-linux-gnu/libc.so.6
(gdb)
Single stepping until exit from function memcpy,
which has no line number information.
mystrndup (str=0xbffff5e2 "veda", n=4) at strndup.c:21
21                              return p;
(gdb)
22                      }
(gdb)
main (argc=2, argv=0xbffff444) at strndup.c:32
32                              display(dup);
(gdb)
display (dup=0x804b008 "veda") at strndup.c:10
10                              char *p = (char *)mem;
(gdb)
Program received signal SIGSEGV, Segmentation fault.
0x00244713 in ?? () from /lib/i386-linux-gnu/libc.so.6
(gdb)
```

The problem seems to be in 'display()' function after the 'char *p = (char *)mem' instruction. There we have call to 'memcpy()' function (defined in the libc.so.6.) Look's like libc bug, before suspecting libc, let's verify if we are passing valid arguments 'memcpy()' .

```
(gdb) print p
No symbol "p" in current context.
(gdb)
```

Here it says that we are not in the context or stack frame where the p is defined. We are in the memcpy function stack frame. To print the p value we need to go to the previous stack frame which is of display() function.

```
(gdb) info stack
#0  0x00244713 in ?? () from /lib/i386-linux-gnu/libc.so.6
#1  0x080484dc in display (dup=0x804b008 "veda") at strndup.c:12
#2  0x08048588 in main (argc=2, argv=0xbffff444) at strndup.c:32
(gdb)
```

The above output shows the three stack frames. The 0th one is of the memcpy() function and the 1st one is the stack frame of display() function and the 2nd one is the main() function stack frame. let's swith to 'display()' stack frame.

```
(gdb) select-frame 1
(gdb) info frame
Stack level 1, frame at 0xbffff370:
 eip = 0x80484dc in display (strndup.c:12); saved eip 0x8048588
 called by frame at 0xbffff3a0, caller of frame at 0xbffff340
 source language c.
 Arglist at 0xbffff368, args: dup=0x804b008 "veda"
 Locals at 0xbffff368, Previous frame's sp is 0xbffff370
 Saved registers:
  ebp at 0xbffff368, eip at 0xbffff36c
(gdb) print p
$2 = 0x8048660 "Hello, World"
(gdb)
```

We are able to print the p value and it contains an address '0x8048660' which should contain “hello, World”. let's verify if this address is valid

```
(gdb) info proc
process 3019
cmdline = '/home/raghu/strndup'
cwd = '/home/raghu'
exe = '/home/raghu/strndup'
(gdb) shell cat /proc/3019/maps
00110000-0012c000 r-xp 00000000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
0012c000-0012d000 r--p 0001b000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
0012d000-0012e000 rw-p 0001c000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
```

```
0012e000-0012f000 r-xp 00000000 00:00 0          [vdso]
0012f000-00289000 r-xp 00000000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
00289000-0028a000 ---p 0015a000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028a000-0028c000 r--p 0015a000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028c000-0028d000 rw-p 0015c000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028d000-00290000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 08:01 973136     /home/raghu/strndup
08049000-0804a000 r--p 00000000 08:01 973136     /home/raghu/strndup
0804a000-0804b000 rw-p 00001000 08:01 973136     /home/raghu/strndup
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7fec000-b7fed000 rw-p 00000000 00:00 0
b7ffe000-b8000000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
(gdb)
```

Address we are looking is in '**08048000-08049000 r-xp 00000000 08:01 973136 /home/raghu/strndup '** region and permissions are read only and write access triggers segmentation fault.

Case 3

```
struct foo
{
   int cat;
   int * dog;
};

void bar (void * arg)
{
   printf("o hello bar\n");
   struct foo * food = (struct foo *) arg;
   printf("cat meows %i\n", food->cat);
   printf("dog barks %i\n", *(food->dog));
}
void main()
{
   int cat = 4;
   int * dog;
   dog = &cat;

   printf("cat meows %i\n", cat);
   printf("dog barks %i\n", *dog);

   struct foo * food;
```

```
   food->cat = cat;
   food->dog = dog;

   printf("cat meows %i\n", food->cat);
   printf("dog barks %i\n", *(food->dog));

   printf("time for foo!\n");
   bar(food);
   printf("begone!\n");

   cat = 5;
   printf("cat meows %i\n", cat);
   printf("dog barks %i\n", *dog);

//  return 0;
}
raghu@techveda:~$ gcc -g gdbex1.c -o gdbex1
raghu@techveda:~$./gdbex1
cat meows 4
dog barks 4
Segmentation fault
```

segmentation fault reported. let's find the instruction that cause fault using gdb.

```
$gdb ./gdbex1
(gdb) run
Starting program: /home/raghu/gdbex1
cat meows 4
dog barks 4

Program received signal SIGSEGV, Segmentation fault.
0x08048486 in main () at gdbexa1.c:24
24              food->cat = cat;
```

above output shows segmentation fault occured at 0x08048486 in main() function. It is also showing exact code where the segmentation fault happens, which is the instruction food0>cat = cat. lets use 'list' command to find the exact line in the source code that caused fault .

```
(gdb) list *(0x08048486)
0x8048486 is in main (gdbexa1.c:24).
19
20              printf("cat meows %i\n", cat);
21              printf("dog barks %i\n", *dog);
22
```

```
23          struct foo * food;
24          food->cat = cat;
25          food->dog = dog;
26
27          printf("cat meows %i\n", food->cat);
28          printf("dog barks %i\n", *(food->dog));
```

Here in this line it shows the instruction address 0x8048486 points to line number 24 of gdbex1.c. Here source code is given with line numbers. The 24 line number contains food->cat = cat. We are assigning food->cat to cat which is a local variable. Food is a pointer to struct foo. What we have to see in this case is, whether food pointer variable contains the valid memory or not.

```
(gdb) print food
$3 = (struct foo *) 0x804852b
```

It looks valid address. Let us print value in the 0x804852b address. We will only print one byte of memory.

```
(gdb) print *((unsigned char *)0x804852b)
                $9 = 129 '\201'
```

we have type-casted address to unsigned char pointer and printed the value at the address using print command. Here the we have 129 in the byte. Let us now access this using the struct variable itself.

```
(gdb) print food->cat
$1 = 449430401
(gdb)
```

This will print the value in the food->cat. You can even use the address to print the structure variables like this:

```
(gdb) print food
        $2 = (struct foo *) 0x804852b
(gdb) print ((struct foo *)0x804852b)->cat
        $3 = 449430401
(gdb) print ((struct foo *)0x804852b)->dog
    $4 = (int *) 0xec830000
(gdb)
```

We are able to access the memory at food. But what happened, why the segmentation fault ?lets verify if address is valid

```
(gdb) info proc
process 1862
cmdline = '/home/raghu/gdbex1'
cwd = '/home/raghu'
exe = '/home/raghu/gdbex1'
(gdb) shell cat /proc/1862/maps
00110000-0012c000 r-xp 00000000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
0012c000-0012d000 r--p 0001b000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
0012d000-0012e000 rw-p 0001c000 08:01 262454     /lib/i386-linux-gnu/ld-2.13.so
0012e000-0012f000 r-xp 00000000 00:00 0          [vdso]
0012f000-00289000 r-xp 00000000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
00289000-0028a000 ---p 0015a000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028a000-0028c000 r--p 0015a000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028c000-0028d000 rw-p 0015c000 08:01 262467     /lib/i386-linux-gnu/libc-2.13.so
0028d000-00290000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 08:01 973519     /home/raghu/gdbex1
08049000-0804a000 r--p 00000000 08:01 973519     /home/raghu/gdbex1
0804a000-0804b000 rw-p 00001000 08:01 973519     /home/raghu/gdbex1
b7fec000-b7fed000 rw-p 00000000 00:00 0
b7ffd000-b8000000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
(gdb)
```

The address fall into **08048000-08049000 r-xp 00000000 08:01 973519 /home/raghu/gdbex1** line in this output. permissions indicate that it is read only memory. This we have seen in the previous examples also. But we need to know why this memory is read only. Right?. For this we need to see the sections of the program. To see the sections of a program, we can use info files command, like this:

```
(gdb) info files
Symbols from "/home/raghu/gdbex1".
Unix child process:
        Using the running image of child process 2724.
        While running this, GDB does not access memory from...
Local exec file:
        `/home/raghu/gdbex1', file type elf32-i386.
        Entry point: 0x8048340
        0x08048134 - 0x08048147 is .interp
        0x08048148 - 0x08048168 is .note.ABI-tag
        0x08048168 - 0x0804818c is .note.gnu.build-id
        0x0804818c - 0x080481ac is .gnu.hash
        0x080481ac - 0x0804820c is .dynsym
        0x0804820c - 0x0804825d is .dynstr
        0x0804825e - 0x0804826a is .gnu.version
```

```
0x0804826c - 0x0804828c is .gnu.version_r
0x0804828c - 0x08048294 is .rel.dyn
0x08048294 - 0x080482b4 is .rel.plt
0x080482b4 - 0x080482e4 is .init
0x080482e4 - 0x08048334 is .plt
0x08048340 - 0x080485bc is .text
0x080485bc - 0x080485d8 is .fini
0x080485d8 - 0x0804861e is .rodata
0x08048620 - 0x08048624 is .eh_frame
0x08049f14 - 0x08049f1c is .ctors
0x08049f1c - 0x08049f24 is .dtors
0x08049f24 - 0x08049f28 is .jcr
0x08049f28 - 0x08049ff0 is .dynamic
0x08049ff0 - 0x08049ff4 is .got
0x08049ff4 - 0x0804a010 is .got.plt
0x0804a010 - 0x0804a018 is .data
0x0804a018 - 0x0804a020 is .bss
0x00110114 - 0x00110138 is .note.gnu.build-id in /lib/ld-linux.so.2
0x00110138 - 0x001101f4 is .hash in /lib/ld-linux.so.2
0x001101f4 - 0x001102d4 is .gnu.hash in /lib/ld-linux.so.2
0x001102d4 - 0x00110494 is .dynsym in /lib/ld-linux.so.2
```

above command dumps all sections of all objects files of the executable binary(sections of the program binary and sections of the libraries that are loaded along with the program) You can see here all the sections with their address ranges.Address 0x0804852b belongs to .text section of the program gdbex1. As you know text section of a program is always read only, you cannot write into it. That means the struct foo pointer variable food doesn't contain the valid address. Here in the program, food is declared but not initialized. IF you do not initialize a variable, a garbage value will be in the memory. With this example we have learnt how to access structure variables in gdb and how to list the sections of a program and that of the libraries that are loaded along with the program.