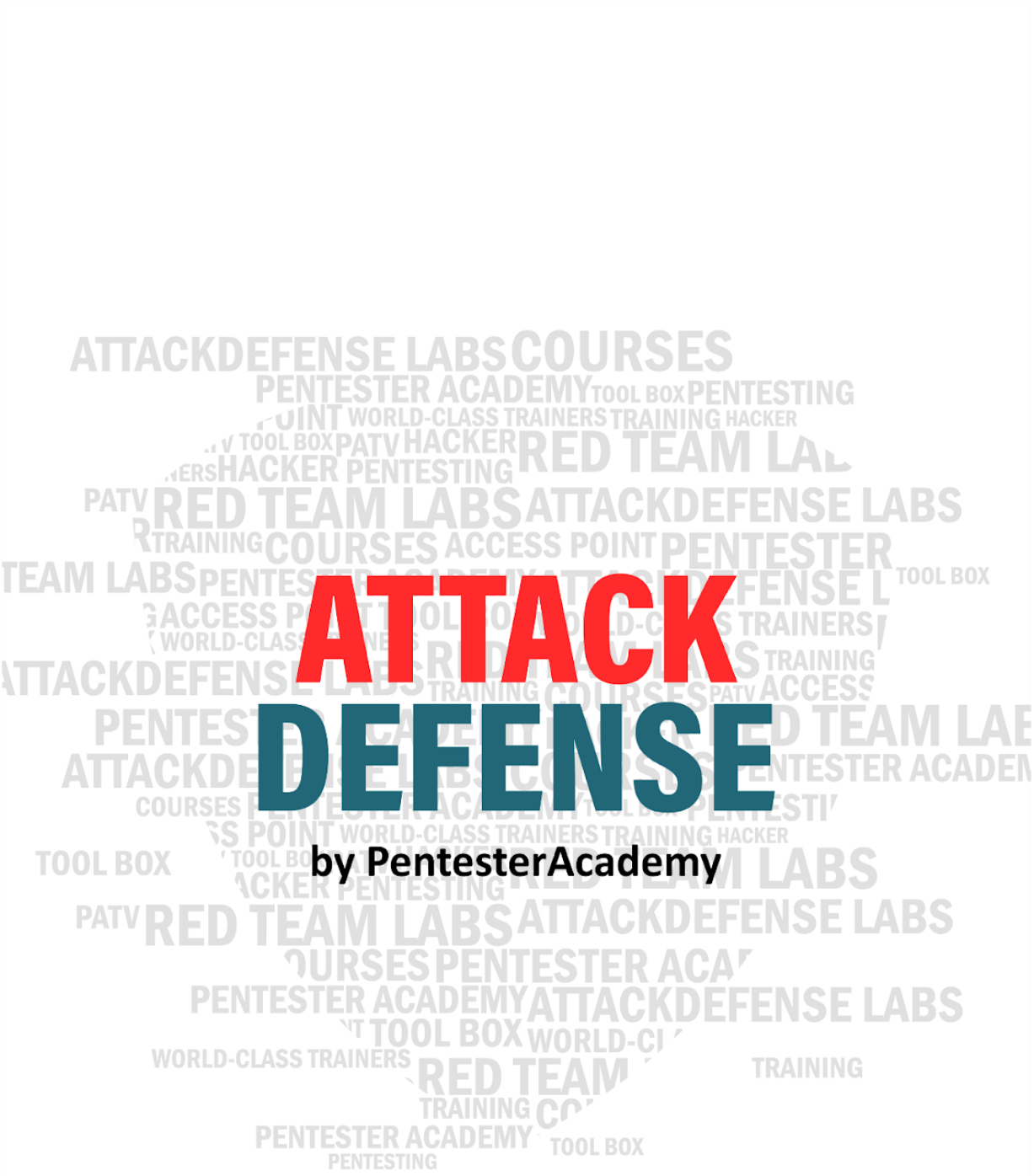
ATTACK
DEFENSE
by PentesterAcademy

Name	Examining the Stack
URL	https://www.attackdefense.com/challengedetails?cid=2168
Type	Reverse Engineering : GDB Basics

Important Note: This document illustrates all the important steps required to complete this lab. This is by no means a comprehensive step-by-step solution for this exercise. This is only provided as a reference to various commands needed to complete this exercise and for your further research on this topic. Also, note that the IP addresses and domain names might be different in your lab.

Objective: Learn how to examine the stack in GDB and check out different commands/options/methods.

Solution:

Stack frame is a block of data which contains the location of the function call in the program, the arguments of the call, and the local variables of the function. This is different for each function call.

Call stack is the allocated memory to stack frames.

Stack frames can be examined when the program execution is stopped.

Initial (or Outermost) frame is the stack frame of the main() function. This is the only frame in the beginning of program execution.

Innermost frame is the present frame in which execution is actually happening.

Frame pointer register contains the address of the frame whose call is under execution.

Frame level (or number) is the number assigned to the frame by the GDB. The level is 0 for the innermost frame, 1 for the frame that called it, and so on upward.

Step 1: Open sample2 binary using gdb.

Command: gdb -q sample2

```
root@localhost:~# gdb -q sample2
Reading symbols from sample2...
```

Step 2: Check the source code in steps.

Command: l (Small case L)

```
(gdb) l
4       #include<unistd.h>
5       #include <pthread.h>
6
7       int sum_f(int x,int y){
8               return x+y;
9       }
10
11      int sum_func(int num1, int num2) {
12              int tsum=0;
13              tsum = sum_f(num1,num2);
```

```
(gdb) l
14              return tsum;
15      }
16
17      int main(int argc, char * argv[]) {
18              int a=0, b=0, result=0;
19
20              if (argc != 3) {
21                      printf("WRONG Params!! \n\n ./sample1 <num1> <num2> \n");
22                      exit(1);
23              }
```

```
(gdb) l
24
25              a = atoi(argv[1]);
26              b = atoi(argv[2]);
27
28              printf("Both numbers accepted for addition. \n");
29
30              result = sum_func(a,b);
31
32              printf("Sum is : %d \n", result);
33
```

Step 3: Add a breakpoint at line 8 and run the program.

Command: start 2 3

```
(gdb) break 8
Breakpoint 1 at 0x724: file sample2.c, line 8.
(gdb) run 2 3
Starting program: /root/sample2 2 3
Both numbers accepted for addition.

Breakpoint 1, sum_f (x=2, y=3) at sample2.c:8
8               return x+y;
```

Step 4: Check the full backtrace for the current program.

Command: bt

```
(gdb) bt
#0  sum_f (x=2, y=3) at sample2.c:8
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
#2  0x00005555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
```

One can also use full backtrace command

Command: backtrace

```
(gdb) backtrace
#0  sum_f (x=2, y=3) at sample2.c:8
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
#2  0x00005555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
```

Step 5: Check the outermost (least recent) one frame in the backtrace.

Command: bt -1

```
(gdb) bt -1
#2  0x00005555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
```

Step 6: Check the innermost (most recent) one frame in the backtrace.

Command: bt 1

```
(gdb) bt 1
#0  sum_f (x=2, y=3) at sample2.c:8
(More stack frames follow...)
```

Step 7: Check the innermost (most recent) two frames in the backtrace.

Command: bt 2

```
(gdb) bt 2
#0  sum_f (x=2, y=3) at sample2.c:8
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
(More stack frames follow...)
```

Step 8: Check all details for all the frames in the backtrace including arguments and local variables.

Command: bt -full

```
(gdb) bt -full
#0  sum_f (x=2, y=3) at sample2.c:8
No locals.
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
        tsum = 0
#2  0x00005555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
        a = 2
        b = 3
        result = 0
```

Step 8: Select frame 1.

Command: frame 1

```
(gdb) frame 1
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
13              tsum = sum_f(num1,num2);
```

Alternatively. Command “select-frame <frame_number>” can be used. It is just like the “frame” command but doesn’t print the information of the switched frame.

Step 9: Show information about the currently selected frame.

Command: frame

```
(gdb) frame
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
13              tsum = sum_f(num1,num2);
```

Step 10: Check the details of the currently selected frame in the backtrace.

Command: info frame

```
(gdb) info frame
Stack level 1, frame at 0x7fffffffe4e0:
 rip = 0x555555554752 in sum_func (sample2.c:13); saved rip = 0x5555555547e1
 called by frame at 0x7fffffffe510, caller of frame at 0x7fffffffe4b8
 source language c.
 Arglist at 0x7fffffffe4d0, args: num1=2, num2=3
 Locals at 0x7fffffffe4d0, Previous frame's sp is 0x7fffffffe4e0
 Saved registers:
  rbp at 0x7fffffffe4d0, rip at 0x7fffffffe4d8
```

All details of the frame like address, arguments, local variables and register values are shown.

Step 11: Check the details of frame 2 in the backtrace.

Command: info frame 2

```
(gdb) info frame 2
Stack frame at 0x7fffffffe510:
 rip = 0x5555555547e1 in main (sample2.c:30); saved rip = 0x7ffff7df90b3
 caller of frame at 0x7fffffffe4e0
 source language c.
 Arglist at 0x7fffffffe500, args: argc=3, argv=0x7fffffffe5f8
 Locals at 0x7fffffffe500, Previous frame's sp is 0x7fffffffe510
 Saved registers:
  rbp at 0x7fffffffe500, rip at 0x7fffffffe508
```

Step 12: Check the arguments of the currently selected frame in the backtrace.

Command: info args

```
(gdb) info args
num1 = 2
num2 = 3
```

Step 13: Check the local variables of the currently selected frame in the backtrace.

Command: info locals

```
(gdb) info locals
tsum = 0
```

Step 14: Select the one frame to the outer side from the current frame.

Command: up 1

```
(gdb) up 1
#2  0x00005555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
30              result = sum_func(a,b);
```

Step 15: Select the one frame to the inner side from the current frame.

Command: up -1

```
(gdb) up -1
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
13              tsum = sum_f(num1,num2);
```

A silent alternative to this command is the “up-silently” command. It works in the same way but doesn’t print anything.

Step 16: Select the one frame to the outer side from the current frame.

Command: down -1

```
(gdb) down -1
#2  0x00005555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
30              result = sum_func(a,b);
```

Step 17: Select the one frame to the inner side from the current frame.

Command: down 1

```
(gdb) down 1
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
13              tsum = sum_f(num1,num2);
```

A silent alternative to this command is the “down-silently” command. It works in the same way but doesn’t print anything.

Step 18: Print the value of the stack pointer for all frames.

Command: frame apply all p $sp

```
(gdb) frame apply all p $sp
#0  sum_f (x=2, y=3) at sample2.c:8
$1 = (void *) 0x7fffffffe4a8
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
$2 = (void *) 0x7fffffffe4b8
#2  0x00005555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
$3 = (void *) 0x7fffffffe4e0
```

Step 19: Print the value of the stack pointer for frame 1.

Command: frame apply 1 p $sp

```
(gdb) frame apply 1 p $sp
#0  sum_f (x=2, y=3) at sample2.c:8
$4 = (void *) 0x7fffffffe4a8
```

Step 20: Print the value of the variable “a” from frame 1.

Command: frame apply 1 p a

```
(gdb) frame apply 1 p a
#0  sum_f (x=2, y=3) at sample2.c:8
No symbol "a" in current context.
```

Step 21: Print the value of the variable “a” for all frames.

Command: frame apply all p a

```
(gdb) frame apply all p a
#0  sum_f (x=2, y=3) at sample2.c:8
No symbol "a" in current context.
```

One can observe the error but not the value of “a” for any frame. To handle this -c or continue option can be used.

Step 22: Print the value of the variable “a” for all frames.

Command: frame apply all -c p a

```
(gdb) frame apply all -c p a
#0  sum_f (x=2, y=3) at sample2.c:8
No symbol "a" in current context.
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
No symbol "a" in current context.
#2  0x000055555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
$6 = 2
```

This method was able to locate the variable “a” and print the value. However, error are thrown for those frames where variable “a” is not present.

Step 23: Print the value of the variable “a” for all frames and suppress the erros.

Command: frame apply all -s p a

```
(gdb) frame apply all -s p a
#2  0x000055555555547e1 in main (argc=3, argv=0x7fffffffe5f8) at sample2.c:30
$7 = 2
```

The same can be achieved by using faas command (printing value of variable “num1”)

Command: faas p num1

```
(gdb) faas p num1
#1  0x0000555555554752 in sum_func (num1=2, num2=3) at sample2.c:13
$8 = 2
```

References:

1. GDB Documentation (https://sourceware.org/gdb/current/onlinedocs/gdb)