B4 - Unix System Programming

B-PSU-402

Bootstrap

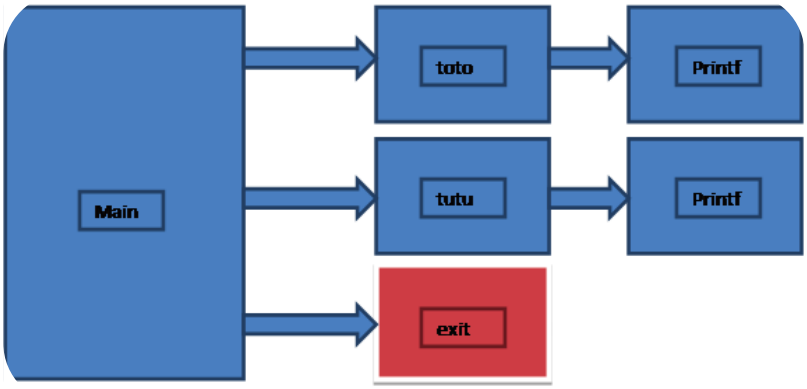
ftrace

2.0

{EPITECH.}

STEP 1: TEST FUNCTION

Throughout this entire Bootstrap, you will need a *test* binary.
It must contain 3 functions according to the following call graph:



The test program must have the following output:

Terminal

```
~/B-PSU-402> make toto
cc -02 -pipe -o toto toto.c
~/B-PSU-402> ./toto
i am in toto()
i am in tutu()
```

STEP 2: SYMBOL MAPPING

We still need a tool (a symbol mapper) to trace a program and find a call.
This call must contain an address that corresponds to a function but in order to use our ftrace, a symbol name is clearer than an address.

To do this, let's make a **symbol-finder** program that takes an hexadecimal address and returns the name that corresponds to the address.
To test the symbol finder on step 1, we first need to retrieve the main's address with **nm**:

Terminal

```
~/B-PSU-402> nm ./test
0000000008049594    d   _DYNAMIC
0000000008049660    d   _GLOBAL_OFFSET_TABLE_
00000000080484cc    R   _IO_stdin_used
                    w   _Jv_RegistrationClasses
0000000008049584    d   __CTOR_END__
0000000008049580    d   __CTOR_LIST__
000000000804958c    D   __DTOR_END__
0000000008049588    d   __DTOR_LIST__
000000000804857c    r   __FRAME_END__
0000000008049590    d   __JCR_END__
0000000008049590    d   __JCR_LIST__
000000000804967c    A   __bss_start
0000000008049678    D   __data_start
0000000008048480    t   __do_global_ctors_aux
0000000008048340    t   __do_global_dtors_aux
00000000080484d0    R   __dso_handle
                    w   __gmon_start__
000000000804847a    T   __i686.get_pc_thunk.bx
0000000008049580    d   __init_array_end
0000000008049580    d   __init_array_start
0000000008048410    T   __libc_csu_fini
0000000008048420    T   __libc_csu_fint
                    U   __libc_start_main@@GLIBC_2.0
000000000804967c    A   _edata
0000000008049684    A   _end
```

Next, launch your symbol finder with the main's address in order to retrieve the symbol.

Terminal

```
~/B-PSU-402> ./symbol-finder ./test 0x080483ec
Symbol main not found.
```

You are allowed to use the *libelf* (which allows you to make Elf parsing easier).

STEP 3: SIMPLE FTRACE

Write a program that traces all of the functions called with a 0xe8 type call.
Here's the related Intel man page:

```
E8 cd        B        Valid        Valid        Call near, relative,
                                                displacement relative to
                                                next instruction, 32-bit
                                                displacement sign extended
                                                to 64-bits in 64-bit mode.
```

Of course you are allowed to use:

- PTRACE_SINGLESTEP to scan your program, instruction by instruction,
- PTRACE_GETREGS to retrieve the eip value from each instruction
- PTRACE_PEEKTEXT to go retrieve the op codes pointed by eip.
  This instruction makes a relative call.
  It is composed of the 0xe8 opcode, and 4 bytes that correspond to an offset to be added to eip in order to attain the call address.

STEP 4: R/M MOD

Let's add a little assembler program that creates FF/2 type op codes.
If we compile it with this command:

Terminal

```
~/B-PSU-402> nasm test.asm -o test
```

and we disassemble it:

Terminal

```
~/B-PSU-402> ndisam -b 32 test

00000000    FF10            call dword ear [eax]
00000002    FFD3            call ebx
00000004    FFD1            call ecx
```

we can see that it corresponds to the op codes.

Check out this table to understand how it works

The first uses [eax] as an effective address and /2 in digital so the r/m mod will be equal to 10.
The second uses ebx as an effective address and /2 in digital, so the r/m mod will be equal to d3.
The first uses ecx as an effective address and /2 in digital, so the r/m mod will be equal to d1.

Now that you have these elements, you can run the FF/2 :-)
Good luck!