Computer Systems B (Security)

University of Bristol, UK

2021

Lab#1

In this lab, we will learn about

1. the internals of a program (its memory layout)
2. howo to use OBJDUMP tool to disassemble a given binary
3. how to use GDB (GNU debugger) to debug a given program
4. Using these tools, how to understand and manipulate a given program (process)

1. Code preparation
A. Compile the following c prog (also given separately as call-convention.c).

```
#include <stdio.h>
int func(int a, int b, int c, int d, int e, int f)
{
    int v1, v2;
    v1=a+b+c;//risky
    v2=d+e+f;//risky
    return (v1+v2)/2;
}

int main()
{
    int x;
    printf("IN the main\n");
    x= func(1,2,3,4,5,6);
    printf("X is: %d\n",x);
    return 0;
}
```

Compilation:

```
gcc -o call-conv64 call-convention.c
```

2. Objdump

As part of the compilation process, compile (GCC) converts the soruce code into the assembly instruction and then the assembler takes in assembly instructions and encodes them into the binary form understood by the hardware. Disassembly is the reverse process that converts binary-encoded instructions back into human-readable assembly. objdump is a tool that operates on object files (i.e. files containing compiled machine code).

A. Run `objdump --help` to see all the avaialble options.
B. Run the objdump as follows and then scroll upto the point when you see `main`.

```
$ objdump -d  call-conv64
```

This extracts the instructions from the object file and outputs the sequence of binary-encoded machine instructions alongside the assembly equivalent.

If the object file was compiled with debugging information, adding the -S flag to objdump will intersperse the original C source.
Run `objdump -d -S call-conv64` to see the source code together with the assembly.

Fig. 1 (The shown program is different from the call-convention)

```
000000000000068a <main>:
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
 68a:   55                      push   %rbp
 68b:   48 89 e5                mov    %rsp,%rbp
 68e:   48 83 ec 50             sub    $0x50,%rsp
 692:   89 7d bc                mov    %edi,-0x44(%rbp)
 695:   48 89 75 b0             mov    %rsi,-0x50(%rbp)
int index=100;
 699:   c7 45 fc 64 00 00 00    movl   $0x64,-0x4(%rbp)
char welcome[20]="Welcome to the Lab\n";
 6a0:   48 b8 57 65 6c 63 6f    movabs $0x20656d6f636c6557,%rax
 6a7:   6d 65 20
 6aa:   48 ba 74 6f 20 74 68    movabs $0x4c20656874206f74,%rdx
 6b1:   65 20 4c
 6b4:   48 89 45 e0             mov    %rax,-0x20(%rbp)
 6b8:   48 89 55 e8             mov    %rdx,-0x18(%rbp)
 6bc:   c7 45 f0 61 62 0a 00    movl   $0xa6261,-0x10(%rbp)
char name[20];
strcpy(name, argv[1]);
 6c3:   48 8b 45 b0             mov    -0x50(%rbp),%rax
 6c7:   48 83 c0 08             add    $0x8,%rax
 6cb:   48 8b 10                mov    (%rax),%rdx
 6ce:   48 8d 45 c0             lea    -0x40(%rbp),%rax
 6d2:   48 89 d6                mov    %rdx,%rsi
 6d5:   48 89 c7                mov    %rax,%rdi
 6d8:   e8 73 fe ff ff          callq  550 <strcpy@plt>
printf ("[*] Hi %s\n",name);
 6dd:   48 8d 45 c0             lea    -0x40(%rbp),%rax
 6e1:   48 89 c6                mov    %rax,%rsi
 6e4:   48 8d 3d c9 00 00 00    lea    0xc9(%rip),%rdi        # 7
 6eb:   b8 00 00 00 00          mov    $0x0,%eax
 6f0:   e8 6b fe ff ff          callq  560 <printf@plt>
printf("[*] %s\n", welcome);
 6f5:   48 8d 45 e0             lea    -0x20(%rbp),%rax
 6f9:   48 89 c6                mov    %rax,%rsi
 6fc:   48 8d 3d bc 00 00 00    lea    0xbc(%rip),%rdi        # 7
 703:   b8 00 00 00 00          mov    $0x0,%eax
 708:   e8 53 fe ff ff          callq  560 <printf@plt>
printf("Index is: %d\n",index);
 70d:   8b 45 fc                mov    -0x4(%rbp),%eax
 710:   89 c6                   mov    %eax,%esi
 712:   48 8d 3d ae 00 00 00    lea    0xae(%rip),%rdi        # 7
 719:   b8 00 00 00 00          mov    $0x0,%eax
 71e:   e8 3d fe ff ff          callq  560 <printf@plt>
return 0;
 723:   b8 00 00 00 00          mov    $0x0,%eax
}
 728:   c9                      leaveq
 729:   c3                      retq
```

Function prologue

arg/reg saving

index

String on stack

Argument passing

Function call

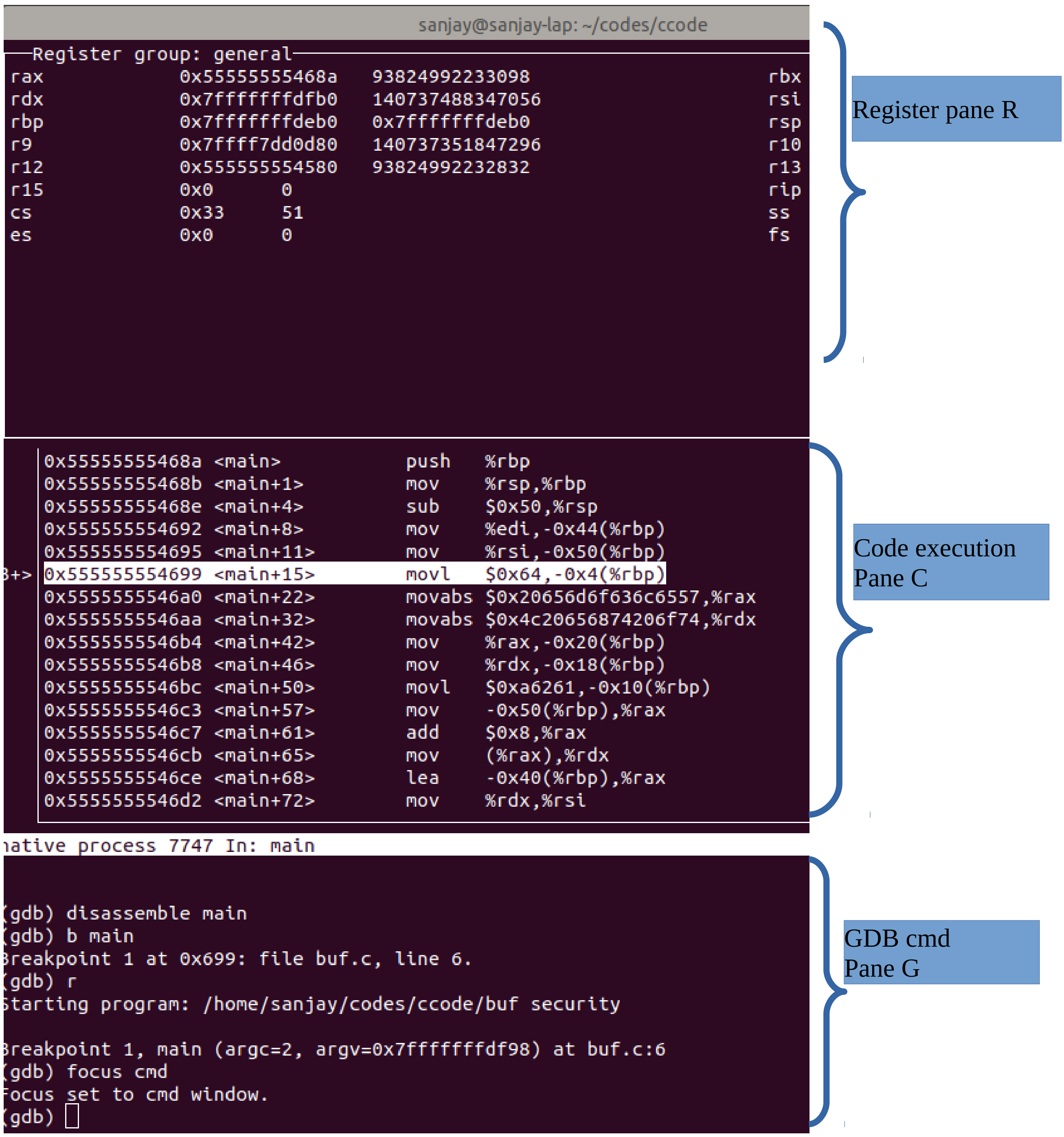
3. GDB

GDB stands for GNU Project Debugger and is a powerful debugging tool for C(along with other languages like C++). It helps you to monitor C programs while they are executing and also allows you to see what exactly happens when your program crashes. You can get the values of the registers and memory (e.g. stack). It allows you to set breakpoints at a certain point in your program execution. Though GDB is a commandline based program, you can, however, invoke its TUI (text user interface) to have separate windows displaying the values of registers, for example.

1. Run the GDB with the following command.

```
$ gdb call-conv64
```

Fig. 2



2. This will take you to the gdb command promt (see the Fig. 2). In that command prompt, type

```
layout regs
focus cmd
b main
run
disassemble main
```

3. At this stage, all the panes will have some values. The top most pane gives you values to all the register. The middle pane shows the assembly code being executed. And the botton pane is for the GDB commandline. You can note the value of `RIP` and the address of the current highlighted line! In the pane C, each line starts with a address, followed by the relative position marker and the instruction.
4. The execution will halt at the entry of main function, bacause you set a breakpoint at the `main (b main)`. Breakpoints can be set either by using the `b *address` OR `b *main+N`. Breakpoints are very useful when you want to analyse the values of register and memory. Try setting a breakpoint at some later point, say `b *main+60` and then run.
5. The program will halt when it reaches `main+60`. Now you can read the value of register, either by looking in the Pane R or by typing GDB command: `info reg`
6. You can also read the memory contentby `x/8xb $rbp-0x4` (remember, `rbp` is the base point, which also pojnts to the stack. In this case you will read 8 bytes starting from `EBP-4`. If you want to read entire stack, you can also use `RSP`. Use `ni` and `si` commands to observe how GDB executes next instruction. Try and get youself familiar with GDB (see the attached GDB cheatsheet)!

Exercise#1:

Compile the given c code (call-convention.c) with the following commands. **[Note: see the appendix A to make sure that your multi-arch compilation support is made available!]**

1. `gcc -m32 -o call-conv32 call-convention.c`
2. `gcc -o call-conv64 call-convention.c`

The above two steps will create two binary files, viz. `call-conv32 and call-conv64.`

1. Open call-conv32 with objdump (`objdump -d call-conv32`)
2. Look out for the disassembly of `main`
3. Observe the parameter passing just before the `call <func>`
4. Look out for the disassembly of `func`
5. Observe how those parameters (arguments) are used.

Repeat the above steps for call-conv64.