

Computer Systems B (Security)

University of Bristol, UK

2023

Addition to Lab#1

In this lab, we will learn about

1. the internals of a program (its memory layout)
2. how 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);
}
```

Compilation:

gcc -o call-conv64 call-convention.c

2. Objdump

As part of the compilation process, compile (GCC) converts the source 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 available 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.

0000000000000068a <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 movabs \$0x4c20656874206f74,%rdx

6aa: 48 ba 74 6f 20 74 68

6b1: 65 20 4c mov %rax,-0x20(%rbp)

6b4: 48 89 45 e0 mov %rdx,-0x18(%rbp)

6b8: 48 89 55 e8 movl \$0xa6261,-0x10(%rbp)

6bc: c7 45 f0 61 62 0a 00

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
```

The screenshot displays the GDB TUI interface with three distinct panes, each highlighted by a blue bracket and label on the right side.

- Register pane R:** This pane shows the state of various CPU registers. The title bar indicates the user is at `sanjay@sanjay-lap: ~/codes/ccode`. The content is organized into two columns. The left column lists registers: `rax`, `rdx`, `rbp`, `r9`, `r12`, `r15`, `cs`, and `es`. The right column lists registers: `rbx`, `rsi`, `rsp`, `r10`, `r13`, `rip`, `ss`, and `fs`. Each register is followed by its hexadecimal value. For example, `rax` is `0x55555555468a` and `93824992233098`.
- Code execution Pane C:** This pane displays the assembly code being executed. The title bar shows the address `0x555555554699` and the label `<main+15>`. The code includes instructions like `push %rbp`, `mov %rsp,%rbp`, `sub $0x50,%rsp`, `mov %edi,-0x44(%rbp)`, `mov %rsi,-0x50(%rbp)`, `movl $0x64,-0x4(%rbp)`, `movabs $0x20656d6f636c6557,%rax`, `movabs $0x4c20656874206f74,%rdx`, `mov %rax,-0x20(%rbp)`, `mov %rdx,-0x18(%rbp)`, `movl $0xa6261,-0x10(%rbp)`, `mov -0x50(%rbp),%rax`, `add $0x8,%rax`, `mov (%rax),%rdx`, `lea -0x40(%rbp),%rax`, and `mov %rdx,%rsi`.
- GDB cmd Pane G:** This pane shows the GDB command prompt and the execution flow. It starts with `(gdb) disassemble main`, followed by `(gdb) b main`, which sets a breakpoint at `0x699` in `buf.c`, line 6. Then `(gdb) r` is entered, starting the program `/home/sanjay/codes/ccode/buf security`. The output shows the breakpoint hit at `main (argc=2, argv=0x7fffffffdf98)` in `buf.c:6`. The prompt `(gdb) focus cmd` indicates the focus is on the command pane, and `Focus set to cmd window.` is displayed.

2. This will take you to the gdb command prompt (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 bottom 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, because 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 content by

x/8xb \$rbp-0x4 (remember, rbp is the base point, which also points 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 yourself familiar with GDB (see the attached GDB cheatsheet)!