Practical Session Week 5

Objectives

1. Introduction t
2. Introduction t

Basic commands o

1. *objdump -d bi embly*
2. *strings binary.file : show all strings*
3. *gcc file.c -o binary.file -g -O0/3*
4. *gcc file.c -S assembly.s -O0/3*

Basic commands on gdb

Please see this link https://sourceware.org/gdb/current/onlinedocs/gdb/

1. set disassembly-flavor intel : show intel syntax instead of AT&T
2. break or b : set a break point
   - b main : break to main function
   - b *0x0342FA0230 : break to this program address

3. run : goes to the first breakpoint
4. continue : run/go to the next breakpoint

5. return : step out of the function by cancelling its execution
6. si : Execute one machine instruction, then stop and return to the debugger
7. x/s : show the content of specific memory address
   - x/s 0x402400 or x/s $rax

8. info registers or i r : show the content of the registers, e.g., i r $rip shows the next instruction to be executed (%rip register holds the next instruction)
9. disas : show the assembly code at this point, or use ‘disas function1’ to display the assembly of this function
10. print : display individual register value
    - print /d $rax : display the value of rax register in decimal
    - print /t $rax : display the value of rax register in binary
    - print /x $rax : display the value of rax register in hexadecimal
11. The “x” command is used to display the values of specific memory locations: “x/nyz”
    - “n” is the number of fields to display
    - “y” is the format of the output, ‘c’ for character, ‘d’ for decimal and ‘x’ for hexadecimal
    - “z” is the size of the field to be displayed, ‘b’ for byte, ‘h’ for 16-bit word, ‘w’ for 32-bit word
    - ‘x/10xw $rsp’ : displays in hex first 10 32-bit contents of the stack

Tasks

1. Create a '.c' file and copy paste the following C code

```
#include <stdio.h>
 int main() {
        int a =
        int b =
        return
        }
```

Compile it with de[...] terminal) using the following command '*gcc file.c –o bin –g*'. Then run it using gdb using '*gdb ./bin*'. Follow the steps below:

- Tell debugger where to pause (gdb) break main
- Run the program (gdb) run
- Go the 2nd next instruction (gdb) si 2
- Look at register values (IA32) (gdb) i r
- Sneak peak at disassembly (gdb) disas

The last command will display the assembly code. '$' prefix is for immediates (constants), and the '%' prefix is for registers. Did you notice the names of the registers?

The assembly code for main() will look like :

```
   0x00000000004004d0 <+0>:    push   %rbp
   0x00000000004004d1 <+1>:    mov    %rsp,%rbp
=> 0x00000000004004d4 <+4>:    movl   $0x5,-0x8(%rbp)
   0x00000000004004db <+11>:   mov    -0x8(%rbp),%eax
   0x00000000004004de <+14>:   add    $0x6,%eax
   0x00000000004004e1 <+17>:   mov    %eax,-0x4(%rbp)
   0x00000000004004e4 <+20>:   mov    $0x0,%eax
   0x00000000004004e9 <+25>:   pop    %rbp
   0x00000000004004ea <+26>:   retq
```

Push and pop instructions have to do with the stack. We will learn more about the stack and %rbp, %rsp registers next week. This program contains redundant operations, e.g., eax register is initialized with the value of 5 by using 2 instructions. The compiler has not generated efficient code.

Type the following command '*set disassembly-flavor intel'* and then type 'disas' again. Do you notice the difference?

Repeat the above by compiling the '.c' code using '*gcc file.c –o bin –g –O2'*. This will enable the compiler to apply optimizations and change the code. Do you notice the difference? Now, the compiler has optimized the program. Given that the instructions in main have no effect (they are not printed or stored somewhere) they are eliminated as redundant code. Now the assembly code is :

```
=> 0x00000000004003c0 <+0>:    xor    %eax,%eax
   0x00000000004003c2 <+2>:    retq
```

Recall from the first week's session that 'xor %eax,%eax' is equivalent to 'mov $0x0,%eax'. So the main() returns zero.

Instructions in AT&T syntax use the format: (*mnemonic, source, destination*). The mnemonic is a human readable name [illegible] the instruction. Source and destination are operands and can be immediate values, registers, memory addresses, or labels. Immediate values are constants, and are prefixed by a $, e.g., $0x5 represents the number 5 in hexadecimal. Register names are always prefixed by a %.

If an operand is inside [illegible]rs to memory address, e.g., movl $0x5,-0x8(%rbp), stores the number '5' to the [illegible]s equivalent to (%rbp - 0x8). Here, %rbp is called the base register (more next w[illegible]splacement. You'll also notice that the mnemonic has the suffix l. This signifies t[illegible]e long (32 bits for integers). Other valid suffixes are byte, short, word, quad.

Another, more complex way of addressing memory is shown by the following instruction:
mov %edx, -0x16(%rbp,%rax,4), which stores the content of %edx to the memory address given by: (-0x16+%rbp + %rax*4), %rax holds the index while %rbp holds the base address. This could be used to store array elements, of 4 bytes each.

2. Create a '.c' file and copy paste the following code. Compile with *'gcc file.c –o exec -g'* option. Use GDB as above to understand the assembly code.


```
#include <stdio.h>

int main() {
      int i, temp=0;

      for (i=0; i<100; i++){
        temp+=i&18;
      printf("\nHi there, i = %d , temp=%d\n", i, temp);
      }

      return 0;
      }
```


The assembly should look like this (I have added comments to better understand what it does):

```
Dump of assembler code for function main:
   0x0000000000400520 <+0>:    push   %rbp                  //save the old base pointer (more next week)
   0x0000000000400521 <+1>:    mov    %rsp,%rbp  //make the stack pointer the base pointer (more next week)
=> 0x0000000000400524 <+4>:    sub    $0x10,%rsp      //allocates 16 bytes in the stack (more next week)
   0x0000000000400528 <+8>:    movl   $0x0,-0x4(%rbp)         //store temp to the stack at location rbp-4
   0x000000000040052f <+15>:   movl   $0x0,-0x8(%rbp)         //store i to the stack at location rbp-8
   0x0000000000400536 <+22>:   jmp    0x40055c <main+60>  //jump to 0x40055c below

   0x0000000000400538 <+24>:   mov    -0x8(%rbp),%eax         //put i variable to %eax
   0x000000000040053b <+27>:   and    $0x12,%eax              // eax=eax & 18
   0x000000000040053e <+30>:   add    %eax,-0x4(%rbp)         //temp=temp+eax
   0x0000000000400541 <+33>:   mov    -0x4(%rbp),%edx         //put temp to edx
   0x0000000000400544 <+36>:   mov    -0x8(%rbp),%eax         //put i to eax
   0x0000000000400547 <+39>:   mov    %eax,%esi               //put i to esi
```

```
0x0000000000400549 <+41>:   mov    $0x400614,%edi       //put a value related to printf to edi
0x000000000040054e <+46>:   mov    $0x0,%eax            //make eax zero as it will be used to store the output of printf()
0x0000000000400553 <+51>:   callq  0x4003f0 <printf@plt> //calls printf. its operands are in edi,esi,edx
0x0000000000400558 8(%rbp)                               //i=i+1
0x000000000040055c x8(%rbp)                              //if i<=99
0x0000000000400560 <main+24>                             //jump back - for loop

0x0000000000400562 ax //put zero to eax as it is the value that main() returns
0x0000000000400567 py ebp to esp and pop ebp (more next week)
0x0000000000400568 %rip (more next week)
```


Go step by step using ‘si’ instruction. Check the values of the registers involved in each step.

Every time a function is called, its operands are always stored in the (%rdi, %rsi, %rdx, % rcx, %r8, %r9) registers in that order. If there are more than six operands, the rest are stored in the stack. The return value of the function is always stored into %rax. The aforementioned registers are 8bytes each; in the case where the function operands are of 4bytes instead of 8, then the (%edi, %esi, %edx, % ecx, %r8, %r9) and eax registers, are used, respectively (they are the same registers).

Stack is a special region of your computer's memory that stores temporary variables created by each function. The stack is a LIFO data structure. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function ends, all of the variables pushed onto the stack by that function, are popped. Once a stack variable is freed, that region of memory becomes available for other stack variables. Each function allocates/deallocates its own space in the stack using two registers %rbp (base pointer) and %rsp (stack pointer). The first shows on the bottom, while the second shows on the top of the function’s space in the stack. This process will be explained next week.