## Lecture 1. Debugging a program

(intel instruction set: refer to

http://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/Chapter\_6/CH06-1.html)

Editing, compiling, running, and debugging a C program in Linux.

Understanding ASM code: where is local variable, stack change during call/ret instruction, ....

Understanding the process image.

1. Example program: ex1.c

#include <stdio.h>

void main(){

int x;

x=30;

printf("x is %d\n", x);

}

2. Compiling and Running

gcc –m32 -o ex1 ex1.c

./ex1

x is 30

3. ASM code

objdump -D -M intel ex1 > ex1.txt

vi ex1.txt

/main

repeat "/" until you find "<main>:"

080483c4 <main>:

80483c4: 55 push ebp

80483c5: 89 e5 mov ebp, esp

80483c7: 83 e4 f0 and esp, 0xfffffff0

80483ca: 83 ec 20 sub esp, 0x20

80483cd: c7 44 24 1c 1e 00 00 00 mov DWORD PTR [esp+0x1c], 0x1e

80483d5: b8 b4 84 04 08 mov eax, 0x80484b4

80483da: 8b 54 24 1c mov edx, DOWRD PTR [esp+0x1c]

80483de: 89 54 24 04 mov DWORD PTR [esp+0x4], edx

80483e2: 89 04 24 mov DWORD PTR [esp], eax

80483e5: e8 0a ff ff ff call 80482f4

80483ea: c9 leave

80483eb: c3 ret

push x

esp = esp – 4

mem[esp]  x

pop x

x  mem[esp]

esp = esp + 4

mov reg1, data

reg1  data

and reg, data

reg  reg AND data

sub reg, data

reg  reg – data

mov DWORD PTR [addr], data

4 byte in mem[addr]  data

call x

push return-addr (the address of the instruction after “call x”)

jump to x

leave

esp  ebp

pop ebp

ret

eip  mem[esp]

esp = esp + 4

(hw1) Edit, compile, and run ex1.c.

(hw2) Get "ex1.txt" as above and show the asm code for "main".

(hw3) Draw the memory map and show all the changes in registers and memory after each instruction up to the "ret" instruction. Assume esp = 0xbffff63c and ebp = 0xbffff6b8 in the beginning of "main".

(hw4) Find corresponding instructions for "x=30;" and "printf("x is %d\n",x);" in the ASM code.

(hw5) What is the memory location of the variable x?

(hw6) Find the memory address where the string "x is %d\n" is stored. Confirm the ascii codes for "x is %d\n" at that address.

(hw7) Show the memory address where main() begins.

4. Debugging

1) compile with "-m32" (for intel output format) and "-g" (for gdb) option

gcc -m32 -g -o ex1 ex1.c

2) copy ".gdbinit" to configure gdb

cp ../../linuxer1/.gdbinit .

3) run gdb

gdb ex1

...

disassemble main -- disassemble main() and show asm code for main

0x804841c <+0>: push ebp -- first instruction of main

....................

display $esp -- display the value of esp after each ni

display $ebp

display $eax

b \*0x804841c -- set break point at addr=0x804841c (first instr addr of main)

................

r -- start running the program

[0x002B:0xFFFFD5EC]------------------------------------------------------[stack]

0xFFFFD63C : 20 83 04 08 00 00 00 00 - F0 5D D0 44 79 D7 D2 44 ........].Dy..D

0xFFFFD62C : 00 00 00 00 00 00 00 00 - 00 00 00 00 01 00 00 00 ................

0xFFFFD61C : 00 00 00 00 00 00 00 00 - 5D 83 CC CE 2B 26 D7 94 ........]...+&..

0xFFFFD60C : 02 00 00 00 02 00 00 00 - 00 60 EC 44 00 00 00 00 .........`.D....

0xFFFFD5FC : B0 C6 FF F7 01 00 00 00 - 01 00 00 00 00 00 00 00 ................

0xFFFFD5EC : 65 D8 D2 44 01 00 00 00 - 84 D6 FF FF 8C D6 FF FF e..D............

--------------------------------------------------------------------------[code]

=> 0x804841c <main>: push ebp

0x804841d <main+1>: mov ebp,esp

0x804841f <main+3>: and esp,0xfffffff0

0x8048422 <main+6>: sub esp,0x20

--------------------------------------------------------------------------------

Breakpoint 1, main () at ex1.c:2

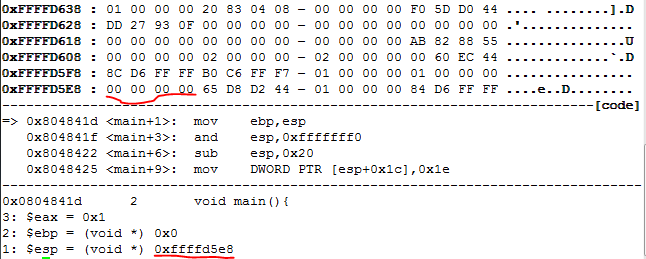
2 void main(){

3: $eax = 0x1

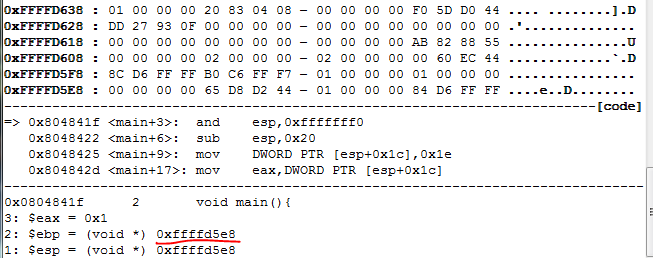
2: $ebp = (void \*) 0x0

1: $esp = (void \*) 0xffffd5ec

ni -- execute next instruction (“push ebp”)



ni -- execute next instruction (“mov ebp, esp”)



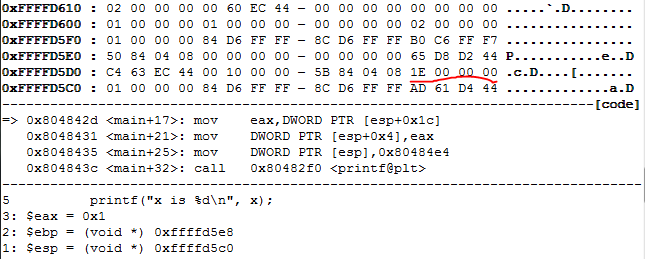
ni -- execute next instruction ("and esp, 0xfffffff0")

...................

ni -- execute "sub esp, 0x20"

....................

ni -- execute "mov dword ptr [esp+0x1c], 0x1e



ni -- execute "mov eax, DWORD PTR [esp+0x1c]

.................................

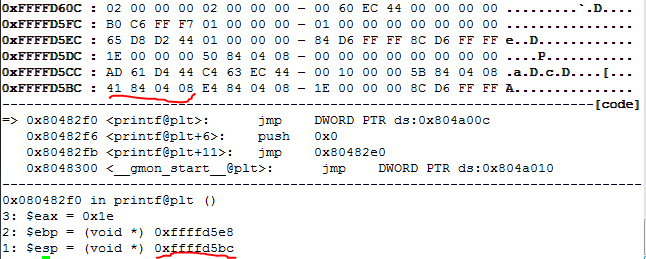
ni -- execute "DWORD PTR [esp+0x4], eax

.................................

ni -- execute DWORD PTR [esp], 0x80484e4

..................................

si -- execute "call printf" with si to enter the function



(HW 8) Follow above steps to show the content of the registers or memory that have been changed after each instruction in main(). For the “call” instruction execute it with “si” and show the changes in the stack.

5. gdb commands

(gdb) b \*addr -- break at addr

(gdb) b funcname -- break at function "funcname"

(gdb) r -- rerun

(gdb) bt -- backtrack stack frames

(gdb) p expr -- print the value of expr ex) p $sp or p/x $eax (in hexa)

(gdb) nexti -- run next instruction (do not go into a function). same as ni.

(gdb) stepi -- run next instruction (go inside a function). same as si.

(gdb) info f -- show the stack frame of the current function

(gdb) display $eip -- show the value of eip after every gdb command

(gdb) display $esp -- show the value of esp after every gdb command

(gdb) info registers -- show the value of all registers

(gdb) info registers eip -- show the value of eip

(gdb) info line --memory address of the current function

(gdb) info line main -- memory address of function main

(gdb) x/8xb addr -- show 8 bytes in hexa starting from addr

(gdb) x/20xh addr – show 20 half words (2 bytes) in hexa starting from addr

(gdb) x/13xw addr – show 13 words (4 bytes) in hexa starting from addr

6. (hw 9) Trace following program with gdb similarly as in hw8.

#include <stdio.h>

int foo(int x){

return x+1;

}

void main(){

int x=10;

x=foo(x);

printf("%d\n",x);

}

7. 64-bit Linux shows different output compared to 32-bit Linux

1) It uses 8-byte registers (rsp, rip, rbp, ...) instead of esp, eip, ebp, ...

2) The calling convention (how to pass function arguments) is different.

64-bit Linux passes function arguments in registers instead of stack:

The first six integer or pointer arguments are passed in rdi, rsi, rdx, rcx, r8, r9 (in that

order left to right), while xmm0, xmm1, xmm2, .., xmm7 are used for floating point arguments.

Additional arguments are passed on the stack and return value is stored in rax.