WHY DEBUGGING IN ASSEMBLY MODE?

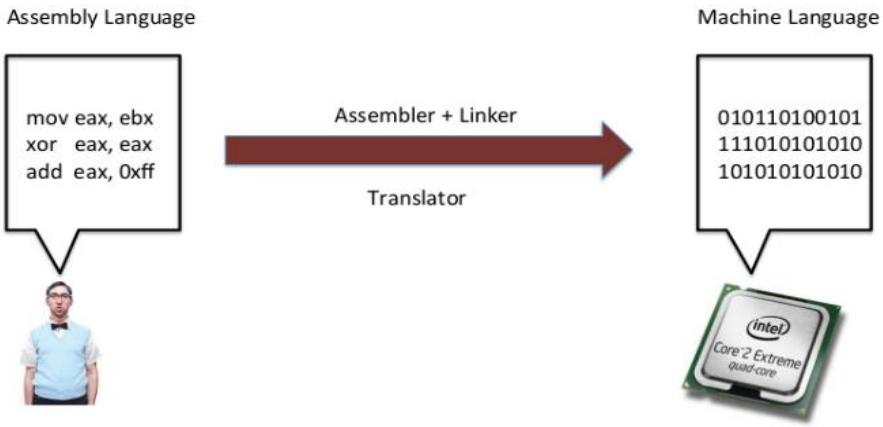
There are many times you cannot perform source debugging. In these situations, you have to debug in assembly mode. Moreover, assembly mode has many useful features that are not present in source debugging. The debugger automatically displays the contents of memory locations and registers as they are accessed and displays the address of the program counter. This display makes assembly debugging a valuable tool that you can use together with source debugging.

HOW TO DO THAT?

1. Launch the program in GDB debugger.
2. Set a breakpoint in the code at the location which you want to alter execution.
3. Execute the program and drive the program so that your breakpoint is hit.
4. Request the debugger to display the disassembly of the code.
5. Get comfortable with the source-assembly mapping.(X86 assembly language)
6. Identify the address of the assembly line you would like to alter.
7. Modify the memory location/registers with new opcodes to alter the logic.
8. Continue execution and now the program will respond to the new logic in the program.
9. To undo the effect of change, restart the program.

X86 ASSEMBLY FUNDAMENTALS

Assembly language (or assembler), is any low-level programming language in which there is a very strong correspondence between the program's statements and the architecture's machine code instructions.



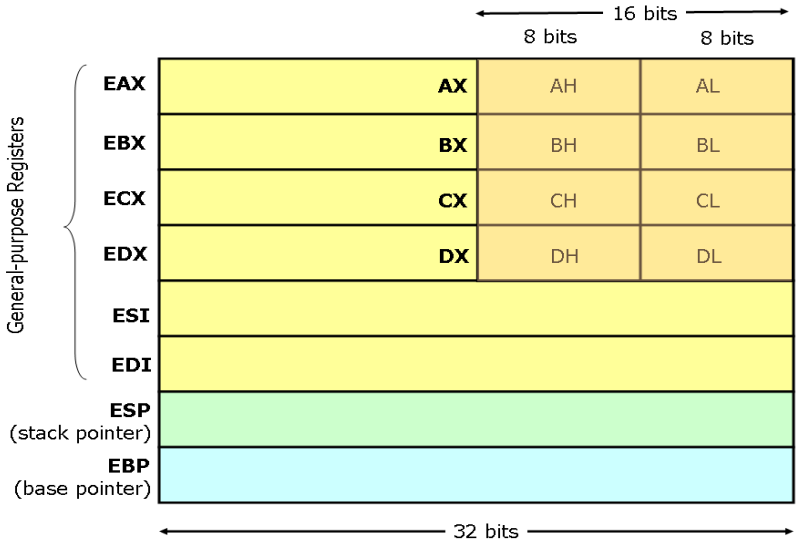
https://www.secjuice.com/guide-to-x86-assembly/

The x86 Architecture:

The x86 architecture has:

- 8 General-Purpose Registers (GPR)
- 6 Segment Registers
- 1 Flags Register
- 1 Instruction Pointer

https://notes.shichao.io/asm/#x86-architecture



What Are Registers?

Registers in assembly programming can be considered to be global variables we use in higher level programming languages for general operations.

Some Different Types of Registers :

- General purpose - Eax, Ebx, Esp, Ebp
- Control – EIP

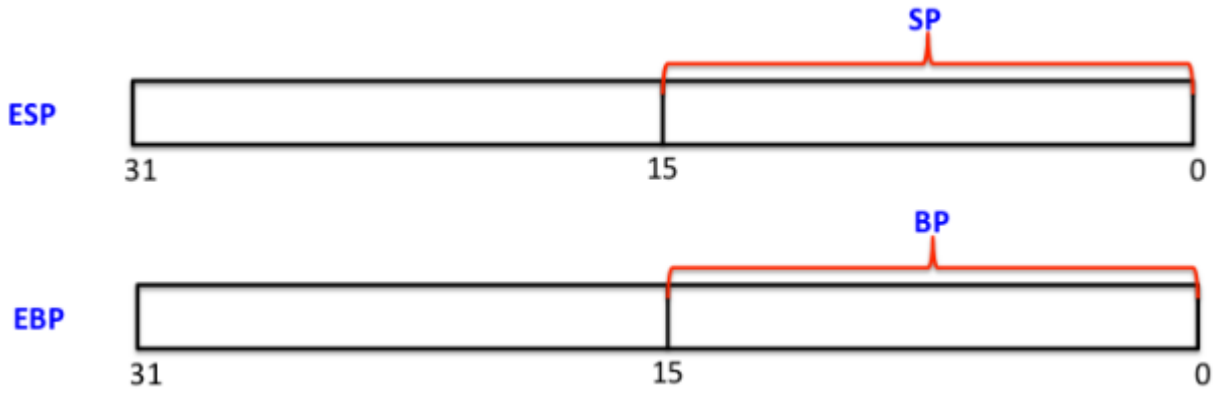
General Purpose Registers:

These are some of the general purpose registers in x86 architecture, each of the above register has capacity of storing 32 bit of data. Think of an EAX register with 32 bit, Lower part of EAX is called AX which contains 16 bit of data, AX is also further divided in two parts AH and AL, each with 8 bits in size, the same goes with EBX, ECX and EDX.

EAX - Accumulator Register - used for storing operands and result data

EBX- Base register - Points to data

ECX - Counter Register - Loop operations



Unlike registers we saw before, the above registers (ESP, EBP) can not be divided in small sizes of 8 bits, however they are divided in upper and lower 16 bits of register.Registers in a cpu are limited, you can't use them to store larger chunks of data and that's where memory comes to play. Data can be stored in memory in a stack data structure, the ESP register serves as an *indirect memory operand* pointing to the top of the stack at any time. EBP points to the base of a stack.

What doesn't fit in registers lives in memory

Memory is accessed either with loads and stores at addresses as if it were a big array, or through PUSH and POP operations on a stack.

Control Registers : EIP

Assembly is executed instruction wise and instructions are written in an orderly fashion.

```
_start:
```

1. mov $5, ecx
2. mov $5, edx
3. cmp ecx, edx

In above given assembly program, Execution is started with the symbol _start: EIP points to the next instruction to execute. Before the 1st instruction of "mov $5, ecx" is executed, EIP points to the address of the first instruction. After it is executed, EIP is then incremented by 1, so it will now point to the second instruction. Program execution would flow this way, as an attacker if we want to take control of the program, we should manipulate the value of EIP.

https://www.secjuice.com/guide-to-x86-assembly/

SOME BASIC INSTRUCTIONS:

Instruction	Effect	Examples
Copying Data		
mov *dest,src*	Copy src to dest	mov eax,10 mov eax,[2000]
Arithmetic		
add *dest,src*	dest = dest + src	add esi,10
sub *dest,src*	dest = dest – src	sub eax, ebx
mul *reg*	edx:eax = eax * reg	mul esi
div *reg*	edx = edx:eax **mod** reg eax = edx:eax ÷ reg	div edi
inc *dest*	Increment destination	inc eax
dec *dest*	Decrement destination	dec word [0x1000]
Function Calls		
call *label*	Push eip, transfer control	call format_disk
ret	Pop eip and return	ret
push *item*	Push item (constant or register) to stack. I.e.: esp=esp-4; memory[esp] = item	push dword 32 push eax
pop *[reg]*	Pop item from stack and store to register I.e.: reg=memory[esp]; esp=esp+4	pop eax
Bitwise Operations		
and *dest, src*	dest = src & dest	and ebx, eax
or *dest,src*	dest = src | dest	or eax,[0x2000]
xor *dest, src*	dest = src ^ dest	xor ebx, 0xffffffff
shl *dest,count*	dest = dest << count	shl eax, 2
shr *dest,count*	dest = dest >> count	shr dword [eax],4

Conditionals and Jumps		
cmp *b,a*	Compare b to a; must immediately precede any of the conditional jump instructions	cmp eax,0
je *label*	Jump to label if b == a	je endloop
jne *label*	Jump to label if b != a	jne loopstart
jg *label*	Jump to label if b > a	jg exit
jge *label*	Jump to label if b ≥ a	jge format_disk
jl *label*	Jump to label if b < a	jl error
jle *label*	Jump to label if b ≤ a	jle finish
test *reg,imm*	Bitwise compare of register and constant; should immediately precede the jz or jnz instructions	test eax,0xffff
jz *label*	Jump to label if bits were **not** set (“zero”)	jz looparound
jnz *label*	Jump to label if bits **were** set (“not zero”)	jnz error
jmp *label*	Unconditional relative jump	jmp exit
jmp *reg*	Unconditional absolute jump; arg is a register	jmp eax
Miscellaneous		
nop	No-op (opcode 0x90)	nop
hlt	Halt the CPU	hlt

https://www.bencode.net/blob/nasmcheatsheet.pdf

TEST

TEST instruction performs a bitwise AND on two operands. The flags SF, ZF, PF are modified while the result of the AND is discarded.

```
; Conditional Jump
test cl, cl   ; set ZF to 1 if cl == 0
je 0x804f430  ; jump if ZF == 1

; Conditional Jump with NOT
test cl, cl   ; set ZF to 1 if cl == 0
jne 0x804f430  ; jump if ZF != 1
```

What is Gdb?

A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise.

GDB allows you to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

GDB - Commands

GDB offers a big list of commands, h/owever the following commands are the ones used most frequently:

- **b N** - Puts a breakpoint at line N
- **b *[Address]** – puts a breakpoint at the specified instruction address.
- **d N** - Deletes breakpoint number N
- **info break** - list breakpoints
- **r** - Runs the program until a breakpoint or error
- **c** - Continues running the program until the next breakpoint or error
- **stepi or si**-Execute one machine instruction (follows a call).
- **backtrace** - Prints a stack trace
- **q** - Quits gdb
- **set disassembly-flavor intel –** sets the syntax of assembly code to intel (easier to understand)
- **disassemble [Function} –** prints out the assembly code for the specified function.

Disassemble a code using gdb:

Compile your C program with -g option. This allows the compiler to collect the debugging information.

```
$ cc -g factorial.c
```

Launch the C debugger (gdb) as shown below.

```
 $ gdb a.out
```

Before starting, we need to change the disassembly style to Intel (for a better readability);

```
set disassembly-flavor intel
```

disassemble code using :

disassemble

Set up a break point inside C program

```
(gdb) Break *address

(gdb) Break line_number

(gdb) Break function
```

Execute the C program in gdb debugger

```
(gdb) run parameters
```

Examining registers

To inspect the current values of registers:
(gdb) info registers
This prints out the current values of all registers.

Note: if you are debugging a 64-bit program, replace the EXX regirsters with RXX (e.g. use $rax instead of $eax). Using 'p $eax' to print just the lower 32 bits of the register doesn't work (at least with some versions of gdb). You have to print a full 64-bit register.

Change memory in registers

```
(gdb)set  $register_name=value
```

Problem:

```
#include<stdio.h>
void main()
{
   char username[20];
   char pwd[10];
   printf("please enter username");
   gets(username);
   if(strcmp(username,"test1234")==0)
   {
      printf("\n correct username \n Enter password");
      gets(pwd);
      if(strcmp(pwd,"pass")==0)
      {
         printf("\n Access granted!");
      }
      else
      {
         printf("wrong password");
         goto exit;
      }
   }
   else
   {
      printf("wrong username");
      goto exit;
   }
   exit:
   printf("program exited.");
}
```

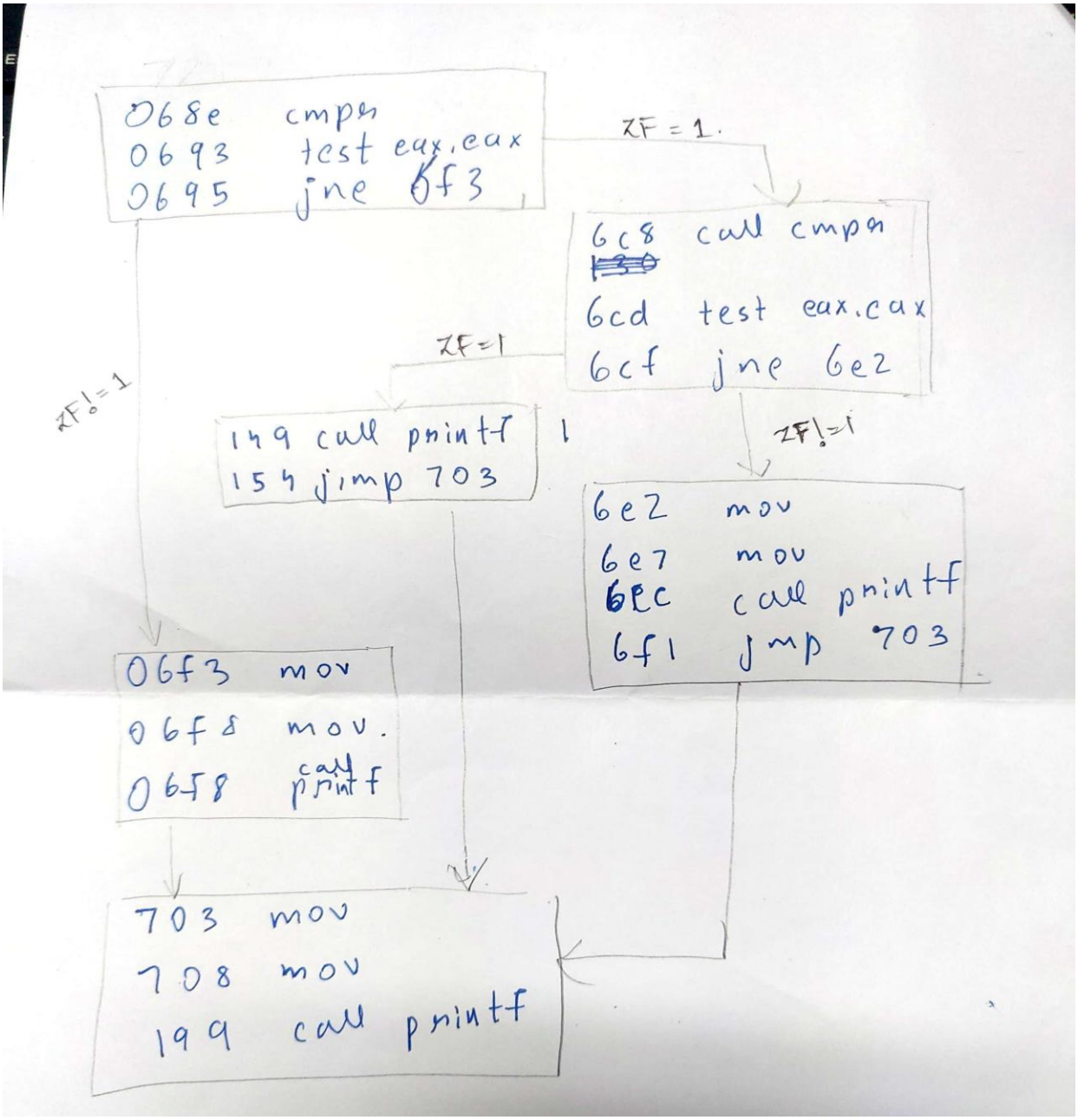
We know the correct username but not the password. Task is to reach “access granted” without knowing the password.

Assembly Code:

```
0x0000000000400646 <+0>:     push   rbp
0x0000000000400647 <+1>:     mov    rbp,rsp
0x000000000040064a <+4>:     sub    rsp,0x30
0x000000000040064e <+8>:     mov    rax,QWORD PTR fs:0x28
0x0000000000400657 <+17>:    mov    QWORD PTR [rbp-0x8],rax
0x000000000040065b <+21>:    xor    eax,eax
0x000000000040065d <+23>:    mov    edi,0x4007b8
0x0000000000400662 <+28>:    mov    eax,0x0
0x0000000000400667 <+33>:    call   0x400500 <printf@plt>
0x000000000040066c <+38>:    lea    rax,[rbp-0x20]
0x0000000000400670 <+42>:    mov    rsi,rax
0x0000000000400673 <+45>:    mov    edi,0x4007ce
0x0000000000400678 <+50>:    mov    eax,0x0
0x000000000040067d <+55>:    call   0x400530 <__isoc99_scanf@plt>
```

```
0x0000000000400682 <+60>:    lea    rax,[rbp-0x20]
0x0000000000400686 <+64>:    mov    esi,0x4007d1
0x000000000040068b <+69>:    mov    rdi,rax
0x000000000040068e <+72>:    call   0x400520 <strcmp@plt>
0x0000000000400693 <+77>:    test   eax,eax
0x0000000000400695 <+79>:    jne    0x4006f3 <main+173>
0x0000000000400697 <+81>:    mov    edi,0x4007e0
0x000000000040069c <+86>:    mov    eax,0x0
0x00000000004006a1 <+91>:    call   0x400500 <printf@plt>
0x00000000004006a6 <+96>:    lea    rax,[rbp-0x30]
0x00000000004006aa <+100>:   mov    rsi,rax
0x00000000004006ad <+103>:   mov    edi,0x4007ce
0x00000000004006b2 <+108>:   mov    eax,0x0
0x00000000004006b7 <+113>:   call   0x400530 <__isoc99_scanf@plt>
0x00000000004006bc <+118>:   lea    rax,[rbp-0x30]
0x00000000004006c0 <+122>:   mov    esi,0x400804
0x00000000004006c5 <+127>:   mov    rdi,rax
0x00000000004006c8 <+130>:   call   0x400520 <strcmp@plt>
0x00000000004006cd <+135>:   test   eax,eax
0x00000000004006cf <+137>:   jne    0x4006e2 <main+156>
0x00000000004006d1 <+139>:   mov    edi,0x400809
0x00000000004006d6 <+144>:   mov    eax,0x0
0x00000000004006db <+149>:   call   0x400500 <printf@plt>
0x00000000004006e0 <+154>:   jmp    0x400703 <main+189>
0x00000000004006e2 <+156>:   mov    edi,0x40081b
0x00000000004006e7 <+161>:   mov    eax,0x0
0x00000000004006ec <+166>:   call   0x400500 <printf@plt>
0x00000000004006f1 <+171>:   jmp    0x400703 <main+189>
0x00000000004006f3 <+173>:   mov    edi,0x40082a
0x00000000004006f8 <+178>:   mov    eax,0x0
0x00000000004006fd <+183>:   call   0x400500 <printf@plt>
0x0000000000400702 <+188>:   nop
0x0000000000400703 <+189>:   mov    edi,0x400839
0x0000000000400708 <+194>:   mov    eax,0x0
0x000000000040070d <+199>:   call   0x400500 <printf@plt>
0x0000000000400712 <+204>:   nop
0x0000000000400713 <+205>:   mov    rax,QWORD PTR [rbp-0x8]
0x0000000000400717 <+209>:   xor    rax,QWORD PTR fs:0x28
0x0000000000400720 <+218>:   je     0x400727 <main+225>
0x0000000000400722 <+220>:   call   0x4004f0 <__stack_chk_fail@plt>
0x0000000000400727 <+225>:   leave
0x0000000000400728 <+226>:   ret
```

Map:



Vulnerability:

A vulnerability, in information technology (IT), is a flaw in code or design that creates a potential point of security compromise for an endpoint or network.

```
if(strcmp(pwd,"pass")==0)
{
   printf("\n Access granted!");
}
```

We can see that the password is checked by strcmp function. if the strings match strcmp returns 0 and hence “access granted” is printed. This is vunerability of h.c.

Assembly code:

```
0x00000000004006c8 <+130>:    call   0x400520 <strcmp@plt>
0x00000000004006cd <+135>:    test   eax,eax
0x00000000004006cf <+137>:    jne    0x4006e2 <main+156>
0x00000000004006d1 <+139>:    mov    edi,0x400809
0x00000000004006d6 <+144>:    mov    eax,0x0
0x00000000004006db <+149>:    call   0x400500 <printf@plt>
```

At assembly level, Strcmp function returns either -1,0 or 1 in EAX register with 0 indicating both strings match.TEST EAX,EAX tests whether EAX is zero or not and sets or unsets the ZF bit.

Eax contains the return value of strcmp. Anding a value with itself gives the same value, so test eax, eax sets the flags based on whatever eax contains. ZF is set when the result of an operation is zero.jne makes a jump when not equal i.e when zf flag=0.

Depending on the value of zf bit, jne either makes the jumps to “wrong password” or to “access granted”. Thus we can manipulate the value of eax register using gdb and assembly code. Setting eax register value to zero manually would indicate that the strings have matched and we would gain access.

DEBUGGING:

```
(gdb) b *0x00000000004006cd
Breakpoint 1 at 0x4006cd: file h.c, line 13.
```

Breakpoint is set at the line where the password is being checked i.e if(strcmp(pwd,"pass")==0)
Instruction address: 0x00000000004006cd
Line number: 13

```
(gdb) run
```

Runs the program till the first(and only) breakpoint set.

```
please enter usernametest1234

 correct username
 Enter password2143

Breakpoint 1, 0x00000000004006cd in main () at h.c:13
13              if(strcmp(pwd,"pass")==0)
```

According to the problem, we know the correct username and need to gain access without knowing the password. Hence correct username and wrong password were entered. We encounter the first breakpoint.

```
(gdb) info registers
rax            0xffffffc2  4294967234
rbx            0x0         0
rcx            0xa         10
rdx            0x70        112
rsi            0x400804 4196356
rdi            0x7fffffffdd70       140737488346480
rbp            0x7fffffffdda0       0x7fffffffdda0
```

```
rsp            0x7fffffffdd70      0x7fffffffdd70
r8             0x0          0
r9             0x7ffff7fdd700      140737353996032
r10            0x4007ce           4196302
r11            0x246      582
r12            0x400550           4195664
r13            0x7fffffffde80      140737488346752
r14            0x0          0
r15            0x0          0
rip            0x4006cd 0x4006cd <main+135>
eflags           0x283    [ CF SF IF ]
cs             0x33         51
ss             0x2b         43
ds             0x0          0
es             0x0          0
fs             0x0          0
gs             0x0          0
```

We listed the information held in the registers. We see that rax register has a nonzero value indicating unmatched strings.

```
(gdb) set $rax=0
```

This is where we manipulate the control flow. As explained earlier the rax is supposed to have 0 if the password was correct. So setting it to zero manually indicates correct password and hence test eax eax sets the zf bit as 1 and jne doesn't jump to "wrong password" and lets us proceed to next instruction.

```
(gdb) info registers
rax            0x0          0
rbx            0x0          0
rcx            0xa          10
rdx            0x70         112
rsi            0x400804 4196356
rdi            0x7fffffffdd70      140737488346480
rbp            0x7fffffffdda0      0x7fffffffdda0
rsp            0x7fffffffdd70      0x7fffffffdd70
r8             0x0          0
r9             0x7ffff7fdd700      140737353996032
r10            0x4007ce           4196302
r11            0x246      582
r12            0x400550           4195664
r13            0x7fffffffde80      140737488346752
r14            0x0          0
r15            0x0          0
rip            0x4006cd 0x4006cd <main+135>
eflags           0x283    [ CF SF IF ]
cs             0x33         51
ss             0x2b         43
ds             0x0          0
es             0x0          0
fs             0x0          0
gs             0x0          0
```

we can see rax is now zero.

```
(gdb) ni
0x00000000004006cf      13              if(strcmp(pwd,"pass")==0)
(gdb)
15              printf("\n Access granted!");
(gdb)
0x00000000004006d6      15              printf("\n Access granted!");
(gdb)
0x00000000004006db      15              printf("\n Access granted!");
(gdb)

0x00000000004006e0      15              printf("\n Access granted!");
(gdb)
29          printf("program exited.");
(gdb)
0x0000000000400708      29          printf("program exited.");
(gdb)
0x000000000040070d      29          printf("program exited.");
(gdb)
30      }
(gdb)
```

Thus we successfully gained access without knowing the password.

VULNERABLE PROGRAM 2:

```
#include<stdio.h>

#include<string.h>


int main(int argc, char**argv)

{

int authentication=0;

char cUsername[10], cPassword[10];

strcpy(cUsername, argv[1]);

strcpy(cPassword, argv[2]);

if(strcmp(cUsername,"admin")==0 && strcmp(cPassword,"adminpass")==0)

{

authentication=1;

}

if(authentication)
```

```
{
printf("access granted");
}
else
{
printf("wrong username and password");
}
return 0;
}
```

Vulnerability:

Access parameters are correct if authentication=1. If the value of authentication is directly manipulated, the security is compromised without figuring out what the username and password is.

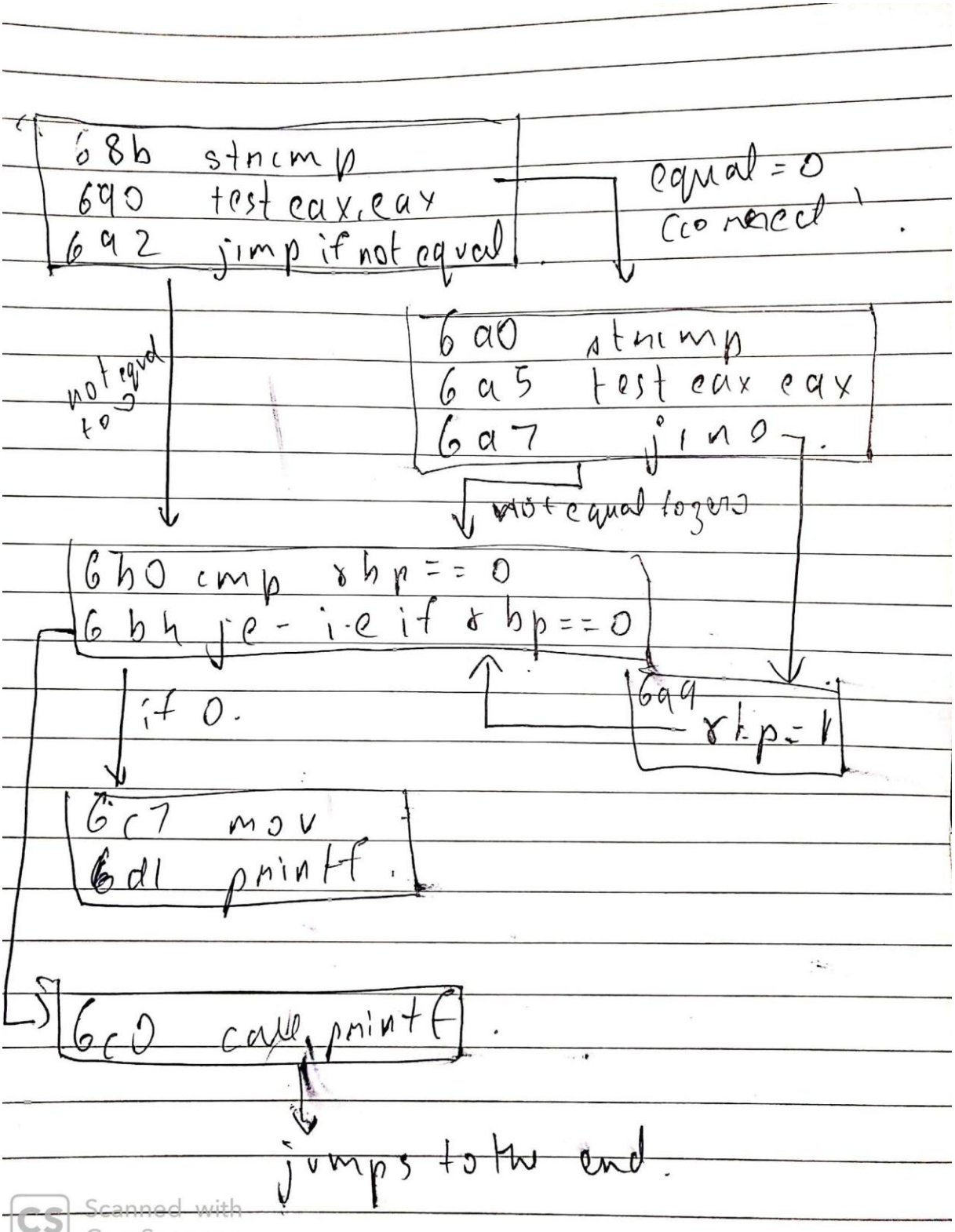
ASSEMBLY CODE: Highlighted instructions are the important ones which help us build a map and understand the flow of program.

```
0x0000000000400626 <+0>:     push rbp
0x0000000000400627 <+1>:     mov rbp,rsp
0x000000000040062a <+4>:     sub rsp,0x50
0x000000000040062e <+8>:     mov DWORD PTR [rbp-0x44],edi
0x0000000000400631 <+11>:    mov QWORD PTR [rbp-0x50],rsi
0x0000000000400635 <+15>:    mov rax,QWORD PTR fs:0x28
0x000000000040063e <+24>:    mov QWORD PTR [rbp-0x8],rax
0x0000000000400642 <+28>:    xor eax,eax
0x0000000000400644 <+30>:    mov DWORD PTR [rbp-0x34],0x0
0x000000000040064b <+37>:    mov rax,QWORD PTR [rbp-0x50]
0x000000000040064f <+41>:    add rax,0x8
0x0000000000400653 <+45>:    mov rdx,QWORD PTR [rax]
0x0000000000400656 <+48>:    lea rax,[rbp-0x30]
0x000000000040065a <+52>:    mov rsi,rdx
0x000000000040065d <+55>:    mov rdi,rax
0x0000000000400660 <+58>:    call 0x4004d0 <strcpy@plt>
0x0000000000400665 <+63>:    mov rax,QWORD PTR [rbp-0x50]
```

```
0x0000000000400669 <+67>:    add rax,0x10
0x000000000040066d <+71>:    mov rdx,QWORD PTR [rax]
0x0000000000400670 <+74>:    lea rax,[rbp-0x20]
0x0000000000400674 <+78>:    mov rsi,rdx
0x0000000000400677 <+81>:    mov rdi,rax
0x000000000040067a <+84>:    call 0x4004d0 <strcpy@plt>
0x000000000040067f <+89>:    lea rax,[rbp-0x30]
0x0000000000400683 <+93>:    mov esi,0x400784
0x0000000000400688 <+98>:    mov rdi,rax
0x000000000040068b <+101>:   call 0x400510 <strcmp@plt>
0x0000000000400690 <+106>:   test eax,eax
0x0000000000400692 <+108>:   jne 0x4006b0 <main+138>
0x0000000000400694 <+110>:   lea rax,[rbp-0x20]
0x0000000000400698 <+114>:   mov esi,0x40078a
0x000000000040069d <+119>:   mov rdi,rax
0x00000000004006a0 <+122>:   call 0x400510 <strcmp@plt>
0x00000000004006a5 <+127>:   test eax,eax
0x00000000004006a7 <+129>:   jne 0x4006b0 <main+138>
0x00000000004006a9 <+131>:   mov DWORD PTR [rbp-0x34],0x1
JMP if not eql-> 0x00000000004006b0 <+138>:      cmp DWORD PTR [rbp-0x34],0x0
0x00000000004006b4 <+142>:   je 0x4006c7 <main+161>
0x00000000004006b6 <+144>:   mov edi,0x400794
0x00000000004006bb <+149>:   mov eax,0x0
0x00000000004006c0 <+154>:   call 0x4004f0 <printf@plt>
---Type <return> to continue, or q <return> to quit---c
0x00000000004006c5 <+159>:   jmp 0x4006d6 <main+176>
JMP if eql--> 0x00000000004006c7 <+161>: mov edi,0x4007a3
0x00000000004006cc <+166>:   mov eax,0x0
0x00000000004006d1 <+171>:   call 0x4004f0 <printf@plt>
JMP-> 0x00000000004006d6 <+176>:      mov eax,0x0
0x00000000004006db <+181>:   mov rcx,QWORD PTR [rbp-0x8]
0x00000000004006df <+185>:   xor rcx,QWORD PTR fs:0x28
0x00000000004006e8 <+194>:   je 0x4006ef <main+201>
0x00000000004006ea <+196>:   call 0x4004e0 <__stack_chk_fail@plt>
```

```
0x00000000004006ef <+201>:    leave
0x00000000004006f0 <+202>:    ret
End of assembler dump.
(gdb) Quit
(gdb)
```

MAP:



Procedure:

First strcmp is for username. If it is correct , it jumps to 6a0 -> second strcmp for password. If that is also correct it jumps to 6a9 where address stored in rbp has now the value set to 1. Remember, rbp is a pointer which points to an address that stores the deciding value (authentication). It then goes to 6b0 where its checked if that address contains 1. If yes, it goes to 6c0 prints “access granted” and exits.

Therefore, we need to control the value of rbp and set it to 1.

TERMINAL(GDB):

```
guest-m6pwhf@kjsce-OptiPlex-3020:~$ gcc -g vuln2.c

guest-m6pwhf@kjsce-OptiPlex-3020:~$ gdb a.out
```

```
(gdb) set disassembly-flavor intel

(gdb) disass main

0x00000000004006b0 <+138>:      cmp DWORD PTR [rbp-0x34],0x0
```

//At 6b0, it is checking if the address storing authentication has value 0 or 1. Note that address is rbp – 0x34. Set a breakpoint here to get the address and manipulate the value in it.

```
(gdb) b *0x00000000004006b0
```

```
(gdb) r adminss pass

Breakpoint 1, main (argc=3, argv=0x7fffffffde48) at vuln2.c:14

14      if(authentication)

(gdb) info registers
```

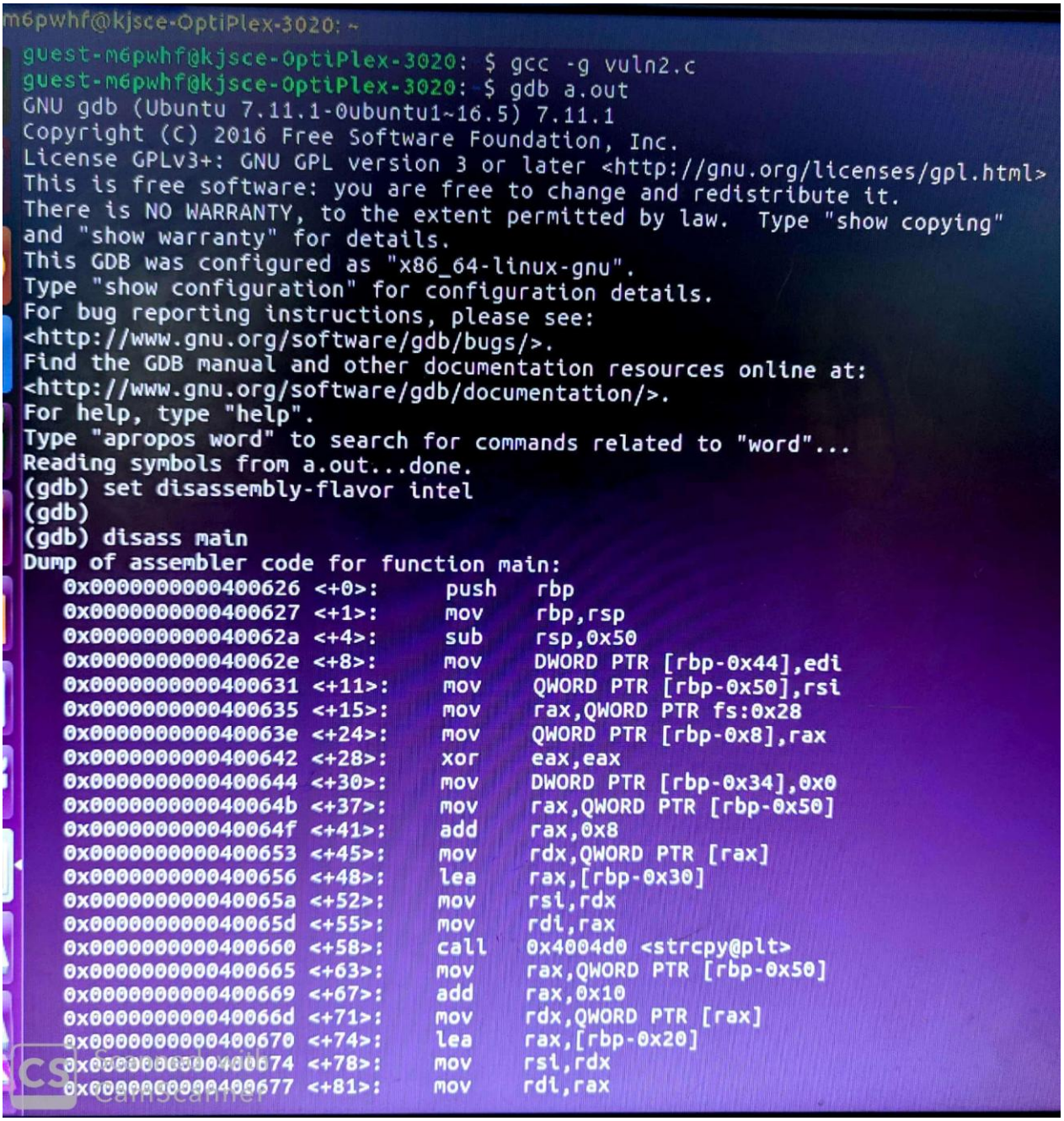
//Here we determine the value of rbp and minus 0x34 which is equal to address of authentication.

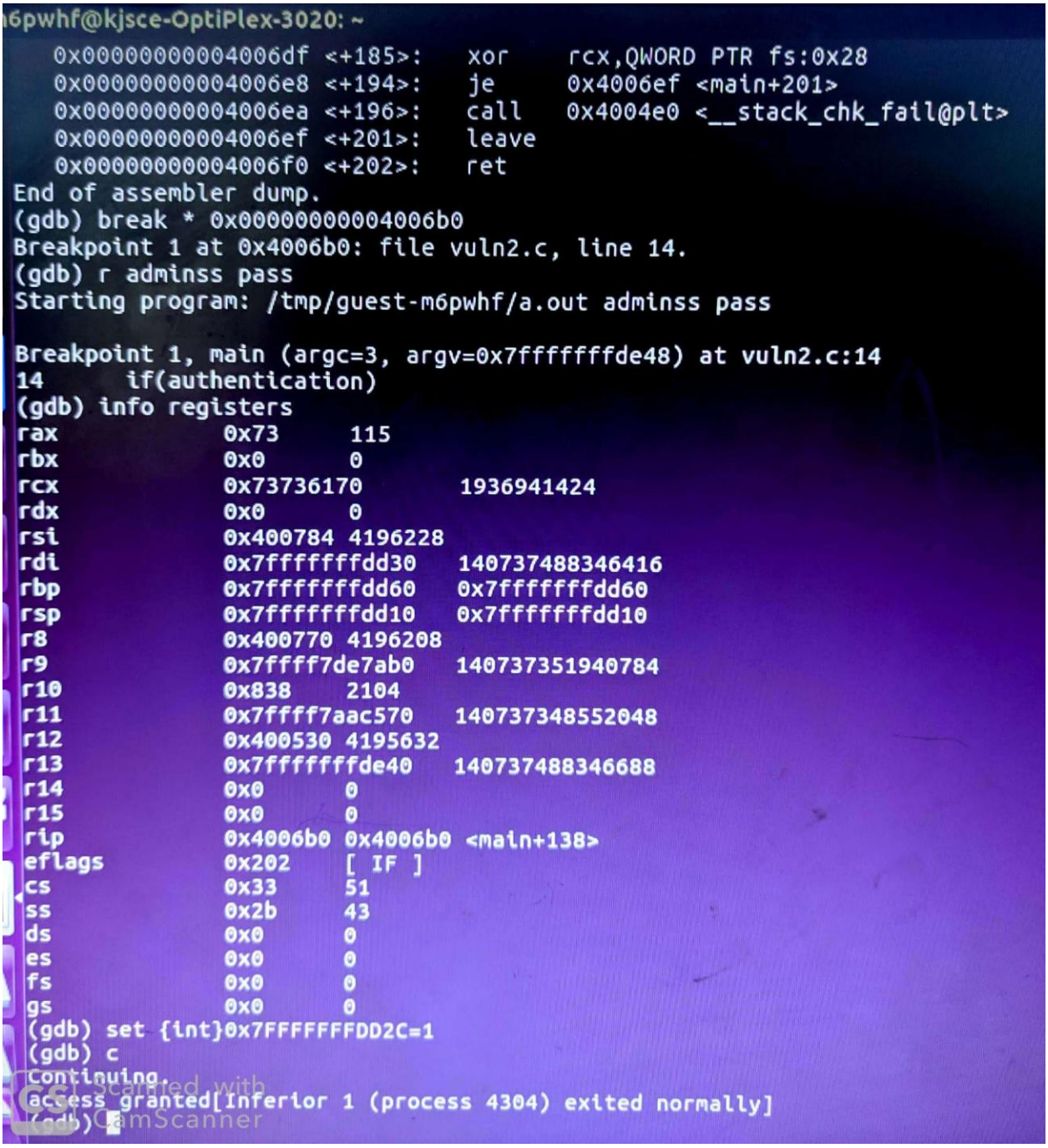
```
rbp 0x7fffffffdd60      0x7fffffffdd60
```

//rbp – 0x34 = 0x7FFFFFFFDD2C

```
(gdb) set {int}0x7FFFFFFFDD2C=1

(gdb) c

Continuing.

access granted[Inferior 1 (process 4048) exited normally]
```

SCREENSHOTS:


m6pwhf@kjsce-OptiPlex-3020: ~
guest-m6pwhf@kjsce-OptiPlex-3020: $ gcc -g vuln2.c
guest-m6pwhf@kjsce-OptiPlex-3020: $ gdb a.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb) set disassembly-flavor intel
(gdb)
(gdb) disass main
Dump of assembler code for function main:
0x0000000000400626 <+0>: push rbp
0x0000000000400627 <+1>: mov rbp,rsp
0x000000000040062a <+4>: sub rsp,0x50
0x000000000040062e <+8>: mov DWORD PTR [rbp-0x44],edi
0x0000000000400631 <+11>: mov QWORD PTR [rbp-0x50],rsi
0x0000000000400635 <+15>: mov rax,QWORD PTR fs:0x28
0x000000000040063e <+24>: mov QWORD PTR [rbp-0x8],rax
0x0000000000400642 <+28>: xor eax,eax
0x0000000000400644 <+30>: mov DWORD PTR [rbp-0x34],0x0
0x000000000040064b <+37>: mov rax,QWORD PTR [rbp-0x50]
0x000000000040064f <+41>: add rax,0x8
0x0000000000400653 <+45>: mov rdx,QWORD PTR [rax]
0x0000000000400656 <+48>: lea rax,[rbp-0x30]
0x000000000040065a <+52>: mov rsi,rdx
0x000000000040065d <+55>: mov rdi,rax
0x0000000000400660 <+58>: call 0x4004d0 <strcpy@plt>
0x0000000000400665 <+63>: mov rax,QWORD PTR [rbp-0x50]
0x0000000000400669 <+67>: add rax,0x10
0x000000000040066d <+71>: mov rdx,QWORD PTR [rax]
0x0000000000400670 <+74>: lea rax,[rbp-0x20]
0x0000000000400674 <+78>: mov rsi,rdx
0x0000000000400677 <+81>: mov rdi,rax


6pwhf@kjsce-OptiPlex-3020: ~
0x00000000004006df <+185>: xor rcx,QWORD PTR fs:0x28
0x00000000004006e8 <+194>: je 0x4006ef <main+201>
0x00000000004006ea <+196>: call 0x4004e0 <__stack_chk_fail@plt>
0x00000000004006ef <+201>: leave
0x00000000004006f0 <+202>: ret
End of assembler dump.
(gdb) break * 0x00000000004006b0
Breakpoint 1 at 0x4006b0: file vuln2.c, line 14.
(gdb) r adminss pass
Starting program: /tmp/guest-m6pwhf/a.out adminss pass
Breakpoint 1, main (argc=3, argv=0x7fffffffde48) at vuln2.c:14
14 if(authentication)
(gdb) info registers
rax 0x73 115
rbx 0x0 0
rcx 0x73736170 1936941424
rdx 0x0 0
rsi 0x400784 4196228
rdi 0x7fffffffdd30 140737488346416
rbp 0x7fffffffdd60 0x7fffffffdd60
rsp 0x7fffffffdd10 0x7fffffffdd10
r8 0x400770 4196208
r9 0x7ffff7de7ab0 140737351940784
r10 0x838 2104
r11 0x7ffff7aac570 140737348552048
r12 0x400530 4195632
r13 0x7fffffffde40 140737488346688
r14 0x0 0
r15 0x0 0
rip 0x4006b0 0x4006b0 <main+138>
eflags 0x202 [IF]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0x0 0
(gdb) set {int}0x7FFFFFFFDD2C=1
(gdb) c
Continuing.
access granted[Inferior 1 (process 4304) exited normally]
(gdb)
Scanned with CamScanner