Labs- Intro To Assembly Language

Skills Assessment

Task 1

- We are contracting for a company, and they find a suspicious binary file.
  - We examine the file with `gdb` and see that it is loading an encoded shellcode to the Stack and storing the `xor` decoding key in `rbx`.
  - We need to decode the shellcode after it is loaded to the Stack and then run the shellcode to get the flag.

Question

- Disassemble 'loaded_shellcode' and modify its assembly code to decode the shellcode, by adding a loop to 'xor' each 8-bytes on the stack with the key in 'rbx'.

-> We first disassemble the file

```
objdump -M intel --no-show-raw-insn --no-addresses -d loaded_shellcode
```

```
<_start>:
        movabs rax,0xa284ee5c7cde4bd7
        push   rax
        movabs rax,0x935add110510849a
        push   rax
        movabs rax,0x10b29a9dab697500
        push   rax
        movabs rax,0x200ce3eb0d96459a
        push   rax
        movabs rax,0xe64c30e305108462
        push   rax
        movabs rax,0x69cd355c7c3e0c51
        push   rax
        movabs rax,0x65659a2584a185d6
        push   rax
        movabs rax,0x69ff00506c6c5000
        push   rax
        movabs rax,0x3127e434aa505681
        push   rax
        movabs rax,0x6af2a5571e69ff48
        push   rax
        movabs rax,0x6d179aaff20709e6
        push   rax
        movabs rax,0x9ae3f152315bf1c9
        push   rax
        movabs rax,0x373ab4bb0900179a
        push   rax
        movabs rax,0x69751244059aa2a3
        push   rax
        movabs rbx,0x2144d2144d2144d2
```

-> We then write the assembly instructions into an assembly file and replacing `movabs` to the `mov` instructions

```
objdump -M intel --no-show-raw-insn --no-addresses -d loaded_shellcode | grep '_start\|movabs\|push' > shellcode.s

sed -i 's/movabs/mov/g' shellcode.s
```

```
cat shellcode.s
```

```
<_start>:
        mov rax,0xa284ee5c7cde4bd7
        push   rax
        mov rax,0x935add110510849a
        push   rax
        mov rax,0x10b29a9dab697500
        push   rax
        mov rax,0x200ce3eb0d96459a
        push   rax
        mov rax,0xe64c30e305108462
        push   rax
        mov rax,0x69cd355c7c3e0c51
        push   rax
        mov rax,0x65659a2584a185d6
        push   rax
        mov rax,0x69ff00506c6c5000
        push   rax
        mov rax,0x3127e434aa505681
        push   rax
        mov rax,0x6af2a5571e69ff48
        push   rax
        mov rax,0x6d179aaff20709e6
        push   rax
        mov rax,0x9ae3f152315bf1c9
        push   rax
        mov rax,0x373ab4bb0900179a
        push   rax
        mov rax,0x69751244059aa2a3
        push   rax
        mov rbx,0x2144d2144d2144d2
```

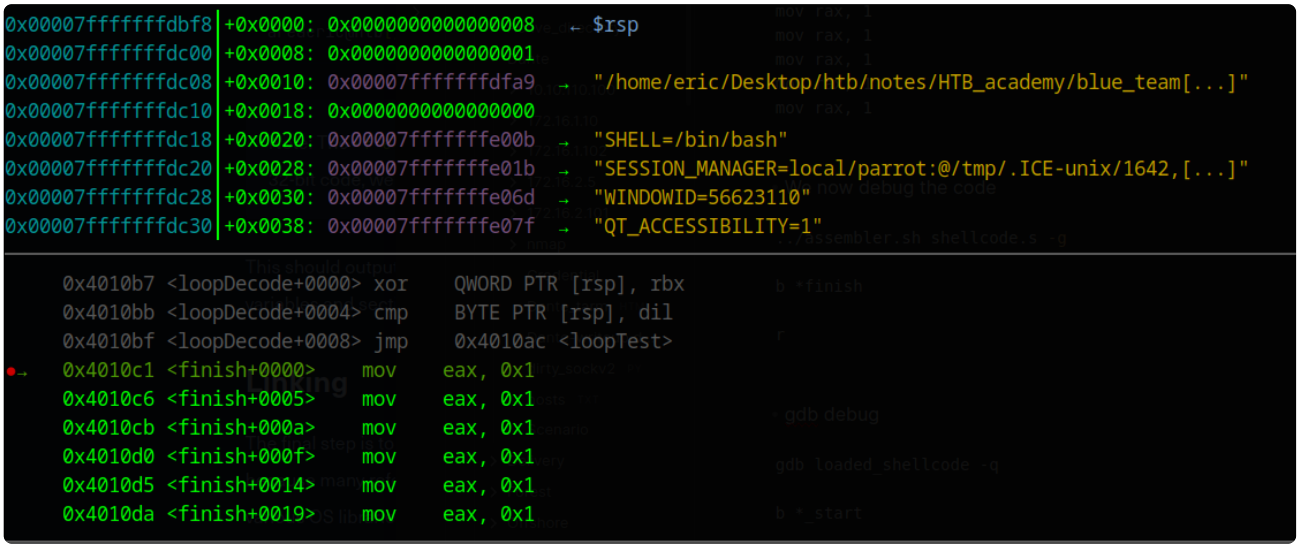
- Now we modify the file as follows

```
# Extra tracker and for pushing rsp registers at the start of file
```

```
mov dil, 8
push rdi

# To be placed after the _start section

initDecode: ; performs initial decoding
xor [rsp], rbx

loopTest:   ; Increments the stack pointer and
                        ; makes comparison on whether it should decode.
add rsp, rdi
cmp [rsp], rdi
jnz loopDecode
jmp finish

loopDecode: ; The loop for decoding
xor [rsp], rbx
cmp [rsp], dil
jmp loopTest

finish: ; We leave this for debugging purposes
mov rax, 1
mov rax, 1
mov rax, 1
mov rax, 1
mov rax, 1
mov rax, 1
mov rax, 1
```

- We now debug the code

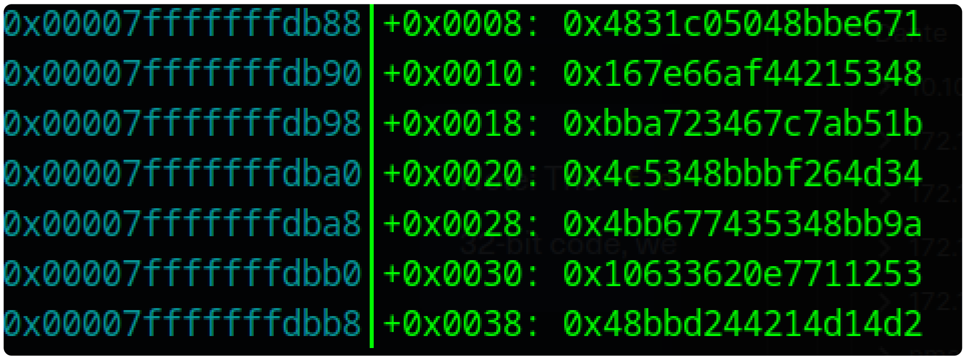
```
../assembler.sh shellcode.s -g

b *finish

r
```

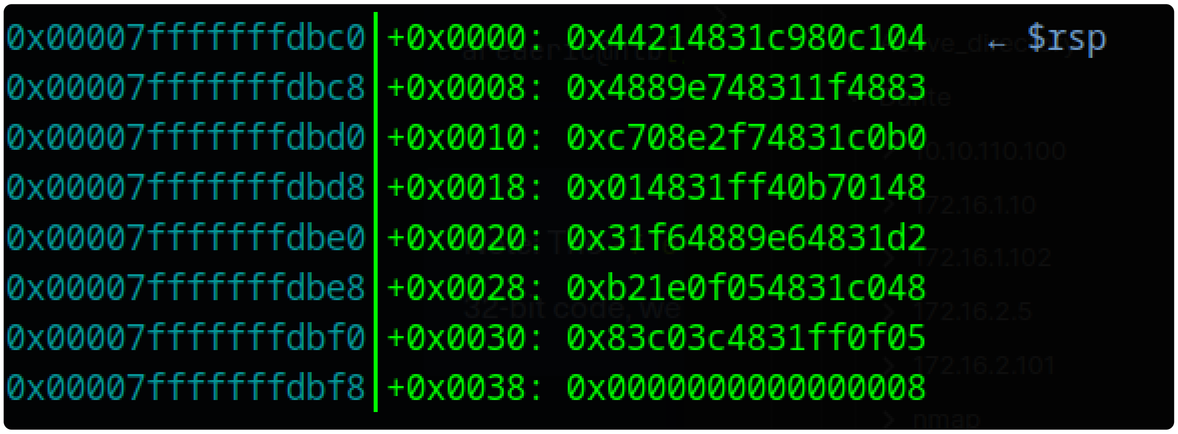


-> We see we are at the bottom of the stack pointer, but we can manually set `$rsp` to move up

- We now record the shellcode

```
set $rsp=0x00007fffffffdb80
si
        -> Record shell code

set $rsp=0x00007fffffffdbc0
si
        -> Record shell code
```





-> Recorded Shellcode

```
0x00007fffffffdb88|+0x0008: 0x4831c05048bbe671
0x00007fffffffdb90|+0x0010: 0x167e66af44215348
0x00007fffffffdb98|+0x0018: 0xbba723467c7ab51b
0x00007fffffffdba0|+0x0020: 0x4c5348bbbf264d34
0x00007fffffffdba8|+0x0028: 0x4bb677435348bb9a
0x00007fffffffdbb0|+0x0030: 0x10633620e7711253
0x00007fffffffdbb8|+0x0038: 0x48bbd244214d14d2
0x00007fffffffdbc0|+0x0000: 0x44214831c980c104
0x00007fffffffdbc8|+0x0008: 0x4889e748311f4883
0x00007fffffffdbd0|+0x0010: 0xc708e2f74831c0b0
0x00007fffffffdbd8|+0x0018: 0x014831ff40b70148
0x00007fffffffdbe0|+0x0020: 0x31f64889e64831d2
0x00007fffffffdbe8|+0x0028: 0xb21e0f054831c048
0x00007fffffffdbf0|+0x0030: 0x83c03c4831ff0f05
```

- Do some style formatting

```
cat decoded_shellcode | awk '{print $2}' | awk -F'0x' '{print$2}' | tr -d \\n
```

```
└──╼ [★]$ cat decoded_shellcode | awk '{print $2}' | awk -F'0x' '{print$2}' | tr -d \\n
4831c05048bbe671167e66af44215348bba723467c7ab51b4c5348bbbf264d344bb677435348
bb9a10633620e771125348bbd244214d14d244214831c980c1044889e748311f4883c708e2f7
4831c0b0014831ff40b7014831f64889e64831d2b21e0f054831c04883c03c4831ff0f05┌─[]
```

- Lastly, we run the shellcode

```
python ../loader.py
'4831c05048bbe671167e66af44215348bba723467c7ab51b4c5348bbbf264d344bb6774
35348bb9a10633620e771125348bbd244214d14d244214831c980c1044889e748311f488
3c708e2f74831c0b0014831ff40b7014831f64889e64831d2b21e0f054831c04883c03c4
831ff0f05'
```

```
└──╼ [★]$ python ../loader.py '4831c05048bbe671167e66af44215348bba723467c7ab
51b4c5348bbbf264d344bb677435348bb9a10633620e771125348bbd244214d14d244214831c
980c1044889e748311f4883c708e2f74831c0b0014831ff40b7014831f64889e64831d2b21e0
f054831c04883c03c4831ff0f05'
HTB{4553mbly_d3bugg1ng_m4573r}$
```

Task 2

- The above server simulates a vulnerable server that we can run our shellcodes on. Optimize 'flag.s' for shellcoding and get it under 50 bytes, then send the shellcode to get the flag. (Feel free to find/create a custom shellcode)

-> We first examine the number of bytes that the current shellcode is taking up.

-> Assemble

```
./assembler.sh flag.s
```

```
python shellcoder.py flag
```

```
└──╼ [★]$ python shellcoder.py flag
6a0048bf2f666c672e74787457b8020000004889e7be000000000f05488d374889c7b800000000ba180000000f05b8
01000000bf01000000ba180000000f05b83c000000bf000000000f05
75 bytes - Found NULL byte
```

-> We see there are also null bytes, so we will have to remove that as well.

- The modifications are listed below:

```
global _start$
$
section .text$
_start:$
    ; push './flg.txt\x00'$
    xor rsi, rsi$
    push rsi                ; push NULL string terminator$
    mov rdi, '/flg.txt' ; rest of file name$
    push rdi              ; push to stack+$
++++$
    ; open('rsp', 'O_RDONLY')$
    mov al, 2           ; open syscall number$
    mov rdi, rsp          ; move pointer to filename$
    ;mov rsi, 0           ; set O_RDONLY flag$
    syscall$
$
```

-> We first xor'd the rsi register that we need to push 0 to and commented out the `mov rsi, 0` as it is already 0 by default.
-> we also change the register to 8 bit register `al` instead of 64 bits register `rax`

```
    ; read file$
    lea rsi, [rdi]      ; pointer to opened file$
    mov rdi, rax        ; set fd to rax from open syscall$
    xor rax, rax          ; read syscall number$
    mov dl, 24         ; size to read$
    syscall$
$
    ; write output$
    mov al, 1          ; write syscall$
    mov dil, 1          ; set fd to stdout$
    ;mov dl, 24         ; size to read$
    syscall$
$
    ; exit$
    mov al, 60$
    xor rdi, rdi$
    ;mov rdi, 0$
    syscall$
$
```

-> Similarly, we xor'd `rax` as we need to make it 0 and used the 8-bit register `dl` instead of its 64-bit version of `rdx`
-> We repeat using 8-bit register for the write output section and also removed the duplicate code of `mov dl, 24` which was done in the `; read file` section
-> Lastly, we performed xor on `rdi` instead of moving a 0 into the register.

- The shell code we constructed had 49 bytes, just one byte under the required byte of 50

```
└──╼ [★]$ python3 ../shellcoder.py flag2

4831f65648bf2f666c672e74787457b0024889e70f05488d374889c74831c0b2180f05
b00140b7010f05b03c4831ff0f05
49 bytes - No NULL bytes
```

- We then pass our shell-code to the vulnerable server get the flag

```
└──╼ [★]$ nc 94.237.62.124 53237
4831f65648bf2f666c672e74787457b0024889e70f05488d374889c74831c0b2180f05
b00140b7010f05b03c4831ff0f05
HTB{5h3llc0d1ng_g3n1u5}
```

Architecture

Assembly Language

Question

In the 'Hello World' example, which Assembly instruction will '00001111 00000101' execute?

- We see the assembly intruction for hello world is as follows:

Code: nasm

```nasm
mov rax, 1
mov rdi, 1
mov rsi, message
mov rdx, 12
syscall

mov rax, 60
mov rdi, 0
syscall
```

- For binary it is as follows

Code: binary

```binary
01001000 11000111 11000000 00000001
01001000 11000111 11000111 00000001
01001000 10001011 00110100 00100101
01001000 11000111 11000010 00001101
00001111 00000101

01001000 11000111 11000000 00111100
01001000 11000111 11000111 00000000
00001111 00000101
```

- So, the mapping of '00001111 00000101' would be mapped to `syscall`

Registers, Addresses, and Data Types

Question

What is the 8-bit register for 'rdi'?

- We see from the table that it would be dil

Description	64-bit Register	32-bit Register	16-bit Register	8-bit Register
Data/Arguments Registers				
Syscall Number/Return value	`rax`	`eax`	`ax`	`al`
Callee Saved	`rbx`	`ebx`	`bx`	`bl`
1st arg - Destination operand	`rdi`	`edi`	`di`	`dil`

- We can achieve the same result using the rule for sub registers:

Size in bits	Size in bytes	Name	Example
`16-bit`	`2 bytes`	the base name	`ax`
`8-bit`	`1 bytes`	base name and/or ends with `l`	`al`
`32-bit`	`4 bytes`	base name + starts with the `e` prefix	`eax`
`64-bit`	`8 bytes`	base name + starts with the `r` prefix	`rax`

-> Where we have base name of `rdi= di` and `l` appended to it, resulting in `dil`.

Assembling & Debugging

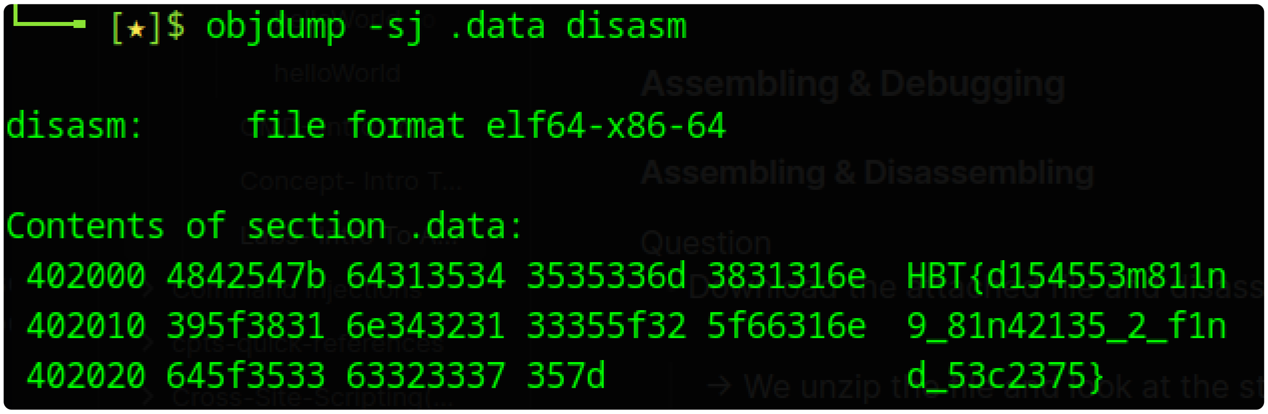
Assembling & Disassembling

Question

- Download the attached file and disassemble it to find the flag

-> We unzip the file and look at the strings of the .text section of the code

```
unzip disasm.zip

objdump -sj .data disasm
```



Debugging with GDB

Question

- Download the attached file, and find the hex value in 'rax' when we reach the instruction at <_start+16>?
  -> Set breakpoint at _start+16 then examine the register rax with hex value
- Starting debugger with gdb

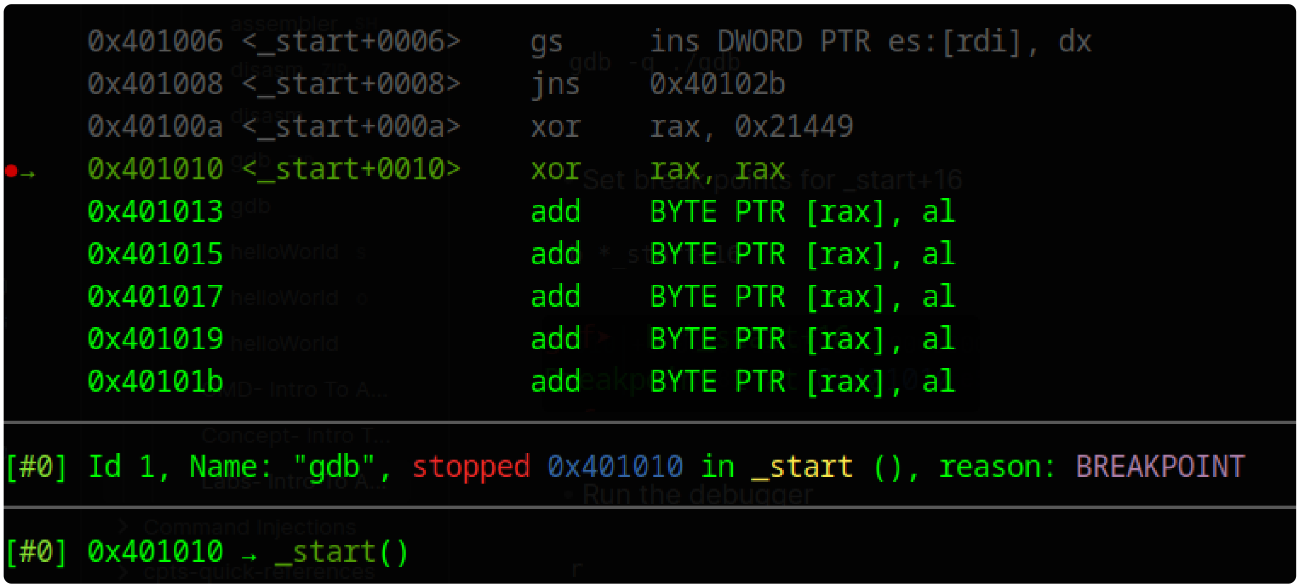
```
gdb -q ./gdb
```

- Set break points for _start+16

```
b *_start+16
```

- Run the debugger

```
r
```



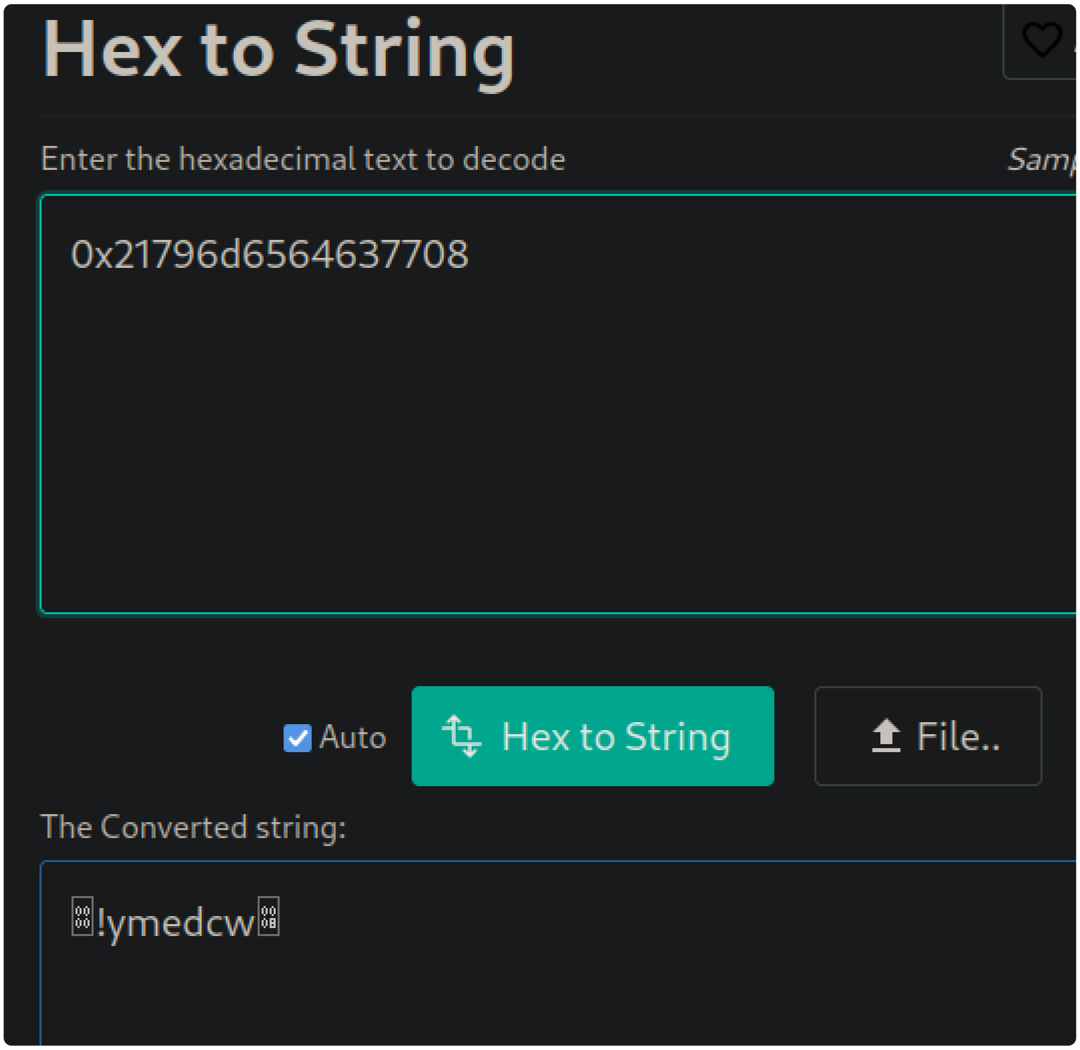
- Examining the values of registers

```
registers
```

-> Found the required value 0x21796d6564637708

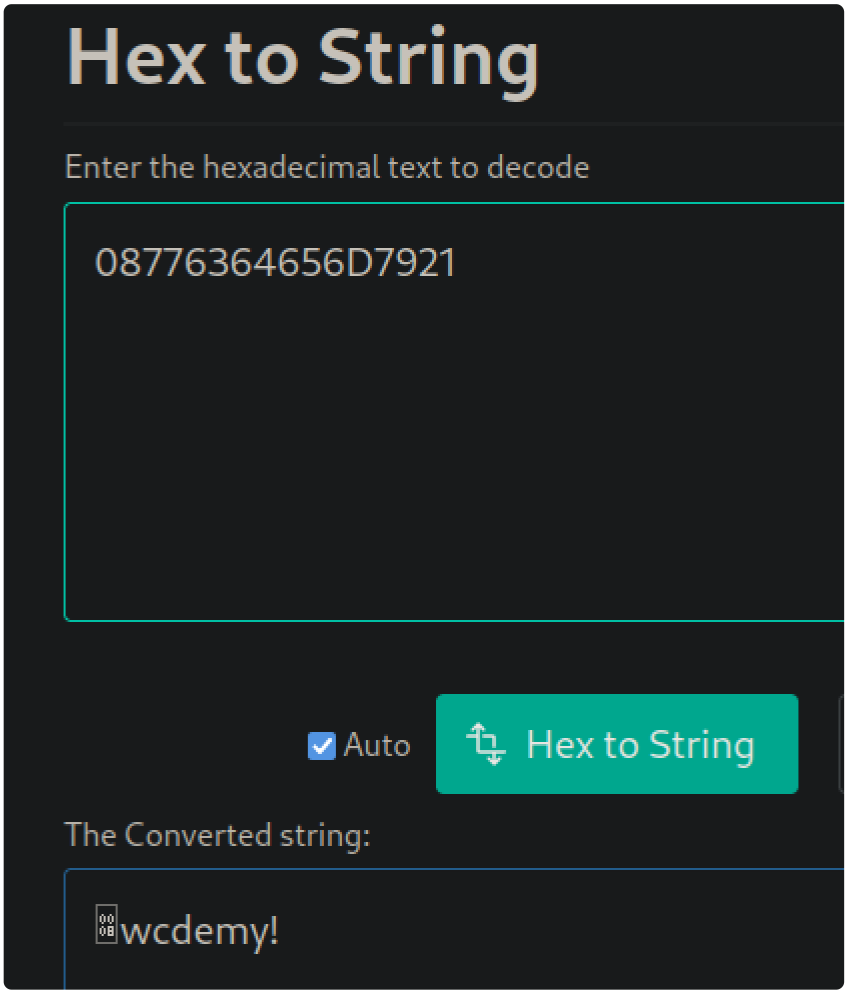
-> Note: If we try stuff like

`x/xb $rax`, we would get cannot access memory, but we would still get the hex value.



-> We can still get the string value, but the string value above is wrong due to little endian.

-> We will need to change reverse the order of bytes to account for endianess of processor.



Basic Instructions

Data Movement

Question

- Add an instruction at the end of the attached code to move the value in "rsp" to "rax". What is the hex value of "rax" at the end of program execution?

-> Add move instruction value of `rsp` to `rax` to the end of the code given

```
mov rax, [rsp]
```

```
global _start$
$
section .text$
_start:$
    mov rax, 1024$
    mov rbx, 2048$
    xchg rax, rbx$
    push rbx$
    mov rax, [rsp]$
```

-> Assemble, link and run the code in gdb

```
./assembler.sh mov.s -g
```

```
└──╼ [★]$ ./assembler.sh mov.s -g
GEF for linux ready, type `gef' to start, `gef config' to configure
89 commands loaded and 5 functions added for GDB 13.1 in 0.00ms using Python engine 3.11
Reading symbols from mov...
(No debugging symbols found in mov)
gef➤
```

-> Breaking at start, running the program, stepping through it and examining the value of registers

```
break _start
r
- Repeated excution si
```

```
$rax   : 0x400
$rbx   : 0x400
$rcx   : 0x0
$rdx   : 0x0
$rsp   : 0x00007fffffffdc28  →  0x0000000000000400
$rbp   : 0x0
$rsi   : 0x0
$rdi   : 0x0
$rip   : 0x0000000000401011  →   add BYTE PTR [rax], al
```

Arithmetic Instructions

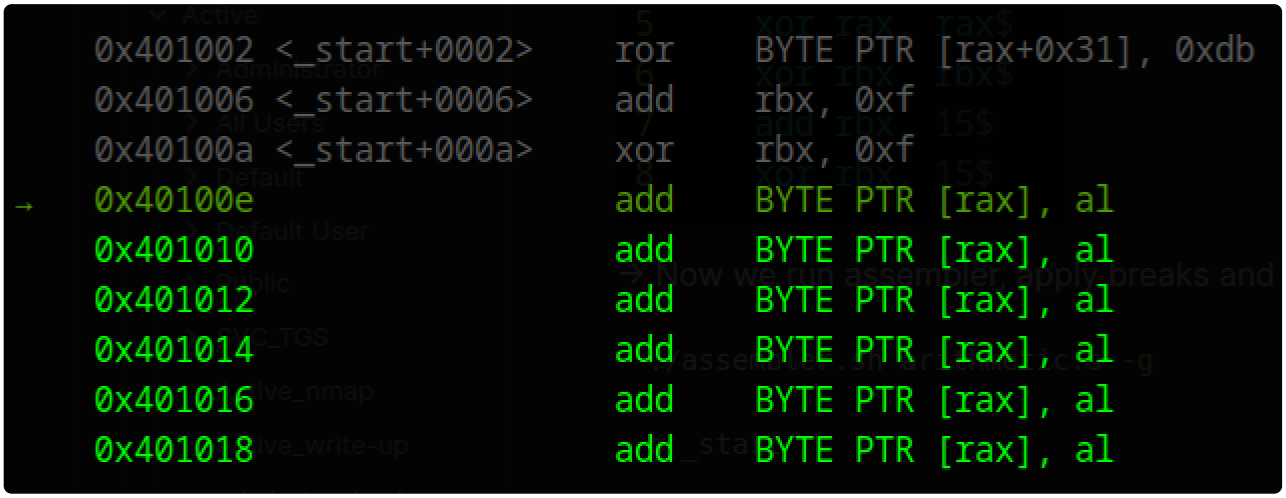
Questions

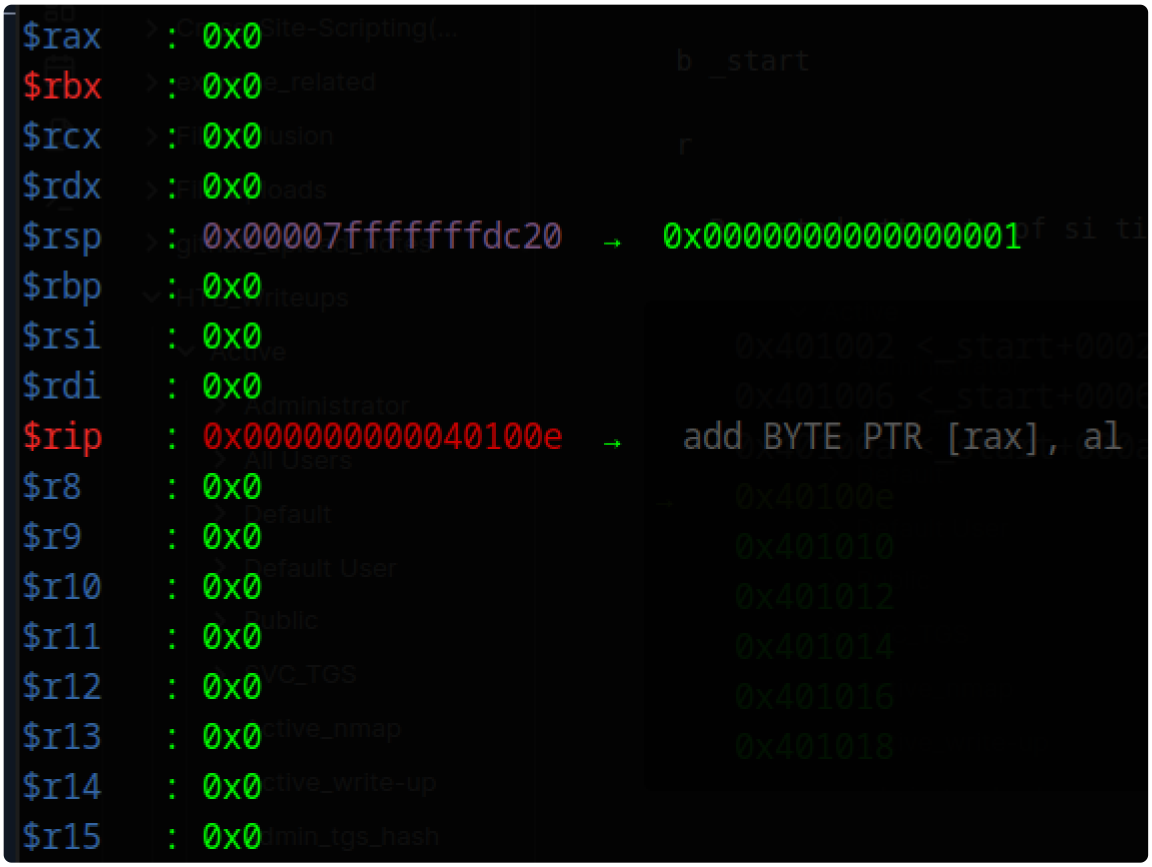
- Add an instruction to the end of the attached code to "xor" "rbx" with "15". What is the hex value of 'rbx' at the end?
  -> We add the instructions to the attached code:

```
global _start$
$
section .text$
_start:$
    xor rax, rax$
    xor rbx, rbx$
    add rbx, 15$
    xor rbx, 15$
```

  -> Now we run assembler, apply breaks and see the result

```
./assembler.sh arithmetic.s -g

b _start

r

- Repeated attempts of si till the end of the code
```





-> We see the value of `rbx` is `0x0`

Control Instructions

Loops

Question

- As we can see, we have successfully used loops to automate the calculation of the Fibonacci Sequence. Try increasing `rcx` to see what are the next numbers in the Fibonacci Sequence.

-> Examine the code:

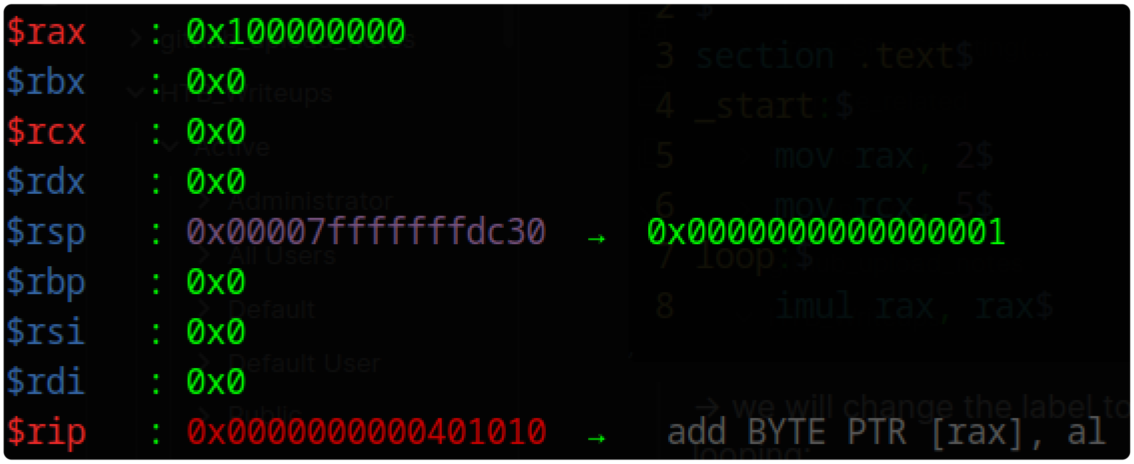
```
global _start$
$
section .text$
_start:$
    mov rax, 2$
    mov rcx, 5$
loop:$
    imul rax, rax$
```

-> we will change the label to `_loop` and add the line `loop _loop` to perform looping:

```
global _start$
$
section .text$
_start:$
    mov rax, 2$
    mov rcx, 5$
_loop:$
    imul rax, rax$
    loop _loop$
```

- Then we confirm our results by assembling the code and examining with debugger

```
./assembler.sh loops -g

b _loop

r

- Repeated si command
```



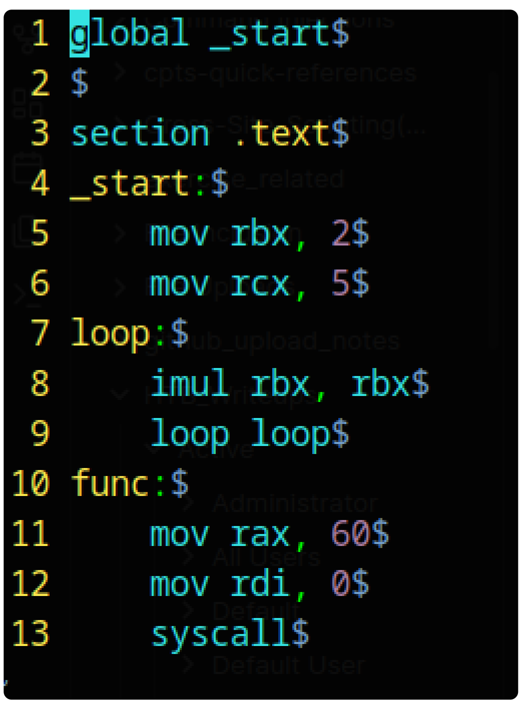
-> We get final result of `$rax=0x100000000`

Unconditional Branching

Question

- Try to jump to "func" before "loop loop". What is the hex value of "rbx" at the end?
  -> We examine the code first

```
vim unconditional.s
```



-> We would add the piece of code `jmp func` at line 9 while pushing `loop loop` back a line.

```
global _start$
$
section .text$
_start:$
    mov rbx, 2$
    mov rcx, 5$
loop:$
    imul rbx, rbx$
    jmp func$
    loop loop$
func:$
    mov rax, 60$
    mov rdi, 0$
    syscall$
```

- We run the assembler and examine the code at the end

```
./assembler.sh unconditional.s -g

b loop

r

- Repeat si

- Examine results of registers at the end
```

```
$rax   : 0x3c
$rbx   : 0x4
$rcx   : 0x5
$rdx   : 0x0
$rsp   : 0x00007fffffffdc20  →  0x0000000000000001
$rbp   : 0x0
$rsi   : 0x0
$rdi   : 0x0
$rip   : 0x000000000040101c  →  <func+000a> syscall
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0
$r13   : 0x0
$r14   : 0x0
$r15   : 0x0
```

-> We get `$rbx=0x4`

Conditional Branching

Question

- The attached assembly code loops forever. Try to modify (mov rax, 5) to make it not loop. What hex value prevents the loop?
  -> We first examine the code

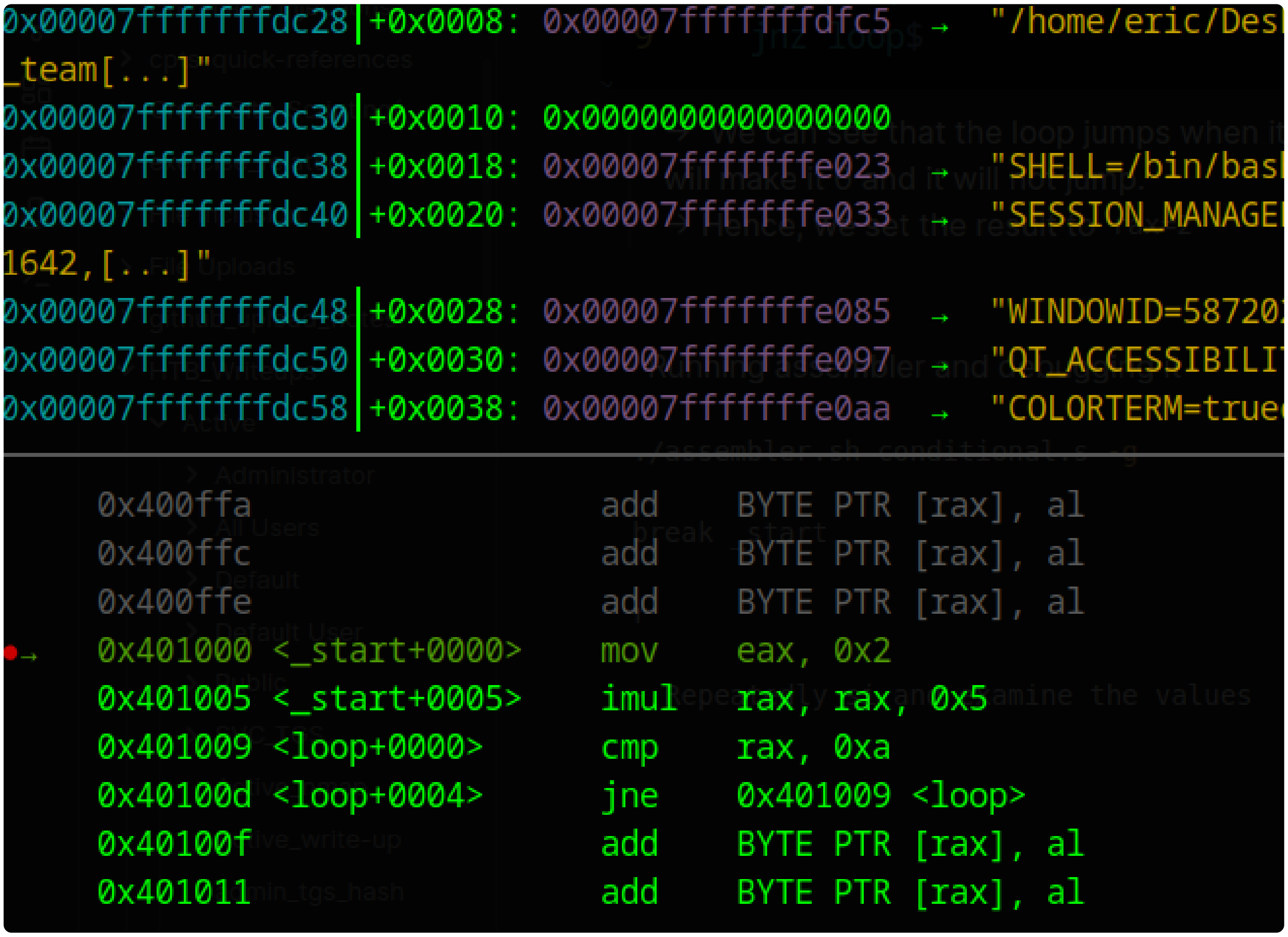
```
vim conditional.s
```

```
global _start$
$
section .text$
_start:$
    mov rax, 5       ; change here$
    imul rax, 5$
loop:$
    cmp rax, 10$
    jnz loop$
```

-> We can see that the loop jumps when its not equal to 0, so setting `rax` to 10 will make it 0 and it will not jump.
-> Hence, we set the result to `rax=2`

- Running assembler and debugging it

```
./assembler.sh conditional.s -g

break _start

r

- Repeatedly si and examine the values
```



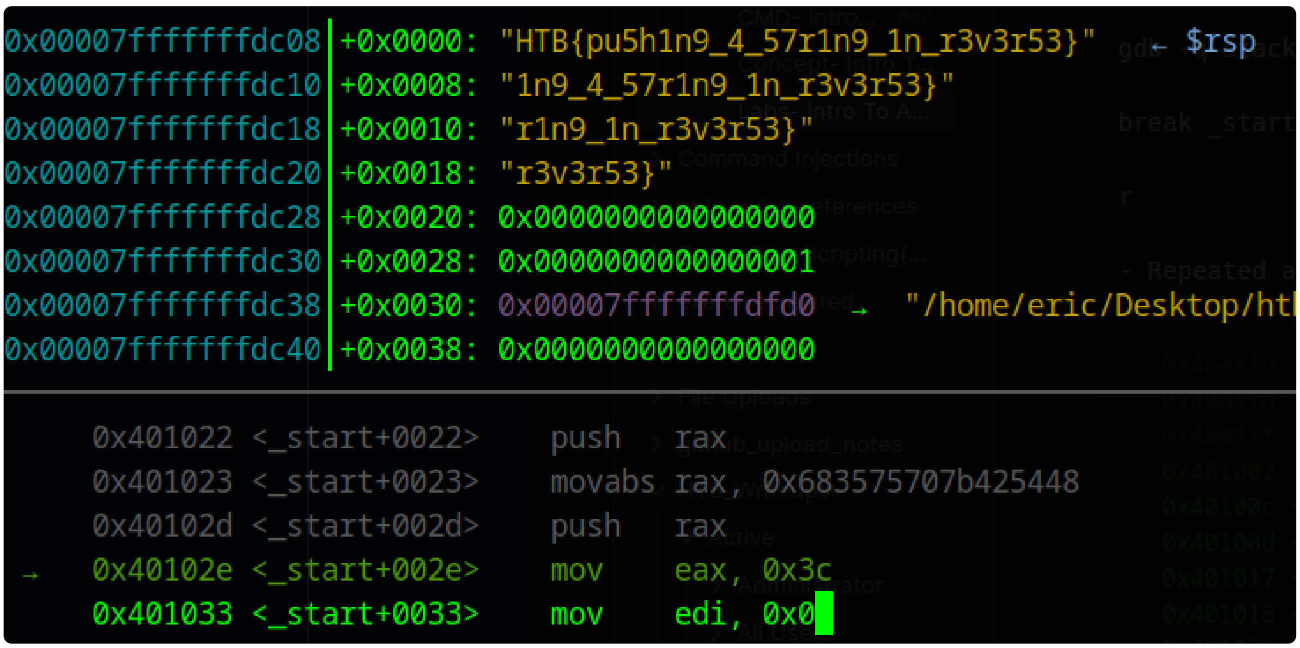
-> We see that the hex value of `0x2` is required

Using the Stack

Question

- Debug the attached binary to find the flag being pushed to the stack
  -> We assemble the code and debug it.

```
gdb -q stack

break _start

r

- Repeated attempt of si and examine the code
```



-> We obtain the flag required.

Syscalls

Question

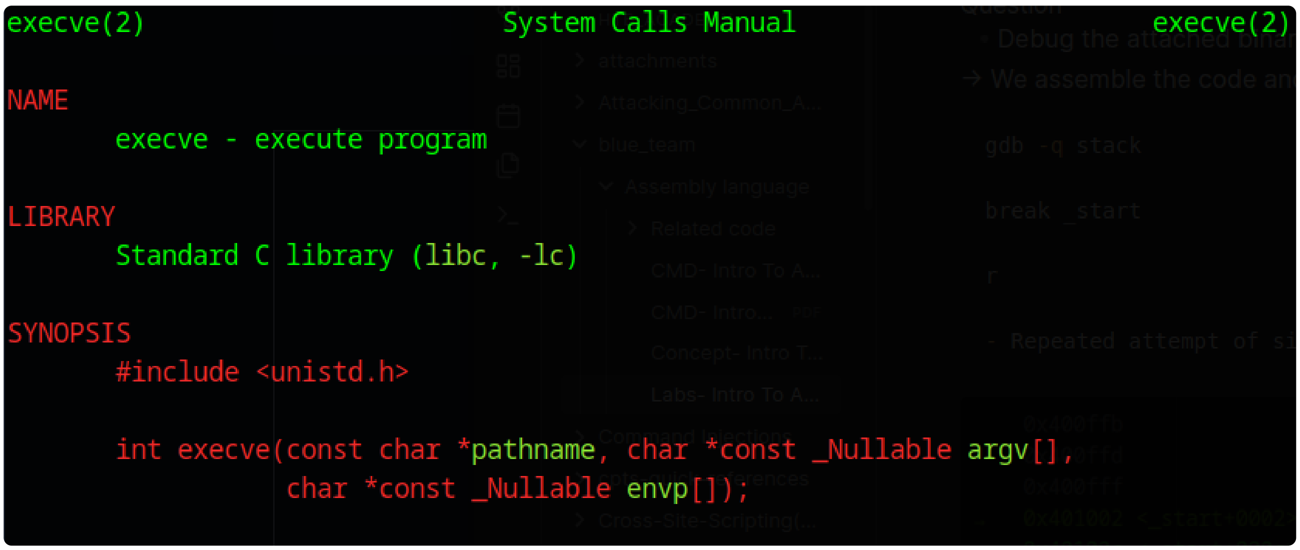
- What is the syscall number of "execve"?
  -> We grep for it for the related system file

```
cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep execve
```

-> syscall number of 59

- How many arguments does "execve" take?
  -> We look at the man page of execve

```
man -s 2 execve
```



-> We see it takes 3 arguments.

Procedures

Question

- Try assembling and debugging the above code, and note how "call" and "ret" store and retrieve "rip" on the stack. What is the address at the top of the stack after entering "Exit"? (6-digit hex 0xaddress, without zeroes)
  -> We assemble the code and debug it accordingly

```
./assembler.sh fib.s -g

b Exit

r
```

```
- Repeated si and examining the results.
```

```
$rax   : 0x3c
$rbx   : 0xd
$rcx   : 0x000000000040102f  →  <printMessage+001b> ret
$rdx   : 0x14
$rsp   : 0x00007fffffffdc28  →  0x0000000000401014  →  <printMessage+0000> mov eax, 0x1
$rbp   : 0x0
$rsi   : 0x0000000000402000  →  "Fibonacci Sequence:\n"
$rdi   : 0x1
```

-> So `rsp` has a value of `0x401014` and points to the first line of `printMessage`

Functions

Question

- Try to fix the Stack Alignment in "print", so it does not crash, and prints "Its Aligned!". How much boundary was needed to be added? "write a number"
  -> We first examine the file

```
vim functions.s
```

```
global  _start$
extern  printf$
$
section .data$
    outFormat db  "It's %s", 0x0a, 0x00$
    message db "Aligned!", 0x0a$
$
section .text$
_start:$
    call print          ; print string$
    call Exit           ; Exit the program$
$
print:$
    mov rdi, outFormat  ; set 1st argument (Print Format)$
    mov rsi, message    ; set 2nd argument (message)$
    call printf         ; printf(outFormat, message)$
    ret$
$
Exit:$
    mov rax, 60$
    mov rdi, 0$
    syscall$
```

-> We see that there is only 8 bytes pushed on top, not abiding to the 16 byte rule for stack pointers.
-> we could add the following

```
sub rsp, 8
call printf
add rsp, 8
```

```
global  _start$
extern  printf$
$
section .data$
    outFormat db  "It's %s", 0x0a, 0x00$
    message db "Aligned!", 0x0a$
$
section .text$
_start:$
    call print          ; print string$
    call Exit           ; Exit the program$
$
print:$
    mov rdi, outFormat  ; set 1st argument (Print Format)$
    mov rsi, message    ; set 2nd argument (message)$
    sub rsp, 8$
    call printf         ; printf(outFormat, message)$
    add rsp, 8$
    ret$
$
Exit:$
    mov rax, 60$
    mov rdi, 0$
    syscall$
functions.s [+]
:wq
```

-> Then, running assembling and running the code, we see

```
nasm -f elf64 functions.s &&  ld functions.o -o functions -lc --dynamic-linker /lib64/ld-linux-x86-64.so.2 && ./functions
```

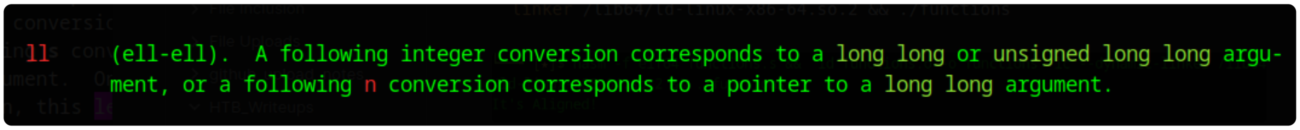
```
└──╼ [★]$ nasm -f elf64 functions.s &&  ld functions.o -o functions -lc --dynamic-linker /lib64/ld-linux-x86-64.so.2 && ./functions
It's Aligned!
```

Libc Functions

Question

- The current string format we are using only allows numbers up to 2 billion. What format can we use to allow up to 3 billion? "Check length modifiers in the 'printf' man page"
  -> We examine the man page

```
man -s 3 printf
```

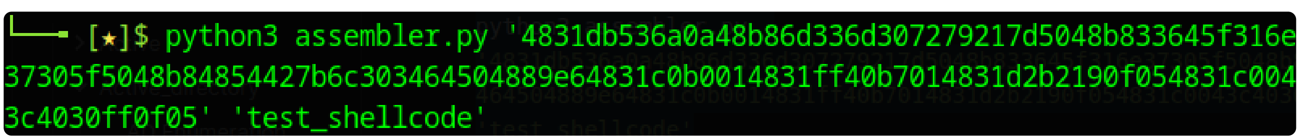


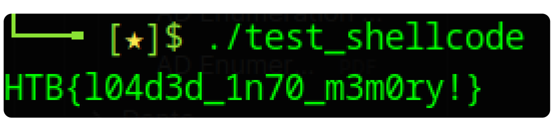
-> We would want to have the length modifier ll (largest unsigned int length modifier), so we would have a final format of `%lld`

Exercise

- Run the "Exercise Shellcode" to get the flag.
- Exercise Shellcode:
  4831db536a0a48b86d336d307279217d5048b833645f316e37305f5048b84854427b6c303464504889e64831c0b0014831ff40b7014831d2b2190f054831c0043c4030ff0f05
  -> We compile it with the assembler and run it:

```
python3 assembler.py
'4831db536a0a48b86d336d307279217d5048b833645f316e37305f5048b84854427b6c303464504889e64831c0b0014831ff40b7014831d2b2190f054831c0043c4030ff0f05'
'test_shellcode'

./test_shellcode
```





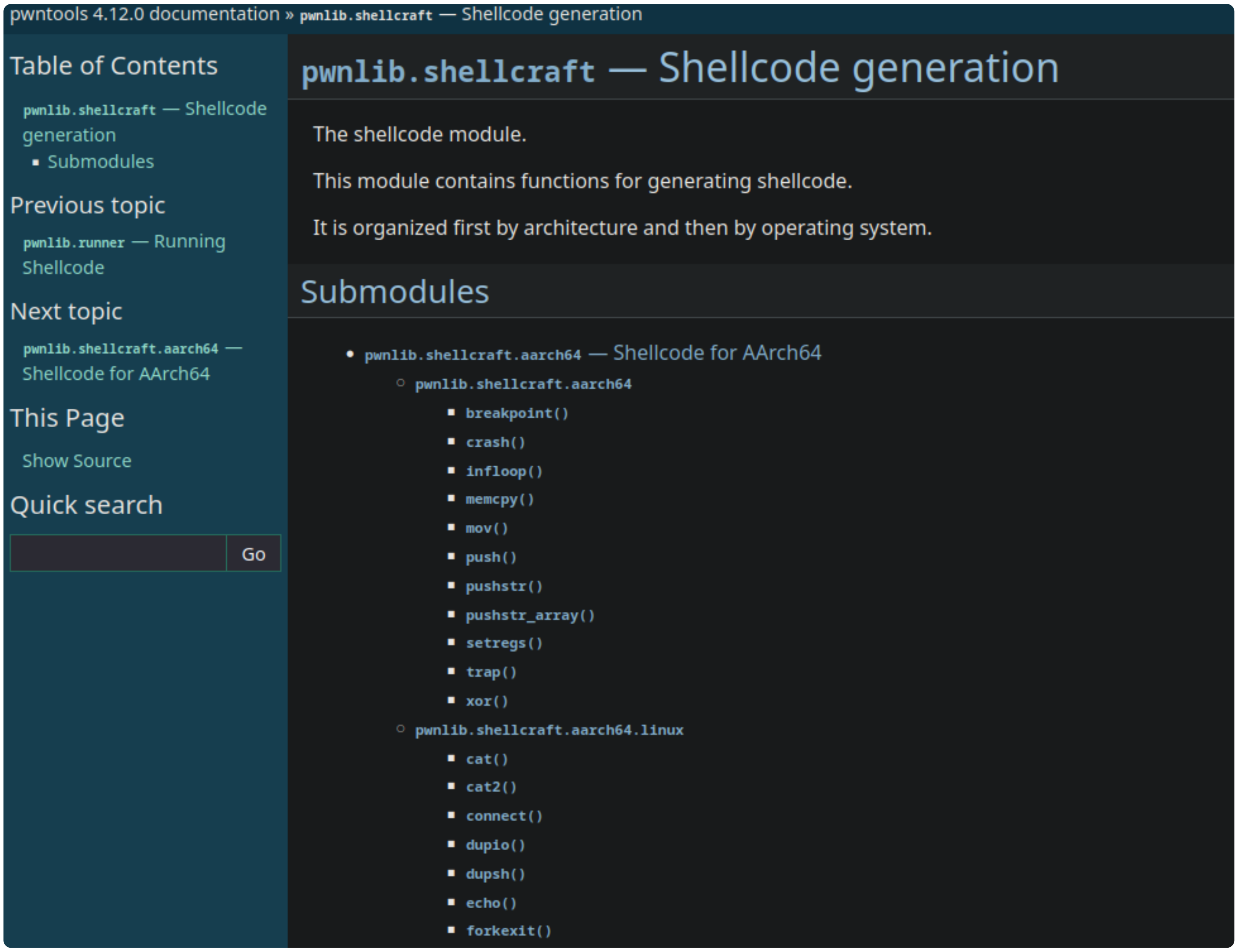
Shellcoding Tools

Question

- The above server simulates an exploitable server you can execute shellcodes on. Use one of the tools to generate a shellcode that prints the content of '/flag.txt', then connect to the sever with "nc SERVER_IP PORT" to send the shellcode.

-> we want to execute the command `cat /flag.txt`

-> We first looked up at the documentation for shellcraft



-> We examine the `cat` and `cat2` command

pwnlib.shellcraft.aarch64.linux

`pwnlib.shellcraft.aarch64.linux.cat(filename, fd=1)` [source]

Opens a file and writes its contents to the specified file descriptor.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'This is the flag\n')
>>> shellcode = shellcraft.cat(f) + shellcraft.exit(0)
>>> run_assembly(shellcode).recvline()
b'This is the flag\n'
```

`pwnlib.shellcraft.aarch64.linux.cat2(filename, fd=1, length=16384)` [source]

Opens a file and writes its contents to the specified file descriptor. Uses an extra stack buffer and must know the length.

Example

```
>>> f = tempfile.mktemp()
>>> write(f, 'This is the flag\n')
>>> shellcode = shellcraft.cat2(f) + shellcraft.exit(0)
>>> run_assembly(shellcode).recvline()
b'This is the flag\n'
```

`pwnlib.shellcraft.aarch64.linux.connect(host, port, network='ipv4')` [source]

Connects to the host on the specified port. Network is either 'ipv4' or 'ipv6'. Leaves the connected socket in x12.

`pwnlib.shellcraft.aarch64.linux.dupio(sock='x12')` [source]

Args: [sock (imm/reg) = x12] Duplicates sock to stdin, stdout and stderr

`pwnlib.shellcraft.aarch64.linux.dupsh(sock='x12')` [source]

Args: [sock (imm/reg) = x12] Duplicates sock to stdin, stdout and stderr and spawns a shell.

`pwnlib.shellcraft.aarch64.linux.echo(string, sock='1')` [source]

Writes a string to a file descriptor

-> We see that the `cat` command is suitable, so we try it out first.

-> Here we start to craft our shell code

```
python3
from pwn import *
context(os='linux', arch="amd64", log_level="error")
dir(shellcraft)

f= tempfile.mktemp()
write(f, '/flag.txt')
shellcode = shellcraft.cat('/flag.txt') + shellcraft.exit(0)
run_assembly(shellcode).recvline()
asm(shellcode).hex()
```

```
>>> shellcode = shellcraft.cat('/flag.txt') + shellcraft.exit(0)
>>> run_assembly(shellcode).recvline()
b'random\n'
>>> asm(shellcode).hex()
'6a7448b82f666c61672e7478506a02584889e731f60f0541baffffff7f4889c66a28586a015f990f0531ff6a3c580f05'
```

-> Where we created an `/flag.txt` file in our host with the file content `random\n`

- To verify, we also try it with loader.py

```
python3 loader.py
'6a7448b82f666c61672e7478506a02584889e731f60f0541baffffff7f4889c66a28586
a015f990f0531ff6a3c580f05'
```

```
└──╼ [★]$ python3 loader.py '6a7448b82f666c61672e7478506a02584889e731f60f0541baffffff7f4889c66a28586a015f990f0531f
f6a3c580f05'
random
```

-> We see it works, so we will send it through nc

-> We send our shell code through and obtain the flag

```
nc 83.136.249.173 51175
```

```
└──╼ [★]$ nc 83.136.249.173 51175
6a7448b82f666c61672e7478506a02584889e731f60f0541baffffff7f4889c66a28586a015f990f0531ff6a3c580f05
HTB{r3m073_5h3llc0d3_3x3cu710n}
```