→ Your-first Register:

```
mov rax, 60
```

Save it as assembly-01.s

run it with `/challenge/check /assembly-01.s' as an argument

gets the flag, but program crashes.

→ Your-First Syscall:

Syscalls are total of 300 types in linux; identified by particular Number, such as '42'

```
mov rax, 42
syscall
```

Save as assembly-02.s

Now, we learnt our first Sys call 'exit'.

'exit' is a syscall that causes program to exit succesfully.

'exit' has syscall no. '60'.

```
mov rax, 60
syscall
```

Save it as assemby-02.s

run it with /challenge/check as argument.

Get flag, without crashing.

→ Exit Codes:

every code exits [redacted] with an exit code as it terminates.

This is done by 'exit'.

Parameters can be passed through registers to the syscalls.

Though 'exit' has only one parameter 'exit code', multiple parameters ~~can~~ can be passed through syscalls.

EXPERIMENT : No. Page No. Date

```
mov rdi, 42
mov rax, 60
syscall
```

Here, we set 'rdi' to '42', then set 'rax' to '60' and called it as 'syscall' 'exit'; the program exits due to exit code '60' but exits with code '42' printed.

→ Building Executables:

Until now, we have only written codes, but the executables are already provided by 'pwn.college'; Now we have to build executable to actually make CPU run the program.

Step-1: write program and name it as 'assembly-04.s

```
.intel_syntax noprefix
.global _start
_start:
mov rdi, 42
mov rax, 60
syscall
```

Here .intel_syntax noprefix states that the syntax used is of 'intel' and with 'no prefixes';

The .global _start states that the program starts from '_start:' and is globally executable.

Step-2: create an object file containing binary executable about the assembly-04.s, which is runnable by CPU.

Teacher's Sign. :

as -o ~~assembly~~-04.o assembly_04.s

'as' is assembly checker or compiler - and '.o' file is object file

<u>Step-3</u>> link the object file with a legitamate program file here 'assembly-04'. using 'ld' command.

ld -o assembly-04 assembly-04.o

<u>step-4</u>> Now executable it and pass it as argument through /challenge /check

Moving Between Registers

we can move registers into registers or values stored in it.

we write program;

```
.intel-syntax noprefix
.global -start
-start:
mov rdi, rsi
mov rax, 60
syscall.
```

The /challenge /check will set 'rsi' a secret value and that value will be set to 'rdi' using 'mov rdi, rsi'

finally we get flag.

→ Tracing Syscalls:

'strace' is used to trace any syscall.
If we run strace with our Program as arg; Strace will provide systemcall for everyline in program and from where it is being called or stored.

It's syntax ~~of~~ output is 'System_call (Parameter, Parameter, ...'
just like function syntax of 'C' language.
just like 'exit' has '60' ~~on~~ syscall, 'alarm' is '37' and is also a syscall.

run /challenge/trace-me with strace and analyze it's systemcalls; look for value it passes through parameter for syscall 'alarm' and give that value as argument to /challenge/submit-number.

EXPERIMENT : No. Date

→ Starting GDB:-

GDB stands for 'GNU Debugger' and is used to hunt down and understand bugs.

Debugger is a tool which helps understand and closely monitor or introspects any process.

Here all processing of gdb output is done by 'pwn.college'.

just take the secret no. and pass it as argument through /challenge/submit-number

$> gdb /challenge/debug-me.

$> /challenge/submit-number

→ Starting Programs in GDB:-

$ gdb program-name

$ gdb starti

here the program will run in gdb, then we have to give command 'starti' to gdb to start debugging.

$> gdb /challenge/debug-me

$> gdb starti

got the number.

$> /challenge/submit-number

Teacher's Sign. :

Computer Memory

SHORT-TERM gets data from LONG TERM → slow but permanent

short term when completes execution to store

Registers/cache → Actual where data is stored

gets data from longterm whenneeded

FAST but temporary

For more info. "JUST READ THE PPT INTO THE MODULE

→ Loading From Memory:

Computers store data, mostly sequentially in memory; where, the memory accessed can be mapped and the idle or memory that is free for now, is not mapped.

we are going to access stored data in memory.

Let us consider this as a chunk of memory.

Address	contents
31337	42

here we can access the memory address for contents as: mov rdi, [31337] where '[ ]' is used to denote address.

This memory [31337] points to the address in memory where the content is stored.

```
.intel_syntax noprefix
.global _start
_start:
 mov rdi, [133700]
 mov rax, 60
 syscall
```

→ More Loading Practice:

Perform same steps as above:

```
.intel-syntax noprefix
.global _start
_start:
 mov rdi, [123400]
 mov rax, 60
 syscall
```

→ Deferencing Pointers:

we can move the memory address into an general purpose register

eg: mov rax, 133700, Now:

Address	Contents
133700	42

Stored address

Register	Contents
rax	133700

Now, address is stored in 'rax' and can be used as reference to or point to memory address.

Now, mov rdi, [rax], will perform same operation as
mov rdi, [133700]

Deferencing is changing reference from one register to another
eg: mov rdi, [rax].

```
.intel-syntax noprefix
.global _start
_start:
  mov rdi, [rax]
  mov rad, 60
  syscall
```

→ Deferencing Yourself:

we can deference ourself or deference same register.

eg: mov [133700], 42 → 42 is stored at [133700]
mov rax, 133700 → rax stores address '133700'.
mov rax, [rax] → Now, rax will have '42'

Now, we have to:

```
.intel-syntax noprefix
.global _start
_start:
  mov rdi, [rdi]
  mov rax, 60
  syscall
```

→ Deferencing with Offsets:

Sometimes, pointers might not directly point towards data in memory, rather might point to collection of data (eg: entire book).

Teacher's Sign. : ______

we need to provide an offset to refer only particular of data we need; from collection.

Address	Contents
133700	50
133701	42
133702	99
133703	14

Register	Contents
rdi	133700

we can do this by giving offset 'eg:' '+1' for first, '+2' for second, etc.

mov rax, [rdi+1]

The addition by '+1' Memory address is '+1' byte and we call this '1 byte difference as an offset'.

Now:

```
.intel_syntax noprefix
.global _start
_start:
mov rdi, [rdi+8]
mov rax, 60
syscall
```

Save as assembly-10.s and perform same steps as above challenge.

address +8 offset value is stored in rdi.

→ Stored Addresses:

we can store address of an memory address into a register and store that register's address to another.

eg:
mov rdi, 123400 → rdi store 123400 address.
mov rdi, [rdi] → rdi becomes value stored at 123400 (which is 133700)
mov rax, [rdi] → we deference [rdi], reading 42 into rax.

Now;

```
.intel_syntax noprefix
.global _start
_start:
 mov rdi, [567800]
 mov rdi, [rdi]
 mov rax, 60
 syscall.
```

Save as assemby11.s and Perform all steps as above challenge

→ Double Dereference:

Now, we have secret_value stored at memory address of secret_location1 which address is stored at secret_location_2 and secret_location2 is stored in rax.

Address	Contents
133700	123400
123400	42

register	contents
rax	133700

EXPERIMENT : No. Page No. Date

Now;

```
.intel-syntax noprefix
.global _start
_start:
    mov rdi, [rax]
    mov rdi, [rdi]
    mov rax, 60
    syscall
```

Save as assembly_12.s and perform same steps as above challenge

→ Tripple Dereferencing:

Now, we have value ~~addr~~ stored in addr-1, addr-1 stored in addr-2 and addr-2 stored in rdi.

So we:

```
.intel-syntax noprefix
.global _start
_start:
```

full code on your own and save as assembly_13.s

→ Writing Output:

Read syscall no. '0' | Write syscall no. '1'

write syscall also needs to specify, via it's parameters, what data to write and where to write it to.

The concept of File Descriptors (FD) we studied in Linux Luminarium: Practicing piping comes hand in hand here:

There are 3 types of FDs:

① FD0: Standard Input is the channel through which process ~~~~ takes Input. eg: shell uses Stdin to read commands that we input.

② FD1: Standard Output is the channel through which processes output normal data, such as flag when it's printed to us or output of utilities such as ls.

③ FD2: Standard error is the channel through which processes output error details. eg: If you mistype a command, shell will output, over stderror, that this command does not exist.

In programming (bash scripts) it is denoted as:

FD0: '<' Input from a file, to a file.

FD1: > or 1> output from a file or process.

FD2: 2> std error of a processes to a particular file or process.

for 'exit' we set 'rdi to '42' and 'rax to '60';
similarly for stdout, ~~we set~~ or to write to Std out, we set 'rdi to '1' and to write Stderror, we set 'rdi' to '2'.

The 'C-syntax' for 'write' syscall is:

$\underbrace{\text{write}}_{\text{syscall}}(\underbrace{\text{file\_descriptor}}_{\text{'1 or 2'}}, \underbrace{\text{memory\_address}}_{\text{from where in the memory to write}}, \underbrace{\text{number\_of\_characters\_to\_write}}_{\text{as it says character count}}$

eg: write(1, 1337000, 10);
we have to perform write of stdoutput of 10 characters from memory address '1337000'.

Now;

```
.intel_syntax noprefix
.global _start
_start:
mov rdi, 1
mov rsi, 1337000
mov rdx, 1
mov rax, 1
syscall
```

save as assembly-file.s and perform same steps as above challenges.

Here, we set 'rdi' to file_descriptor parameter of 'write' syscall i.e to '1' stdout.
then we set 'rsi' (because 'rsi' is linux favoured as 2nd parameter acceptor) to 'memory address '1337000' and 'rdx' (since rdi is used to store 1st parameter) to '1' i.e. 'character_count'
then 'rax' to '1' representing 'syscall code for 'write'
in C-syntax it will be:

write(1, 1337000, 1);

→ Chaining Syscalls:

Now, we have to chain multiple syscalls as in previous program we only give 'write' syscall and not 'exit' due to which the program does not exit cleanly, to do this we will give two syscalls now, 'write' to perform std-out from memory and 'exit' ~~to~~ to exit it cleanly.

Now;

```
.intel_syntax noprefix
.global _start
_start:
mov rdi, 1
mov, rsi, 1337000
mov, rdx, 1
mov rax, 1
syscall
mov rdi, 42
mov rax, 60
syscall
```

code for writing stdoutput as in previous challenge.

Here, '42' is stored in rdi, replacing '1', since it's already executed above and '60' in rax replacing '1', since it's already executed above and cleanly exits with code '42'.

Save as assembly-1s and perform same steps as previous challenges,

→ Writing Strings:

Now instead of a single character, we have 14 char long string stored at the memory location '1337000'.

Now;

```
.intel-syntax noprefix
.global _start
_start:
mov rdi, 1
mov rsi, 1337000
mov rdx, 14
mov rax, 1
syscall
mov rdi, 42
mov rax, 60
syscall.
```

we set char-count from '1' to '14', here, to get '14' character or '14 byte' string as output.

save as assembly-16.s and perform same steps as previous challenge.

→ Reading Data:

Now, let's first read data to memory '1337000' then write it from memory location '1337000' and then exit cleanly.

C-syntax of read:

read (file-descriptor, memory-location, char-count);

file-descriptor → '0' → for stdin

memory-location → 1337000

char-count → 8 byte data, so, '8'

Now,

```
.intel-syntax noprefix
.global _start
_start:
mov rdi, 0
mov rsi, 1337000
mov rdx, 8
mov rax, 0
syscall
mov rdi, 1
mov rsi, 1337000
mov rdx, 8
mov rax, 1
syscall
mov rdi, 42
mov rax, 60
syscall
```

read into memory '1337000' '8 byte' of data

write from memory '1337000', 8 byte of data

exit cleanly with code '42'.

save as assembly-17.s and perform same steps as previous challenges.