→ Debugging Programs:

- 'run program usin' gdb' 'r' is used to 'run' 'start' is used to apply breakpoint at 'main', 'starti' is used to apply breakpoint at '_start'. 'c' is used to continue execution of program. 'core <PATH>' is used to coredump detail of program.

  'attach <PID>' is used to attach some other already running program.

  'start' <Arg1> <Arg2> ---- can be used to pass arguments

  'breakpoint' is used to set breakpoints to '*param name'.

- Now,

  'r' to run

  'c' to contin

→ Inspecting Registers:

Now, we will learn about printing values of registers.

values of all registers → 'info registers'.

Particular register's value → 'print', or 'p' with '$ name of reg'

eg: p $rdi → value of rdi in decimal.

p/x $rdi → value of rdi in hex (x).

p/gx $rdi → value of rdi in greater hex (gx).

Now, figure out current random value of 'r12' in hex.

'run challenge'.

gdb 'r' → stops at breakpoint

gdb p/x $r12 → prints value of 'r12' in hex

copy and gdb 'c'

now paste value. →

→ Examining Memory:

Now, we will learn to use gdb to peek into process memory

We can examine contents of memory using x/<n><u><f> <address> parameterized command.

Here, <u> → unit size to display.

→ b (1 byte), h (2 bytes), w (4 bytes), g (8 bytes).

<f> → Valid formats to display.

→ d (decimal), x (hexadecimal), s (string), i (instruction).

<n> → No. of elements to display.

<address> → Address can be specified using register name, symbol name or absolute address. we can also supply mathematical expression when specifying address

- eg: x/8i $rip → Prints 8 instructions from current instruction pointer.
  x/16i main → prints first 16 instructions of main and stops.
  disassemble main or disas main → Prints all instructions of main.
  x/16gx $rsp → prints first 16 values on stack.
  x/gx $rbp-0x32 → Prints local variable stored there on stack.

We can set correct assembly syntax to intel using:
'set disassembly-flavor intel'

Now, run program and open in gdb
gdb 'r'
gdb 'c'
Now, a random value is set /dev/urandom i.e. 'read' call
gdb disas main.
look for 'read@plt' and then;
read (fd, buf, size)
↓ ↓ ↓
rdi rsi rdx
look in debugger; 'buf' is 'rsi' and 'rsi' gets 'rax'
'rax' set to 'lea rax, [rbp-0x18]'
meaning random value stored at '[rbp-0x18]
gdb x/gx $rbp-0x18
this gets a hexadecimal value
gdb 'c'
enter value

→ Setting Breakpoints:

stepi <n> & nexti <n> helps to move one instruction forward in
↓ ↓
si <n> ni <n>
Program's execution. '<n>' is useful to perform multiple steps at once but is optional.
'finish' command is used to finish execution of current function.
we can use 'break <address>' to set breakpoints manually at a required address.
'continue' or 'c' is used to continue execution until program hits a 'breakpoint'.
'display/<n><u><f>' command works exactly same format as 'x/<n><u><f>' and prints next specified no. of instructions or stack values, etc.
'layout regs' command helps put gdb into it's 'TUI' mode (Text User Interface) that shows contents of all registers as well as nearby instructions.
Now, we have to debug program to get info.

first we will open program in gdb,
then run it using 'gdb' r', then it will stop at a breakpoint, we will dissassemble it to understand the data & control flow and states of registers by using 'disas'
Then, we will look for entire code and the current breakpoint try understanding the code after break, notice a <read@plt> at an address, meaning a read operation is performed on 'rbp-0x18' at that address, so set break right after that state using 'break *main + offset';
Then read value stored or read into rbp-0x18 by using 'x/gx $rbp-0x18'; 'c' and enter the value, repeat by doing 'c' and 'x/gx $rbp-0x18' because the program loops for 'n' iterations. Done.

→ GDB Scripting:-

Scripts help automate gdb commands execution.
we can write gdb script using '.x.gdb' or '.gdb' extension and include it in gdb by using '-x <path-to-script>', we can also execute individual command from script using '-ex' <command>
we can use multiple '-ex' arguments. we can also put commands that are always executed in '~/.gdbinit'.

eg:

```
start
break *main+42
commands        <- Start of command
silent
set $local_var = *(unsigned long long*)($rbp-0x32)
printf "current value : %llx\n", $local_var
continue
end             <- end of command
continue
```

Teacher's Sign. :

'silent' indicates that gdb should not report hit to breakpoint, to make output cleaner.
'set' is used to set diff. variables in our gdb session. Then we output our variable as formatted string.

Now, we will do same thing done above, just automate it using scripting:

```
start                                          <- Start script
break *main+offset                             <- will execute commands when this offset of main is reached & breakpoint it.
commands
  silent                                       <- runs the script silently without giving tons of output
  set $secret_value = *(unsigned long long*)($rbp-0x  <- set a local variable secret_value having value of '$rbp-0x18' since read is done on it
  printf "The secret value is: %llx\n", $secret_value  <- prints value of long long variable stored above
  continue
end                                            <- end the command
continue                                       <- continue program execution.
```

→ Modifying Data:

GDB has full control over the process targeted i.e. we can not only view and understand the code & data flow but can also modify it.

we can modify state of program by using 'set' command. eg: we can use 'set $rdi = 0' to set value at 'rdi' to '0'.
we can also use 'set *((uint64_t*)$rsp)=0x1234' to set first value on stack to '0x1234'.
we can also use 'set *((uint16_t*)$0x31337000)=0x1337 to set 2 bytes at '0x31337000' to '0x1337'.
eg: suppose target reads from some socket on fd 42 and target is some networked application. maybe it would be easier for purposes of your analysis if target instead read from 'stdin'. we can achieve that using foll. gdb script:

```
start
catch syscall read
commands
  silent
  if ($rdi==42)
    set $rdi = 0
  end
  continue
end
continue
```

Now, let's create a script that takes the random value stored at memory address and loads it directly into the memory address where our user input is being stored to automatically get the flag without reading inputs.

EXPERIMENT : No. Page No. Date

start

address at which scanf is performed or user input is taken → break *main+625

commands

silent

store value at stack memory rbp-0x18 i.e. random value into local variable 'rand' this is identified by 'read@plt' from disassemble where 'read' reads the value from stack memory for comparison. → set $rand = *(unsigned long long*)($rbp-0x18)

store the random value from rand into memory location in stack where the scanned user input is stored, represented by 'mov' ---- directly after 'scanf' in disassemble. address is 'rbp-0x10' → set *(unsigned long long*)($rbp-0x10) = $rand

store address of instruction of instruction at 'mov' i.e. here main+630 in the 'rip' so that it will skip scanf address → set $rip = *main+630

continue

end

continue.

→ Modifying Execution:

Using GDB we can perform privilege escalation if the program is handing access to the 'root' or 'admin'.

Now, we will run challenge, and read description.

It says to run a function or call it in execution.

gdb call (void)win()

we get flag as we performed escalation.

→ Broken Functions:

Now, we have win() function broken as it points to a memory location that is '0x000000' i.e. '0' which does not exists and hence 'call (void)win()' does not call the win() function.

So, now lets fix it.

First let's dissass main to see if it exists or is being called in main observe output and that it does not contain or call win().

Now find 'win' if it exists in program.

gdb info function win

look for address of 'win' function.

Notice that the 'win' function's address is out of bound of 'main' meaning 'main' can't access or call win directly, so why is it happening?

let's dissass 'win'

gdb disass win

look for something suspicious such as '0' Deferencing (NULL-Deferencing) or invalid memory location.

Now let's set breakpoint to 'win'

gdb break * win

Now, lets try calling win directly to debug it.

gdb call (void) win()

Now, the break will trigger and Now, the starting address of win is stated as break and stopped or halted it's execution, so the next remaining instructions and their addresses are stored in rip.

So now, lets look for some few next instructions

gdb x/20i $rip → displays next 20 instructions to be executed from halted ones.

Notice that Program's moving '0x0' ie '0' value to stack memory with address 'rbp-0x8' and then that memory's deferenced to 'rax', which is then used to increment and call win in main.

Hence we were getting error because of 'NULL deferencing', so now lets move one step next to instructions in 'rip' until we reach the mov rax, QWORD PTR [rbp-0x8] → NULL deference instruction, this

gdb 'si' and repeat this until the memory location of this is reached.

Now cross-verify by peeking at value stored at that address.

gdb x/gx $rbp-0x8. → '0x0000000' → True hypothesis

Now lets check for valid stack memory locations which can be utilized by using: gdb x/16gx $rbp-0x10, notice some memory locations are valid and non-empty, choose any one, here I set 'rbp-0x40'

set memory address at rbp-40 to memory address of 'rbp-0x8' → gdb set {unsigned long} ($rbp-0x8) = $rbp-0x40

Set integer value to zero (optional) ← gdb set {int}($rbp-0x40) = 0

move one instruction further from current mov function ← gdb ni

continue execution ← gdb c