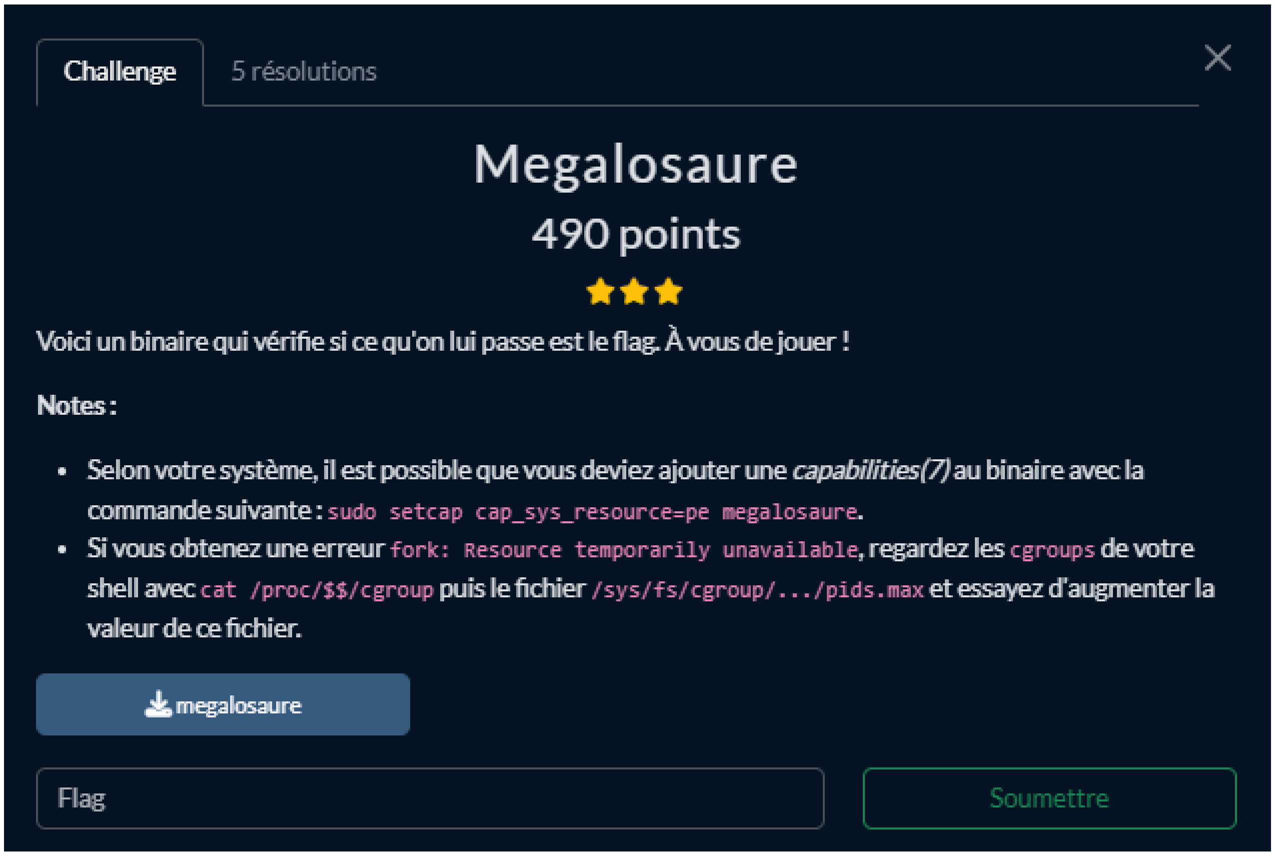
`Megalosaure` is a hard challenge (★★★) in the `reverse` category of the FCSC 2024.


Challenge
5 résolutions

Megalosaure
490 points
★★★

Voici un binaire qui vérifie si ce qu'on lui passe est le flag. À vous de jouer !

Notes :

- Selon votre système, il est possible que vous deviez ajouter une *capabilities(7)* au binaire avec la commande suivante : `sudo setcap cap_sys_resource=pe megalosaure`.
- Si vous obtenez une erreur `fork: Resource temporarily unavailable`, regardez les `cgroups` de votre shell avec `cat /proc/$$/cgroup` puis le fichier `/sys/fs/cgroup/.../pids.max` et essayez d'augmenter la valeur de ce fichier.

Gathering information

The only resource for this challenge is the binary `megalosaure` an `ELF x86-64` executable.

The challenge description hint us to add the capability `cap_sys_resource` to the binary.
With a little search on Google, enabling this capability grants unlimited access of the computer resources (no disk limit, no FD limit, ...).

This is not common, it will be for sure part of the challenge ;)

In WSL, enabling the capability has not effect, so I will run it as root.

```
$> sudo ./megalosaure
Enter the flag:
aaaaaaaaaaaaaaaaaaaa
Wrong flag format!
```

I try to run the binary and enter some password, i got `Wront format`.

Before opening the binary with a decompiler, I start it with `strace`.

```
$> sudo strace ./megalosaure
execve("./megalosaure", ["./megalosaure"], 0x7ffec8e633f0 /* 13 vars */) = 0
brk(NULL)                               = 0x5653e3a83000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)

 ... Common stuff ...

pipe2([3, 4], 0)                        = 0
pipe2([5, 6], 0)                        = 0
pipe2([7, 8], 0)                        = 0
pipe2([9, 10], 0)                       = 0
pipe2([11, 12], 0)                      = 0
pipe2([13, 14], 0)                      = 0
 ... ~ 9900 pipe2 syscall /!\

write(1, "Enter the flag:\n", 16)       = 16
```

This is a lot of pipes !

> **pipe**() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe.
>
> **Source** : https://man7.org/linux/man-pages/man2/pipe.2.html

So a pipe is simply a channel to exchange data with two ends (read end / write end).

Now, I have a few leads :

- Find the password format
- Find why there is pipes for validating a password
- And find the Flag ⚑

Reverse the binary

I open the binary in Binary Ninja.

The binary does not contain a lot of functions, but a lot of read-only data.

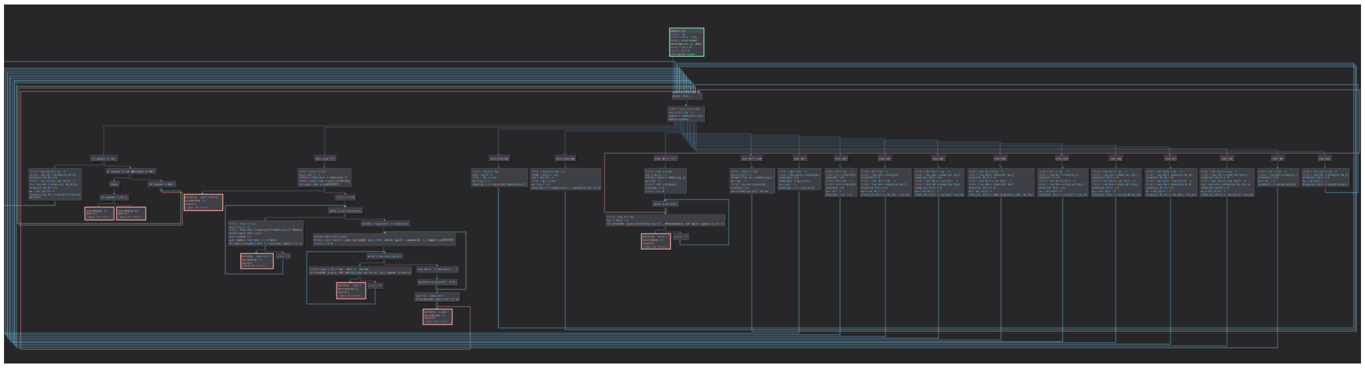
The reversing part was not very hard, pretty linear.

The main parts are :

- The main function
- Custom VM Interpreter
- Parallel VM Set Executor

Custom VM Interpreter

I immediately recognize a Custom VM interpreter at address : `0x1249`



The VM is composed of :

- a FIFO Stack : operations pop and push values
- a ROM : the ROM contains all the instructions and some operands
- a RAM : the RAM is only used as Input and Output of VMs
- 19 different opcode :

```
READ_PIPE : 0x0         DUP : 0x5         XOR : 0xa         NOT :
0xf
READ_RAM : 0x1          ADD : 0x6         AND : 0xb         NEG :
0x10
READ_ROM : 0x2          SUB : 0x7         OR : 0xc          NOP :
0x11
WRITE_PIPE : 0x3        MUL : 0x8         SHR : 0xd         EXIT :
0x12
WRITE_RAM : 0x4         MOD : 0x9         SHL : 0xe
```

Here is a pseudo code for two opcode :

```
# READ_PIPE : pseudo code
def read_pipe():
        # how many pipe to read (from the ROM)
        pipe_count = rom[ip++]

        # list of pipe IDs (from the ROM)
    pipe_ids = rom[ip:ip+pipe_count]
    ip += pipe_count

        # read all pipes and wait until all reads complete
    values = read_all_pipes_in_parallel(pipe_ids)
        stack.pushall(values)
```

```
# ADD : pseudo code
def add():
        a = stack.pop()
        b = stack.pop()
        stack.push(a + b)
```

For now, everything look like a classic VM challenge, where the VM contains all the logic to validate the password ....

.... except the pipes !

Parallel VM Set Executor

VMs are grouped by sets, they are executed in parallel and exchange data through pipes.

```
# Parallel VM Set Executor : pseudo code
def execute_vm_set(vm_set):
        for vm in vm_set:
                # spawn a new process for each vm in the set
                pid = fork()
                if pid == 0: # fork child
                        execute_vm(vm)

        wait_all_vm()
```

The main function

The main function asks the flag and validate it by chunk of 16 bytes.

```
# common RAM for all VMS
ram = [ ... ]
```

```
# A set of VM for validation the 4 chars
first4_vm_set = [ … ]

# A set of VM for each round (44)
vm_sets = [[ … ], [ … ],  … ]

def entry():
        create_all_pipes()
        flag = ask_the_flag(len = 0x70)

        # validate the 4 flag chars
        execute_vm_set(first4_vm_set)

    # check 16 bytes at a time
        for chunk in range(9):
                # write the flag chunk in RAM (as input for the VM set)
                flag_chunk = flag[chunk*8:(chunk+1)*8]
                write_flag_chunk_in_ram(flag_chunk)

                # 44 step to transform the flag
                for round in range(44):
                        execute_vm_set(vm_sets[round])

        # check the result (output of the VM set)
                result = read_the_result_in_ram()
                if result ≠ key[chunk]
                        return "Nope."

        return "Win."
```

Write a disassembler

I extracted all VM Set from the binary and write a python script to disassemble each operation in a readable format.

Here is the disassembly of the VM Set which check `flag[0:4]`.

> Note :
>
> - `ram[0]` contains `flag[0:4]`
> - `ram[1]` contains `0x1337`
> - `ram[2]` contains `0xa4e1a60a`

```
════ Worker 0 ════
stack[0] ← ram[0x0]
write 1 pipes
- pipe[0] ← stack[0]
STOP

════ Worker 1 ════
stack[0] ← ram[0x1]
write 1 pipes
- pipe[1] ← stack[0]
STOP

════ Worker 2 ════
read 1 pipes
- stack[0] ← pipe[0]
read 1 pipes
- stack[1] ← pipe[1]
stack[0] ← stack[0] * stack[1]
write 1 pipes
- pipe[2] ← stack[0]
STOP

════ Worker 3 ════
read 1 pipes
- stack[0] ← pipe[2]
stack[1] ← ram[0x2]
stack[0] ← stack[0] ^ stack[1]
write 1 pipes
- pipe[3] ← stack[0]
STOP

════ Worker 4 ════
read 1 pipes
- stack[0] ← pipe[3]
stack[1] ← dup stack[0]
stack[2] ← dup stack[1]
ram[0x2] ← stack[3]
ram[0x1] ← stack[2]
```

```
ram[0x0] ← stack[1]
STOP
```

By following the data through pipes, we get this equation :

```
(flag * 0x1337) ^ 0xa4e1a60a = 0x00000000
(flag * 0x1337) = 0xa4e1a60a
flag = 0x43534346
flag = "FCSC"
```

This seems right !

Solving the challenge

There are many ways to solve this type of challenge : write a good disassembler and search for recognizable patterns, write an automatic solver, ...

The VM has the following properties :

- Fully deterministic
- The IP (Instruction pointer) is not input-dependent (there is no conditional jumps, or arbitrary jumps)
- The Stack pointer is not input-dependent (there is no conditional push/pop)

This is the perfect conditions for symbolic execution !

Writing a symbolic executor

We could use a generic symbolic executor like `angr`, who auto-magically lift assembler and solve the whole binary. But I suspect that `angr` would not well support `forks` and `pipes`, ...

So lets write our own with `Z3` !

The input symbolic values are each chunk of 4 bytes of the `flag`. (`flag_0`, `flag_1`, `flag_2`, ...)

The `Z3` API provides `BitVectors` (`BV`) with fixed-size, when adding two `BV`, we get back another `BV`. Behind the scenes, the resulting `BV` is a tree of operations (similar as an AST).

So the code of a symbolic executor is very close of a concrete executor.

```
# ADD : Concrete executor
def add():
        a: int = stack.pop()
        b: int = stack.pop()
        c: int = a + b
        stack.push(a + b)

# ADD : Symbolic executor
def add():
        a: z3.BV = stack.pop()
        b: z3.BV = stack.pop()
        c: z3.BVAdd = a + b
        stack.push(c)
```

After the symbolic execution, we get back a model, which is a **large** set of **very large** equations mapping inputs (flag) to outputs.

`Z3` supports a lot of different theories : `BitVectors` , `Arrays` , `Regex` , `Floats` , `Sequences` , ...
For counterpart, `Z3` is not fastest solver for specific models, our model contains only `BitVector` so i used this solver instead : `boolector` .

```
$> time boolector smt_model -m
unsat


________________________________________________________
Executed in    1.19 secs    fish           external
   usr time    1.13 secs    0.00 micros    1.13 secs
   sys time    0.06 secs  422.00 micros    0.06 secs
```

In only 1.2 seconds, `boolector` proves ... that there is NO flag that matches the model.
There is an bug somewhere ...

Debugging the symbolic executor

I copy-paste my symbolic executor and transform it in a concrete executor to compare with the real binary.

At each round of the first iteration, I printed the output of both.

```
Real binary :
- Iter 0 Round 0 : 2461343629
- Iter 0 Round 1 : 2579746662
- Iter 0 Round 2 :  ...
```

```
My executor :
- Iter 0 Round 0 : 2461343629
- Iter 0 Round 1 : 2579746658 /!\
- Iter 0 Round 2 :  ...
```

There is only a difference of 4, the most probable error is a semantic mismatch of some opcode between the symbolic executor and the real binary.

Since writing through a pipe is done by a syscall, I used `strace` to dump all values written in pipes.

```
$> sudo strace -xx -f -e write ./megalosaure 2> all_writes.txt
```

After filtering and comparing, I found that the semantic of the right-shift is different between `C` and `Z3`.

```
C  : 0b1000 >> 2 = 0b0010  (right logical shift)
Z3 : 0b1000 >> 2 = 0b1110  (right arithmetic shift)
```

The correct `Z3` operator is `LShR()`

Get the flag !

After the change, the symbolic executor output a valid model.

```
$> time boolector smt_final

sat
(
  (define-fun flag_0 () (_ BitVec 32) #b00110100001101010011010001111011)
  (define-fun flag_1 () (_ BitVec 32) #b00110010001100110110010000110010)
  (define-fun flag_2 () (_ BitVec 32) #b01100011001101110011001001100101)
  (define-fun flag_3 () (_ BitVec 32) #b01100100001100100110010101100100)
  (define-fun flag_4 () (_ BitVec 32) #b00110111011001100110001100110111)
  (define-fun flag_5 () (_ BitVec 32) #b00110001001101000011001101100100)
  (define-fun flag_6 () (_ BitVec 32) #b01100001001101110110010100111000)
  (define-fun flag_7 () (_ BitVec 32) #b00110110001100010011000000110100)
  (define-fun flag_8 () (_ BitVec 32) #b00110110001101010011000101100010)
  (define-fun flag_9 () (_ BitVec 32) #b01100110011001000011010001100101)
  (define-fun flag_10 () (_ BitVec 32) #b01100011001100000011001100110111)
  (define-fun flag_11 () (_ BitVec 32) #b00110101011000010011011000110000)
  (define-fun flag_12 () (_ BitVec 32) #b01100010001100010011000100110110)
  (define-fun flag_13 () (_ BitVec 32) #b01100101001110000011000101100011)
  (define-fun flag_14 () (_ BitVec 32) #b00110010001100110110010001100100)
  (define-fun flag_15 () (_ BitVec 32) #b01100001001100000011001000110100)
  (define-fun flag_16 () (_ BitVec 32) #b00000000000000000111110101100011)
)


________________________________________________________
Executed in   29.86 secs    fish           external
   usr time   29.65 secs  154.00 micros   29.65 secs
   sys time    0.18 secs  180.00 micros    0.18 secs
```

Concatening the binary string and fixing the endianness gives the flag !