# So Above

## Task

* Category: The Magician
* Points: 165
* Solves: 19

Description:

No matter how far a path you blaze, there you are.

The answers are in the binaries.

https://so-above-soo2epha.eames.satellitesabove.me

container\_name=$(docker create --rm --env "FLAG=${flag}" --env "SEED=${seed}" $image\_name) docker cp "$submission\_path" "$container\_name:/submission.ts" docker start --attach "$container\_name"

Attachement:

* template.ts, a template for a Deno script
* so-above-20230331.docker.bz2, a docker image export

## Solution

Having a look at the driver.rb shipped with the docker image we can see that a bunch of challenges are dynamically created and tested agains Deno running our submission in a restricted environment:

dest = File.join \_\_dir\_\_, 'src', 'quick\_maths.c'  
  
NUMBER\_TO\_GENERATE.times do |n|  
 out\_f = File.open dest, 'w'  
  
 out\_f.puts <<~EOS.strip  
 #include <stdbool.h>  
  
 bool quick\_maths(double run) {  
 EOS  
  
 want = rand(0..UINT32\_MAX)  
 run = want  
  
 rand(10..20).times do  
 stmt = nil  
 result = nil  
  
 loop do  
 # skipping multiplication for now; idiv vs. fdiv ;\_;  
 operation = %i{+ - /}.sample   
 operand = num\_in\_range(operation)  
  
 result = run.send operation, operand  
  
 # next here loops again  
 next if result >= UINT32\_MAX  
 next if result <= 0  
  
 stmt = "run = run #{operation} #{operand};"  
 stmt += " // #{result}" if ENV['CHALLENGE\_DEV\_DEBUG']  
 break  
 end  
  
 out\_f.puts stmt  
 run = result  
 end  
  
  
 out\_f.puts <<~EOS.strip  
 return (run == #{ run });  
 }  
 EOS  
  
 out\_f.close  
  
 `make clean all`  
  
  
 size = `wc -c build/ominous\_etude`.split[0].to\_i  
 digest = `sha256sum build/ominous\_etude`.split[0]  
  
 new\_omen\_name = "generated"  
 FileUtils.mv 'build/ominous\_etude', "challs/#{new\_omen\_name}"  
 FileUtils.mv dest, "challs/#{new\_omen\_name}.c"  
  
 $stderr.puts JSON.dump({  
 'sha256' => digest,  
 'size' => size,  
 'answer' => want.to\_s  
 })  
  
 got = nil  
  
 IO.popen("deno run --cached-only --allow-read=/chall/challs/generated,/deno-dir --allow-env=SOLVER\_DEBUG /submission.ts #{new\_omen\_name}",   
 'r+',  
 # should redirect child stderr to my stdout  
 :err => :out) do |line|  
 got = line.gets  
 line.close  
 end   
  
 puts got  
 puts got.to\_i == want  
  
 IO.popen("/chall/challs/#{new\_omen\_name}", 'w+', :err => :out) do |result|  
 result.puts got  
 puts result.gets  
 puts did\_get = result.gets.strip  
 result.close  
 unless "cool :)" == did\_get   
 puts result.puts  
 puts "got a wrong answer"  
 exit 1  
 end  
 end  
end

The above ruby scripts generated a series of floating point operations and compiles a binary, the operations look something like the following:

bool quick\_maths(double run) {  
run = run - 254; // 3153546112  
run = run + 12157; // 3153558269  
run = run + 11229; // 3153569498  
run = run + 10156; // 3153579654  
run = run + 8342; // 3153587996  
run = run + 137; // 3153588133  
run = run / 15; // 210239208  
run = run + 10389; // 210249597  
run = run + 12961; // 210262558  
run = run + 5540; // 210268098  
run = run - 142; // 210267956  
return (run == 210267956);  
}

We need to supply the correct input number such that quick\_maths returns true.

Having a look at submission.ts we can tell that we have been provided with some modules:

// deno-lint-ignore-file prefer-const  
import iced from "npm:iced-x86@1.18.0"  
import elfinfo from "npm:elfinfo@0.4.0-beta"  
import predicates from "npm:@tool-belt/type-predicates@1.2.2"

This is looking good so far, we can also read the binary, but not the source code, so lets start by parsing the elf and extracting quick\_maths from the symbols:

const file\_data = await Deno.readFile("/chall/challs/generated") // Deno.readFile("generated")  
const info = await elfinfo.open(file\_data)  
const qmSymbol = info.elf.sections.filter(x => x.name == ".symtab")[0].symbols.filter(x => x.name == "quick\_maths")[0]  
  
function getBytes(addr, len) {  
 return file\_data.slice(addr, addr + len)  
}  
  
let funcData = file\_data.slice(parseInt(qmSymbol.value), parseInt(qmSymbol.value) + qmSymbol.size)

But what now? Well looking at the assembly we can figure out that the function always has the same structure:

00000000000011c9 <quick\_maths>:  
 11c9: 55 push %rbp  
 11ca: 48 89 e5 mov %rsp,%rbp  
 11cd: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 11d2: f2 0f 10 4d f8 movsd -0x8(%rbp),%xmm1  
 11d7: f2 0f 10 05 61 0e 00 movsd 0xe61(%rip),%xmm0 # 2040 <\_IO\_stdin\_used+0x40>  
 11de: 00   
 11df: f2 0f 58 c1 addsd %xmm1,%xmm0  
 11e3: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 11e8: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 11ed: f2 0f 10 0d 53 0e 00 movsd 0xe53(%rip),%xmm1 # 2048 <\_IO\_stdin\_used+0x48>  
 11f4: 00   
 11f5: f2 0f 5c c1 subsd %xmm1,%xmm0  
 11f9: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 11fe: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 1203: f2 0f 10 0d 45 0e 00 movsd 0xe45(%rip),%xmm1 # 2050 <\_IO\_stdin\_used+0x50>  
 120a: 00   
 120b: f2 0f 5c c1 subsd %xmm1,%xmm0  
 120f: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 1214: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 1219: f2 0f 10 0d 37 0e 00 movsd 0xe37(%rip),%xmm1 # 2058 <\_IO\_stdin\_used+0x58>  
 1220: 00   
 1221: f2 0f 5e c1 divsd %xmm1,%xmm0  
 1225: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 122a: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 122f: f2 0f 10 0d 29 0e 00 movsd 0xe29(%rip),%xmm1 # 2060 <\_IO\_stdin\_used+0x60>  
 1236: 00   
 1237: f2 0f 5c c1 subsd %xmm1,%xmm0  
 123b: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 1240: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 1245: f2 0f 10 0d 1b 0e 00 movsd 0xe1b(%rip),%xmm1 # 2068 <\_IO\_stdin\_used+0x68>  
 124c: 00   
 124d: f2 0f 5e c1 divsd %xmm1,%xmm0  
 1251: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 1256: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 125b: f2 0f 10 0d 0d 0e 00 movsd 0xe0d(%rip),%xmm1 # 2070 <\_IO\_stdin\_used+0x70>  
 1262: 00   
 1263: f2 0f 5e c1 divsd %xmm1,%xmm0  
 1267: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 126c: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 1271: f2 0f 10 0d ff 0d 00 movsd 0xdff(%rip),%xmm1 # 2078 <\_IO\_stdin\_used+0x78>  
 1278: 00   
 1279: f2 0f 5c c1 subsd %xmm1,%xmm0  
 127d: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 1282: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 1287: f2 0f 10 0d f1 0d 00 movsd 0xdf1(%rip),%xmm1 # 2080 <\_IO\_stdin\_used+0x80>  
 128e: 00   
 128f: f2 0f 5c c1 subsd %xmm1,%xmm0  
 1293: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 1298: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 129d: f2 0f 10 0d e3 0d 00 movsd 0xde3(%rip),%xmm1 # 2088 <\_IO\_stdin\_used+0x88>  
 12a4: 00   
 12a5: f2 0f 5e c1 divsd %xmm1,%xmm0  
 12a9: f2 0f 11 45 f8 movsd %xmm0,-0x8(%rbp)  
 12ae: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 12b3: 66 0f 2e 05 d5 0d 00 ucomisd 0xdd5(%rip),%xmm0 # 2090 <\_IO\_stdin\_used+0x90>  
 12ba: 00   
 12bb: 0f 9b c0 setnp %al  
 12be: ba 00 00 00 00 mov $0x0,%edx  
 12c3: f2 0f 10 45 f8 movsd -0x8(%rbp),%xmm0  
 12c8: 66 0f 2e 05 c0 0d 00 ucomisd 0xdc0(%rip),%xmm0 # 2090 <\_IO\_stdin\_used+0x90>  
 12cf: 00   
 12d0: 0f 45 c2 cmovne %edx,%eax  
 12d3: 5d pop %rbp  
 12d4: c3 ret  
 12d5: 66 2e 0f 1f 84 00 00 cs nopw 0x0(%rax,%rax,1)  
 12dc: 00 00 00   
 12df: 90 nop

We can use this to extract the operations from the disassembly:

let decoder = new iced.Decoder(64, funcData)  
decoder.ip = qmSymbol.value  
  
let instructions = decoder.decodeAll()  
let formatter = new iced.Formatter()  
  
let parsed: any[] = []  
for (let i = 2; i < instructions.length - 11; i += 4) {  
 let dataAddr = formatter.format(instructions[i + 2]).split(" ")[1].split(",")[0]  
 let data = getBytes(Number(dataAddr), 8)  
  
 parsed.push([  
 (new Buffer(data)).readDoubleLE(),  
 formatter.format(instructions[i + 3]).split(" ")[0]  
 ])  
}  
  
let targetAddr = formatter.format(instructions[instructions.length - 4]).split(" ")[1].split(",")[0]  
let target = (new Buffer(getBytes(Number(targetAddr), 8))).readDoubleLE()

So now that we have all the data we need all we have to do is reverse the operations and print the number:

parsed.reverse().forEach(e => {  
 switch (e[1]) {  
 case "addsd":  
 target -= e[0]  
 break;  
  
 case "subsd":  
 target += e[0]  
 break;  
  
 case "divsd":  
 target \*= e[0]  
 break;  
  
 default:  
 throw ":("  
 }  
})  
console.log(BigInt(target))

And that’s all there is to it, a rather short solve, yielding us the flag: flag{november958844india4:GErixL2HwXYaZp4a9KD9taPbeJohPmuBNqtTcKFIxBPLR5gocn9SM5jUnhwkmLK7bUhbBNCrEDqW0miE7kBg\_rQ}